UNIVERSITÉ DU QUÉBEC À MONTRÉAL

MÉTHODES BIO-INFORMATIQUES BASÉES SUR LES FLUX DE TRAVAUX POUR LE SÉQUENÇAGE À HAUT DÉBIT.

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

JÉRÉMY GOIMARD

JUIN 2018

REMERCIEMENTS

"Plus on partage, plus on possède. Voilà le miracle." [Léonard Nimoy]

Lors de la réalisation d'une maîtrise, c'est le partage qui est la principale source du travail de recherche. C'est un apprentissage intense et régulier basé sur l'échange et le travail collaboratif.

Je tiens à remercier mon directeur de mémoire, le Pr Abdoulaye Baniré Diallo pour sa patience face à mes nombreuses questions. Je le remercie également pour sa vision de la recherche, son implication et son soutien.

Je remercie également Étienne Lord pour sa disponibilité, ses réponses rapides et ses propositions toujours productives. Je tiens également à remercier tous les étudiants du laboratoire de bio-informatique de l'UQAM avec qui j'ai eu l'occasion de travailler et d'échanger.

Mes remerciements se portent aussi vers Calcul Québec pour l'accès aux clusters.

Je remercie également mes proches pour leur soutien dans ma démarche.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ACRONYMES

ADN acide désoxyribonucléique. 3

API application programming interface. 11, 27

CPU central processing unit. 22

EV environnement virtuel. v, vi, ix, xi, xv, 21–23, 25–27, 35, 36, 54

KVM Kernel-based Virtual Machine. 21

LXC LinuX Container. 23, 24

LXD LinuX Deamon. 24

SHD séquençage à haut débit. v–vii, ix, xi, xv, 1–6, 8, 12–15, 24, 26, 30, 34–37, 47, 52, 54, 56, 58–62, 67, 74, 77, 85–87, 95

SNPs single nucleotide polymorphisms. 67, 70, 77

VDI virtual disk image. 20

VHD virtual hard disk. 20

VMDK VMware Virtual Machine Disk Format. 20

VPH virtualisation par hôte. v, ix, xi, 16, 19, 20, 24–26

RÉSUMÉ

En biologie, l'analyse de données issues du séquençage à haut débit (SHD) se compose d'un ensemble de programmes informatiques dont l'assemblage forme un flux de travaux représentant les étapes d'analyse, programme après programme. Ces flux de travaux en bio-informatique sont gérés par des plateformes logicielles polyvalentes qui permettent d'accéder à des programmes et de les assembler sans avoir de connaissances particulières en informatique.

Cependant, une grande part des programmes utilisés dans les flux fonctionnent avec leurs propres types de données. Ils nécessitent souvent de nombreuses dépendances liées à des librairies ou d'autres programmes et ne supportent souvent que le système d'exploitation Linux. Une solution à ce problème est la virtualisation d'applications préinstallées sur de nombreux systèmes d'exploitation, telle que proposée par l'environnement virtuel (EV), ou conteneur. À la fois souple et léger, il permet l'automatisation et le transfert des programmes sur de nombreux supports. L'EV fournit un excellent moyen de répéter des expériences.

En raison de la taille des données et la complexité des tâches informatiques, les programmes d'analyse des SHD sont exigeants en ressources matérielles. Pour pallier ce problème, le calcul distribué, quel qu'en soit le niveau de complexité, permettrait l'accès à des ressources matérielles importantes. Il pourrait ainsi répondre aux besoins de traitement des données issues du SHD.

En tant que plateforme logicielle de flux de travaux, Armadillo 1.1 répond à une grande partie des attentes. Afin d'y ajouter de nouveaux programmes d'analyse de SHD, ce mémoire propose des solutions pour automatiser l'ajout de nouveaux programmes et de faciliter leur mise à jour. Il propose également un environnement virtuel basé sur Docker qui permet l'utilisation de ces programmes sur l'ensemble des systèmes d'exploitation. Enfin, ce travail explore la possibilité d'exécuter des flux de travaux à la fois localement et sur des calculs distribués.

Mots Clés *flux de travaux, bio-informatique, séquençage haut débit, environnement virtuel, Docker, calcul distribué, cluster, Armadillo*

INTRODUCTION

La facilité d'accès et la prolifération de données issues du séquençage à haut débit (SHD) amènent de plus en plus de biologistes à utiliser ces données afin de développer de nouvelles approches et de nouvelles méthodes analytiques. À cette fin, des outils informatiques performants et adaptés sont nécessaires pour examiner la quantité massive de données biologiques issues du SHD. Ces outils se composent souvent d'un ensemble de programmes informatiques dont l'agencement et les connexions forment les flux de travaux.

Une grande partie des programmes proposés pour analyser les données issues du SHD sont développés pour des systèmes d'exploitation de type Linux. Afin de les utiliser dans d'autres systèmes d'exploitation, la virtualisation complète et l'environnement virtuel sont deux approches possibles. La virtualisation complète est une solution relativement simple, mais pouvant demander d'importantes ressources pour le système hôte. De son côté, l'environnement virtuel, récemment devenu plus accessible, est plus léger et plus facilement automatisable.

D'autre part, l'analyse de ces données a un fort coût computationnel (consommation processeur et mémoire vive, temps d'exécution, stockage des données). Effectuée sur des machines locales, l'analyse peut prendre de nombreuses heures et limite l'utilisation de celles-ci à d'autres fins. Pour remédier à ce problème, la parallélisation et le calcul distribué présentent des conditions de développement intéressantes et offrent l'opportunité d'effectuer ces tâches computationnelles plus rapidement grâce à l'accès à de nombreuses ressources matérielles.

À travers cette recherche, notre objectif est de développer et de tester l'hypothèse

d'un outil de gestion de flux de travaux multiplateformes pour les scientifiques. Celui-ci permettra une automatisation du traitement des données par des programmes de SHD intégrés. Les flux de travaux seront disponibles localement ou exécutables au travers du système de calcul distribué. Le chapitre 1 de ce mémoire introduira les problématiques et besoins reliés à la recherche en biologie et leurs réponses informatiques. Cette partie comporte un rappel de l'historique du SHD, ainsi qu'une présentation des flux de travaux existants actuellement et des programmes associés avec leurs points faibles et leurs points forts. Le chapitre 2 s'intéresse plus particulièrement aux systèmes de virtualisation ainsi qu'aux avantages et limites du calcul distribué. Le chapitre 3 portera sur les améliorations apportées au logiciel Armadillo afin de répondre à nos objectifs ainsi que sur les résultats obtenus.

CHAPITRE I

BIOLOGIE ET BIO-INFORMATIQUE

1.1 Introduction

"La vie, ce concept mystérieux, est ramenée à la présence d'acide désoxyribonucléique. Il n'y a plus de frontière entre matière animée et inanimée. Tout n'est qu'une question de degré de complexité." (Albert Jacquard).

Le séquençage de l'acide désoxyribonucléique (ADN) a connu des avancés majeures dans les années 2000. Depuis, les techniques n'ont cessé d'évoluer, passant du séquençage de l'ADN nucléotidique à celui de l'intégralité de la cellule (Liu *et al.*, 2012). En nous basant sur l'évolution du séquençage, nous présenterons une définition du séquençage à haut débit (SHD) ainsi que ses liens étroits avec la bio-informatique. Nous présenterons les flux de travaux et leurs plateformes logicielles en bio-informatique. La dernière partie portera sur les défis quant à l'intégration des programmes traitants de SHD dans les plateformes de flux de travaux.

1.2 Le séquençage à haut débit (SHD)

1.2.1 Définition, histoire et lien avec la bio-informatique

Le séquençage classique s'est développé autour du séquençage de l'ADN nucléique. Il demande beaucoup de temps, de main d'œuvre et d'étapes avant l'obtention des données. Les méthodes utilisées ont été améliorées lors du projet de séquençage

du génome humain (van Dijk *et al.*, 2014).
Comme le présente Liu *et al.* (2012) dans son article, le séquençage à haut débit (*High-Throughput Sequencing (HTS)* ou *Next Generation Sequencing (NGS)*) est un ensemble de méthodes apparues depuis 2005, produisant, à faibles coûts et rapidement, des millions de séquences en s'affranchissent des étapes classiques de clonage. Ces méthodes utilisent des approches parallèles permettant le séquençage de nombreux fragments simultanément. Bien que le développement initial de ces technologies soit principalement dédié à l'humain (pour la détection de maladies, de cancers, etc.), on trouve également des usages dans l'analyse des végétaux qui possèdent souvent des génomes bien plus complexes. Ces technologies sont, dans tous les cas, génératrices de nombreuses données qui nécessitent des outils de traitement informatique pour en extraire les résultats et les rendre interprétables par les scientifiques.

1.2.2 Les défis du SHD

Bien que certains formats, comme *fastafile* ou *samfile*, soient bien décrits et normalisés comme on le retrouve dans les articles de Li *et al.* (2009) et Lipman et Pearson (1985), il existe presque autant de formats que de programmes SHD. Cette variabilité est due au fait que les formats répondent aux besoins des scientifiques et aux questions biologiques auxquelles ils souhaitent répondre. Les scientifiques les créent en fonction des besoins et des résultats qu'ils souhaitent obtenir. Les formats de fichiers sont donc de formats très variables.

De plus, le séquençage à haut débit, quelle que soit la génération, utilise des méthodes génératrices d'un très grand nombre de données. L'utilisation de fichiers de très grande taille pose des difficultés pour la gestion de ces données tant dans leur sauvegarde que les transferts. De même, le traitement de ces fichiers sur une seule machine est souvent problématique et il est souvent préférable d'utiliser les

avancées technologiques telles que la parallélisation et le calcul haute performance.

Enfin, une autre limitation des programmes de SHD est la gestion des dépendances vers des librairies spécifiques ou d'autres programmes. Ces interconnexions rendent l'installation de certains programmes fastidieuse ou nécessitent de bonnes connaissances dans l'infrastructure tant logicielle que matérielle. C'est un des points les plus problématiques pour l'acquisition et la mise à disposition des programmes, en vue de répéter d'un flux de travail comme le montre Gerlach *et al.* (2015) et Collberg et Proebsting (2016) dans leurs articles. Or, cette répétabilité est une base importante de la bio-informatique. C'est dans cet objectif que des outils de flux de travaux dédiés à la bio-informatique ont vu le jour.

1.3 Les plateformes de flux de travaux en bio-informatique

1.3.1 Les flux de travaux

Il existe de nombreux logiciels permettant d'analyser, de comparer et de transformer les résultats obtenus par les outils de séquençages. Ainsi, un scientifique souhaitant répondre à une questions de biologie, va souvent utiliser un ensemble de logiciels les uns à la suite des autres. Cet enchaînement d'outils peut être considéré comme un flux de travaux dans lequel transitent, parfois conditionnellement, les données d'un outil à l'autre.

Les flux de travaux se décomposent en deux grands modèles : le contrôle de flux (*control-flow*) et le flux par les données (*data-flow*). Le premier détermine l'agencement des programmes avant leur exécution. Il est souvent utilisé dans le cadre de flux de travaux d'affaires. Tandis que le second se base sur la disponibilité des données d'une étape à l'autre et est plus souvent utilisé dans le cadre de flux de travaux en science (Migliorini *et al.*, 2011). Ludäscher *et al.* (2009) présente les flux de travaux scientifiques comme "un ensemble de tâches individuelles organi-

sées lors de la conception du flux". L'exécution de celui-ci est orchestrée selon le flux de données et les tâches dépendantes spécifiées par le concepteur. Le flux de travail est souvent conçu de manière visuelle.

L'enchaînement des outils peut poser des problèmes, car il existe une multitude de formats différents d'entrées et de sortie selon les programmes. La sortie de l'un n'est donc pas forcément l'entrée de l'autre et les fichiers (ou données) ont parfois besoin d'être transformés. Des plateformes rassemblant ces outils ont été créées afin de faciliter la communication entre les outils, leur utilisation, la répétition des expériences et la création d'expérience *in silico* (Lord *et al.*, 2012).

1.3.2 Les pipelines

Les pipelines sont une forme réduite de flux de travail permettant de répondre efficacement à une ou plusieurs questions biologiques précises. Ils proposent généralement un enchaînement de plusieurs outils dans une seule direction. Les étapes doivent être résolues les unes après les autres pour atteindre l'objectif. Les pipelines sont de conception simple dans la mesure ou ils ne permettent pas de boucles conditionnelles ou l'ajout d'autres programmes que ceux initialement présents sans en modifier le code. La majorité des programmes proposés nécessitent un apprentissage des formats des fichiers permettant la configuration des pipelines.

1.3.2.1 Quelques logiciels de pipelines

Les logiciels de pipelines sont aussi nombreux que les question biologiques nécessitant l'intervention linéaire d'un certain nombre de programmes. La sélection de pipelines suivante s'appuie sur ceux utilisant Dockers, étant au code source ouvert et orienté vers la résolution de problèmes biologiques impliquant les SHD.

Snakemake créé en 2012 (Köster et Rahmann) est un logiciel de flux de travaux

en Python. S'appuyant sur un flux de travail facilement lisible, il peut s'exécuter à différentes échelles depuis la station de travail classique jusqu'au Cluster. Il a également pour objectif de réduire la complexité de création d'un flux de travail dans un langage de spécifications et règles proches du style de langage Python. Les règles définissent également la création des fichiers de sortie en fonction des fichiers d'entrée.

Nextflow (Nextflow, 2017) est disponible depuis 2013 pour les systèmes d'exploitation de type POSIX. Il est écrit en JAVA et Groovy. Ce logiciel permet la création rapide de pipelines par l'ajout de "tâches" sans avoir besoin de connaissances en programmation. Il supporte les technologies de conteneurs Docker ou Singularity [1], et le partage des pipelines pour une meilleure reproductibilité des pipelines. Il est portable par son langage qui apporte une couche d'abstraction entre la logique du pipeline et son exécution. Il est également utilisable à différentes échelles du calcul distribué à l'infonuagique. La parallélisation est de type flux de données déclarées lors de la création du pipeline. Nextflow propose également un suivi du pipeline au cours de son exécution.

Genome Analysis Toolkit ou GATK (Bahassi et Stambrook, 2014) est un logiciel développé au Broad Institute pour analyser les données de séquençage à haut débit. Sa boîte à outils offre une grande variété de logiciels mettant l'accent sur la découverte de variantes et le génotypage, ainsi que sur l'assurance de la qualité des données. Ces outils ont été principalement conçus pour traiter les exomes et les génomes entiers générés avec la technologie de séquençage Illumina. À l'origine développé pour la génétique humaine, GATK est capable de traiter les données génomiques de différents organismes, et ce, avec différents niveaux de

1. Singularity container (https ://singularity.lbl.gov/) Singularity is a container solution created by necessity for scientific and application driven workloads.

ploïdie. Écrit en Java, la version 4 (2017) est sous licence open source [2] BSD 3. Ces outils proposent des flux de travaux comme base commune de bonnes pratiques pour certains types d'analyses [3].

Durant l'année 2015, Folarin *et al.* proposent **NGSeasy**. Très récent, ce logiciel propose des pipelines de quelques outils classiques de SHD encapsulés dans Docker. Entièrement sous Docker, il ne propose pas une interface utilisateur et ne semble pas facile d'utilisation. Néanmoins, cette intégration complète le rend disponible pour tous les systèmes d'exploitation.

La même année, Belmann *et al.* (2015) soumettent **Bioboxes** qui a pour objectif de faciliter le transfert d'expérimentation à travers la standardisation d'interface de conteneur. Tout comme NGSeasy, c'est un programme essentiellement en ligne de commande qui s'éloigne des standards pour la création de flux de travail pour des non-initiés. De même, bien que l'ajout d'un nouveau programme se passe entièrement sous Docker, la création des conteneurs peut rester complexe.

Encore en 2015, **TOGGLE**, pour 'toolbox for generic NGS analyses' Monat *et al.* (2015) et écrit en Perl, est une suite d'outils capables d'être utilisés sous la forme de pipelines. Les fichiers de configuration des pipelines utilisés sont simples et permettent l'accès aux options des programmes utilisés. TOGGLE permet de suivre l'évolution du pipeline et des données. TOGGLE est utilisable à différentes échelles depuis une simple machine à du calcul distribué. Il ne présente pas d'interface utilisateur et reste sous la forme d'un terminal en ligne de commande.

En 2017, **Cluster Flow** (P. Ewels et Andrews), écrit en Perl, propose 40 modules et une simple syntaxe pour les assembler. C'est un logiciel de pipeline dédié aux

2. Licence GATK4 (https ://github.com/broadinstitute/gatk/blob/master/LICENSE.TXT)

3. Exemple : Germline short variant discovery (SNPs + Indels) (https ://software.broadinstitute.org/gatk/best-practices/workflow ?id=11145)

grilles de calcul, cluster et l'infonuagique. Il n'est pas prévu pour être utilisé au niveau local et se présente sous la forme de ligne de commande.

1.3.3 Les plateformes

Les plateformes ont en général pour objectif de faciliter la construction, la transmission et la répétition de flux de travaux. Elles intègrent des interfaces d'accès aux options de différents programmes et facilitent les interactions entre les programmes. Dans un premier temps, nous verrons les caractéristiques des plateformes, puis nous présenterons différents logiciels qui répondent à ces critères.

1.3.3.1 Les caractéristiques des plateformes de flux de travaux

Voici ci-dessous, quelques points qui résument les caractéristiques définissant les plateformes de flux de travaux en bio-informatique :

- Exécuter des flux de travail abstraits afin de privilégier l'utilisateur dans son interaction et sa personnalisation de l'interface (souvent à l'aide de techniques graphiques) (Woollard *et al.*, 2008).
- Intégrer l'analyse et la visualisation des données tout en simplifiant la compréhension et la cognition de la procédure (Woollard *et al.*, 2008).
- Aider les utilisateurs à trouver des composants, des flux de travail et des ensembles de données sur la base des caractéristiques ou fonctions souhaitées (Woollard *et al.*, 2008).
- Favoriser la possibilité d'extension pour que l'intégration de futurs services et logiciels dont les interfaces et les protocoles de communication n'affecteront pas les autres composants déjà fonctionnels (Lin *et al.*, 2009).
- Fournir le code permettant d'assurer la reproductibilité dans les résultats pour éviter la dissémination de résultats erronés (Collberg et Proebsting, 2016).

— Fournir le support pour la surveillance de l'état du flux de travail et des défaillances ainsi qu'un mécanisme pour attraper, localiser et gérer les défaillances automatiquement ou avec une intervention humaine minimale (Lin *et al.*, 2009).
— Valider les flux de travaux nouvellement créés en respectant les exigences et les contraintes des composants et d'un ensemble de données hétérogènes tout en gérant la collection, l'organisation et la réorganisation des données (ex. transformation vers d'autres formats) (Collberg et Proebsting, 2016).
— Offrir le support pour les systèmes de calcul distribué ou tout autre système informatique dématérialisé (comme l'infonuagique) (Lin *et al.*, 2009).
— Faciliter l'évolution et la gestion des versions des workflows précédemment créés (Woollard *et al.*, 2008) et faciliter une interconnexion avec d'autres systèmes (ex. d'autres flux de travaux) (Lin *et al.*, 2009).

1.3.3.2 Les systèmes de gestion des flux de travaux

Il existe plus d'une vingtaine de plateformes permettant d'effectuer des flux de travaux en bio-informatique parmi les plus connues, on peut citer Taverna ou Galaxy. Nous présenterons quelques-unes des plateformes qui nous semblent répondre aux critères énoncés dans la section 1.3.3.1, qui sont en code source ouvert et qui sont encore maintenus.

Taverna, créée en 2006, est une plateforme issue de plusieurs plateformes scientifiques et dont l'objectif est la gestion de flux de travaux. Elle est écrite en Java et se présente sous de nombreuses formes, que ce soit pour des applications locales ou distantes (Hull *et al.*, 2006). Les flux de travaux issus de Taverna sont souvent complexes. Ainsi, plusieurs ajouts ont été développés tels que DistillFlow (Chen *et al.*, 2014) ou BIFI (Yildiz *et al.*, 2014) dont les objectifs sont entre autres de simplifier ces flux.

La plateforme **Galaxy** est née en 2010. C'est une plateforme de services web installables localement ou sur un serveur Unix ou Mac. Elle est écrite en Python et JavaScript. Cette plateforme propose un très grand éventail de capsules pour des logiciels de bio-informatiques. Elle comprend également une grande communauté et de nombreux tutoriels. Elle fait partie des premiers logiciels de flux de travaux utilisés en bio-informatique. Néanmoins, une bonne installation et un bon fonctionnement de cette plateforme ne sont pas faciles (Goecks *et al.*, 2010). Ayant conscience de cette difficulté, un effort important de ces dernières années a permis la séparation du coeur de Galaxy (gestion du flux de travail, interface, etc.) et l'émergence de Toolshed comme installateur de programmes (Blankenberg *et al.*, 2014). En 2014, Docker a été introduit pour l'installation et l'utilisation de programme dans Galaxy (Galaxy-Docker, 2016).

En 2012, Okonechnikov *et al.* proposent **UGENE** qui inclue un flux de travail visuel pouvant être effectué localement ou dans un environnement de calcul distribué. UGENE est un logiciel multiplateforme pour la bio-informatique. Il est écrit en C++ soutenu par Qt un cadriciel permettant le développement d'applications en multiplateformes. Il se base sur le concept d'application programming interface pour les échanges. Les logiciels externes qu'il propose ont également besoin d'être multiplateformes et compatibles avec les différentes versions pour fonctionner limitant l'intégration de nouveaux logiciels. L'utilisation du *workflow designer* permet la création de flux de travaux simples.

Armadillo voit également le jour en 2012 avec une version publique disponible 1.1. C'est une plateforme de flux dédiée aussi bien à l'analyse phylogénétique qu'à des analyses classiques de bio-informatiques. Il est écrit en Java et propose des capsules pour de nombreux programmes multiplateformes. Il gère aussi bien le flux de données que le contrôle de flux. Il peut être utilisé en ligne de commande et installé sur des serveurs. Il est donc possible de décrire son flux de travail

localement et de l'exporter ensuite. La gestion des flux est un des points importants de développement (Lord *et al.*, 2012). L'ajout de nouveaux programmes intégré dans Armadillo passe généralement par un programmeur qui propose et compose l'interface. Actuellement, ce travail n'est pas homogène et présente des difficultés en cas de mise à jour d'un programme.

BioDepot-workflow-Builder (BwB) Hung *et al.* (2017) est accessible depuis 2017. Il est basé sur Orange 3 de Biolab et NoVnc. Ses éléments sont créés en Python et utilisent les librairies QT5, Docker-Py et PyQt5. Il propose la création de flux de travaux par glissement et ajout à l'aide d'une interface utilisateur simple. Les composants des flux de travaux sont uniquement des conteneurs Docker accessibles depuis des dépôts publics ou fournis par l'utilisateur.

1.3.3.3 Comparatif de plateformes de flux de travaux

D'après les caractéristiques citées dans la section 1.3.3.1 et dans l'objectif de comparer les plateformes présentées dans la section 1.3.3.2, nous avons extrait et reconstruit les critères de différenciations suivants :

- Installer le logiciel de manière totalement opérationnelle (ITO).
- Ajouter de nouveaux programmes (ANP)
- Créer un flux de travail et son accessibilité côté utilisateur (FT).
- Supporter de multiples systèmes d'exploitation, Linux, Mac, Windows (MSE).
- Utiliser localement, sur un serveur ou un système de calcul distribué (Usage - Local,Server,CD (calcul distribué))
- Rendre portable sur un système calcul distribué (Port).
- Contenir des programmes SHD (SHD).
- Utiliser et l'intégrer la gestion de conteneur (Cont) [4].

Nous avons comparé les plateformes logicielles citées précédemment dans le ta-

4. N.B. Les aspects portabilités et gestion de conteneurs seront présentés dans le chapitre 2.

bleau comparatif 1.1. Les valeurs que l'on y retrouve sont les suivantes :

— Facile (F) / Partiellement Facile (P) / Difficile (D)

— Facile d'utilisation (FU) / Pas Facile d'utilisation (PFU)

— Non applicable ou sans information (-)

— Ligne de commande (CLI) / Interface Graphique pour l'utilisateur (GUI)

— Uniquement (U) / Pas uniquement (PU)

— Nécessite une instance spécifique (S) / Est incluse dans le logiciel de base (I) / N'est pas inclus dans le logiciel de base (NI)

Tableau 1.1: Comparaison des logiciels de flux de travaux

	ITO	ANP	FT	MSE	Usage			Port	SHD	Cont
					Local	Server	CD			
Snakemake	P	H	-	yes	Cli	Cli	Cli	I	U	-
NGSeasy	P	H	-	yes	Cli	-	Cli	I	U	I
Bioboxes	P	H	NFU	yes	Cli	-	Cli	I	U	I
Galaxy	P	D	FU	non	-	GUI	GUI	S	PU	S
Ugene	F	D	FU	P	G	CLI	CLI	I	U	NI
Taverna	F	P	NFU	oui	G	CLI	CLI	S	PU	NI
BwB	P	P	NFU	oui	G	-	-	I	U	I
Armadillo1.1	F	P	FU	oui	G	CLI	-	-	PU	NI
Objectif	**F**	**F**	**FU**	**oui**	**G**	**CLI**	**CLI**	**I**	**PU**	**I**

Par exemple pour les programmes *Taverna* et *Objectif*, le tableau 1.1 se lit de la manière suivante :

Le logiciel *Taverna* :

— a une installation facile,

— a une installation partielle de nouveaux programmes,

— propose une interface utilisateur pas toujours facile d'utilisation,

— supporte de nombreux systèmes d'exploitation avec l'aide d'une instance spécifique,

— peut être exécuté localement ou sur un serveur en ligne de commande,

— propose nativement des programmes SHD et d'autres programmes de bio-

informatique,

— peut être exécuté nativement sur un système de calcul distribué,

— ne propose pas de déploiement d'applications via des conteneurs.

Tandis que le logiciel "*Objectif*" :

— a une installation facile,

— a une installation facile de nouveaux programmes,

— propose une interface utilisateur facile d'utilisation,

— supporte de nombreux systèmes d'exploitation sans restriction,

— peut être exécuté localement ou sur un serveur et sur un système de calcul distribué en ligne de commande,

— propose nativement des programmes SHD et d'autres programmes de bio-informatique,

— peut être exécuté nativement sur un système de calcul distribué,

— propose de déploiement d'applications par l'intermédiaire de conteneurs.

Comme nous le présentons dans le tableau 1.1, aucune des plateformes existantes actuellement ne répond entièrement aux besoins d'utilisateurs de programmes SHD. Nous avons introduit dans le tableau un logiciel *Objectif* tel qu'il répondrait positivement à l'ensemble de nos critères. Il semble que le programme *Armadillo1.1* soit le plus proche de notre objectif. Nous verrons dans le chapitre 3 quelles ont été améliorations apportées sur ce logiciel *Armadillo1.1* pour atteindre le logiciel *Objectif*.

1.4 Conclusion

Dans ce chapitre, nous avons vu l'importance de l'informatique dans la résolution de problèmes biologiques dans le domaine des SHD. Nous avons également vu qu'il était possible d'utiliser des plateformes de flux de travaux tout en abordant la virtualisation et le calcul distribué. Ces derniers n'ayant pas été définis, nous les présenterons dans le chapitre suivant.

CHAPITRE II

INFORMATIQUE, LE SHD EN MULTI-SYSTÈMES

2.1 Introduction

Dans ce chapitre, nous aborderons les problématiques relatives à l'informatique. Nous présenterons des solutions appropriées dans le but d'atteindre les objectifs de programmation fixés dans le chapitre 1. Nous nous intéresserons tout d'abord à la virtualisation qui répond à l'objectif d'utilisation multisystèmes au travers de la virtualisation par hôtes et l'environnement virtuel. Par la suite, nous porterons notre intérêt sur le calcul distribué et le parallélisme dans le but de mieux comprendre quels en sont ses avantages, ses contraintes et ses liens potentiels avec le SHD. Cette dernière partie présentera également quelques *clusters* de Calcul Québec.

2.2 La virtualisation

2.2.1 Introduction

La virtualisation consiste à faire fonctionner différents environnements, systèmes d'exploitation ou applications, invités sur une machine hôte. Elle apporte plusieurs bénéfices, car elle permet d'utiliser un autre système d'exploitation et ses logiciels sans avoir à redémarrer son ordinateur. Elle fournit l'opportunité de tester des nouveaux systèmes d'exploitation ou programmes ne fonctionnant pas nativement

dans le système d'exploitation hôte sans déstabiliser son environnement quotidien. La virtualisation fournit une zone de tests pour de nouveaux logiciels dans des environnements contrôlés, isolés et sécurisés. Enfin, elle facilite le partage des systèmes d'exploitation d'un ordinateur à l'autre en disposant d'un hyperviseur compatible (Scott et Irvine, 2000).

2.2.2 La virtualisation par hôte

2.2.2.1 Définition

Nous prendrons appui sur l'article de Popek et Goldberg (1974) pour définir les différents types de virtualisation. Selon les auteurs, le moniteur de machine virtuelle, ou hyperviseur est une couche applicative qui permet l'exécution de machines virtuelles. Il existe deux types de virtualisation par hyperviseur : la native (*"bare-metal"*) sur un système hôte, et celle par l'intermédiaire d'un système d'exploitation hôte (*"host-based"*). Ces deux systèmes de virtualisation permettent le déploiement de systèmes d'exploitation dît "invités" qui utilisent les ressources mises à disposition par leurs hôtes.

La virtualisation native se base sur un hyperviseur de type 1 qui met directement en contact le matériel hôte avec les machines virtuelles. Elle est souvent utilisée pour le déploiement des serveurs multitâches et a été développée dans ce sens. Ce mode de virtualisation reste difficilement exploitable sur des machines locales. Schématisée dans la figure 2.1, la première couche (en gris clair) est celle du matériel hôte sur quel se repose l'hyperviseur (en vert). Les couches suivantes sont les machines virtuelles qui se composent du système d'exploitation et des applications.

L'hyperviseur de type 2 correspond au système de virtualisation par hôte (VPH). La virtualisation sur un système d'exploitation hôte permet le déploiement de

différents services via des systèmes invités. Elle facilite l'intégration de systèmes d'exploitation différents ou de besoins et services plus diversifiés. Le système invité est présent sur la machine hôte sous la forme de fichiers qui peuvent être facilement déplacés, copiés ou dupliqués. Il existe différentes techniques de virtualisation par hôte.

La figure 2.2 schématise la virtualisation par hôte. La sous-figure 2.2a représente la virtualisation complète et la sous-figure 2.2b la paravirtualisation. De la base vers le haut, on retrouve d'abord les éléments présents sur l'hôte. Ils se composent du matériel du système hôte (en gris clair), du système d'exploitation hôte (en bleu), de l'hyperviseur (en vert) et du matériel émulé par l'hyperviseur (en gris clair). Puis, on retrouve les systèmes invités composés également d'un système d'exploitation et d'applications.

Sous la forme complète, l'hyperviseur instaure des environnements clos, isolés, avec des ressources précises et disponibles issues de la machine hôte. Dans ce cadre, l'invité n'a accès qu'au matériel virtuel proposé par l'hyperviseur et repose donc intégralement sur ce dernier. Sous la forme de paravirtualisation (2.2b), l'hyperviseur met directement en contact la machine invitée avec le matériel de l'hôte. Il joue alors un rôle de passeur en interceptant les informations de l'invité et en les transmettant à l'hôte. Cette approche demande au système hôte de prendre "conscience" de la présence du système invité. Cela nécessite la mise à disposition de pilotes (*driver*) spécifiques, rarement développés pour des virtualisations sur des machines locales. La forme paravirtualisation est plus souvent utilisée sur des serveurs où la recherche de gains de performances est importante ce qui est rarement le cas pour les virtualisations locales.

Bien qu'il existe de nombreuses formes de virtualisation, la solution de virtualisation complète par hôte est bien répandue en bio-informatique comme le montre

Field *et al.* (2006) dans son article. En effet, ce type de virtualisation peut répondre aux besoins de réplications des expérimentations et à la diffusion des outils. Il existe ainsi des solutions complètes de suite de logiciels préinstallés sur des systèmes d'exploitation (ex. *Biolinux*, *Scientific Linux* ou *Fedora Scientific*).

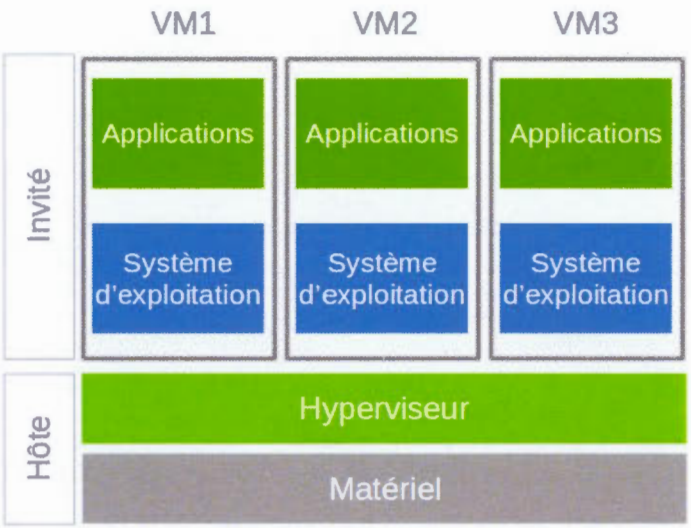
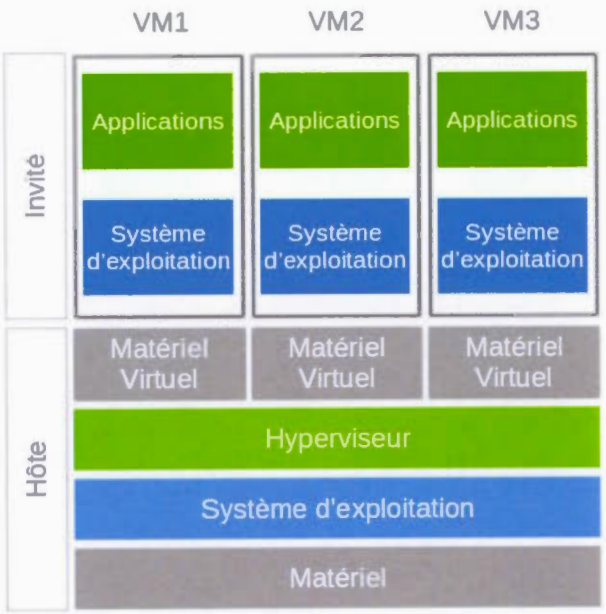
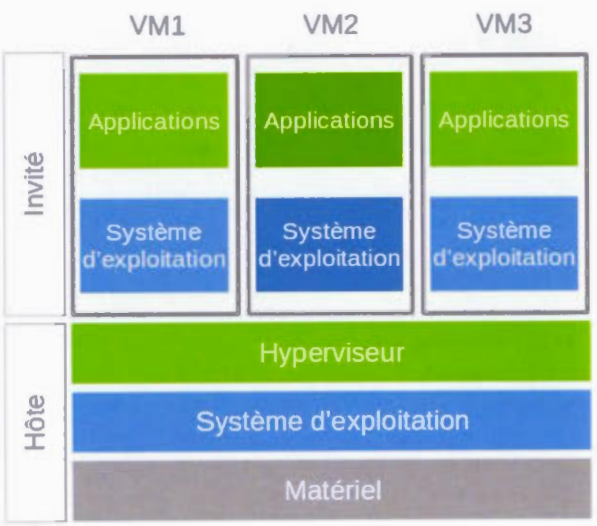


Figure 2.1: Hyperviseur type 1 (Native)



(a) Virtualisation complète



(b) Paravirtualisation

Figure 2.2: virtualisation par hôte (VPH)

2.2.2.2 Les logiciels de VPH

La virtualisation par hôte nécessite donc un logiciel ayant le rôle d'hyperviseur entre l'hôte et l'invité. Il s'agit de la partie la plus importante de la virtualisation. Nous présenterons ci-dessous les programmes les plus connus avec leurs avantages et leurs inconvénients. Ils permettent la création d'un environnement virtualisé sur des postes de travail et sur des serveurs.

Oracle VM Virtualbox est un logiciel de virtualisation, sous différentes licences selon les produits, qui permet d'installer et d'utiliser rapidement de nombreuses distributions. Il est souvent utilisé pour tester l'installation de nouveaux programmes. D'après Oracle (Oracle VM VirtualBox, 2017), Oracle VM Virtualbox permet également de prendre en charge rapidement les versions d'un programme selon un système d'exploitation particulier. Ce logiciel supporte de nombreuses formes de machines virtuelles créées à partir d'autres hyperviseurs. Il peut également fonctionner avec les virtualisations machines de type Intel VT-x et AMD-V et propose également un mode paravirtualisation. Oracle VM Virtualbox peut utiliser différents formats d'images de disque tel que virtual disk image (VDI), VMware VMware Virtual Machine Disk Format (VMDK) ou encore le format Microsoft Virtual PC virtual hard disk (VHD). Enfin, il est possible de créer des instantanés des machines en cours d'utilisation.

VMWare est une suite de logiciels propriétaires ((VMware, 2017)). Ses versions "poste de travail" aussi appelées *VMWare Workstation*, *VMware Player* et *VMware Fusion* pour Mac, oscillent entre une virtualisation complète et une paravirtualisation selon les besoins de la machine invitée et l'offre de la machine hôte. VMWare souhaite ainsi offrir de meilleures performances que ses concurrents tout en gardant une sécurité maximale. Ses versions serveur sont *VMware ESXi* (une solution de type native).

KVM est une infrastructure de virtualisation du noyau Linux. Elle offre nativement une virtualisation des types Intel VT-x et AMD-V. Le mode paravirtualisation est accessible pour les machines invitées Linux et Windows en utilisant le cadre VirtIO. Celui-ci inclut la paravirtualisation de la carte Éthernet, l'accès aux disques en lecture écriture, l'ajustement des requêtes invitées pour l'usage de la mémoire et l'interface graphique (via des pilotes de VMWare) (Guedes *et al.*, 2014; Kivity *et al.*, 2007).

2.2.3 L'environnement virtuel

En parallèle à la virtualisation complète, on trouve également l'environnement virtuel (EV). Nous définirons celui-ci dans un premier temps, puis nous présenterons différents logiciels de cette famille d'outils de virtualisation. Enfin, nous nous attarderons sur le logiciel de gestion de conteneurs nommé Docker.

2.2.3.1 Définition

Dans leurs articles respectifs Guedes *et al.* (2014) et Merkel (2014) présentent l'environnement virtuel, conteneur ou encore *Operating system-level virtualization* comme une méthode de virtualisation pour laquelle le noyau du système d'exploitation autorise l'existence de plusieurs instances isolées. Ces instances possèdent différentes appellations comme **conteneurs** (*containers*), ou **conteneurs de logiciels** (*software containers*). Le système invité possède son propre espace mémoire isolé, mais partage l'ensemble des autres ressources avec l'hôte. Le conteneur permet donc d'alléger la combinaison hôte/invité et de faciliter l'accès au matériel hôte. Toutefois, les ressources partagées complexifient l'isolation entre les systèmes invités et dans la relation hôte à invité. Les instructions qu'il émet n'étant pas interceptées, le système invité doit impérativement être compatible avec le système hôte.

Essentiellement développé sous Linux et connu depuis très longtemps, ce dernier a effectué un grand retour grâce à la simplification de l'accès aux ressources du noyau Linux jusque-là réservé aux experts. Ainsi, l'environnement virtuel Linux se base principalement sur deux composantes du noyau Linux.

Le **gestionnaire de groupes** (*cgroups* ou *control groups*) contrôle les ressources disponibles pour les conteneurs (Anderson, 2015). Il gère également les limitations et les priorités d'accès aux ressources (central processing unit (CPU), mémoire, block I/O, le réseau, etc.) (Linux Kernel Organization, 2017).

Le **Linux *namespace*** est une fonctionnalité du noyau qui permet l'isolation et la virtualisation de ressources du système pour des processus. L'isolation porte sur les environnements d'exploitation incluant les hiérarchies de procédés, le réseau, les identifiants utilisateurs, les noms d'hôtes (*hostnames*) et les points de montages des fichiers systèmes (Biederman et Networx, 2006; Namespaces, 2017).

Contrairement aux machines virtuelles, l'environnement virtuel n'a besoin de posséder qu'une partie d'un système d'exploitation pour exécuter directement une application (Bernstein, 2014). La figure 2.3 schématise le fonctionnement d'un environnement virtuel. Dans celle-ci, on retrouve la partie matérielle (en gris clair) à la base de la figure, puis le système d'exploitation hôte (en bleu). La troisième couche (en vert) est celle du logiciel qui gère les conteneurs présents dans la dernière couche sous la forme d'applications cloisonnées dans des conteneurs (Xavier *et al.*, 2013). Le système d'exploitation des conteneurs étant très léger, il est très rarement représenté.

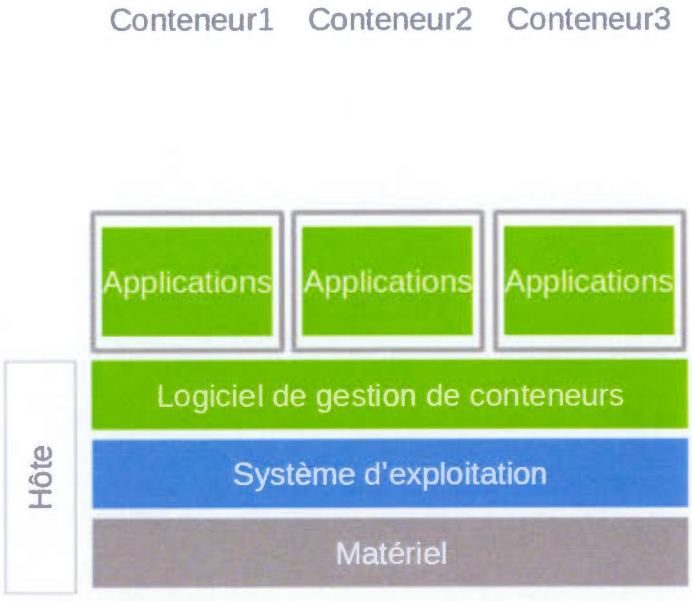


Figure 2.3: environnement virtuel (EV) ou Conteneur

2.2.3.2 Les logiciels de gestion de l'environnement virtuel

Les logiciels présentés ci-dessous ont été sélectionnés sur la base de critères reliés à la gestion des flux de travaux. Ils doivent notamment permettre de créer des environnements virtuels sur des machines locales et sur le calcul distribué, être présents, si possible, sous de multiples systèmes d'exploitation (Linux, Mac, Windows), rester libre avec un code source ouvert et être activement maintenu. Les logiciels qui correspondent à ces critères sont les suivants :

Créé en 2008, le **conteneur Linux (*LinuX Container*), LXC** utilise les fonctionnalités de groupe de contrôle (*"cgroup"*) et l'isolation par nom (*"isolation namespace"*) pour mettre en place des conteneurs d'applications ou de systèmes qui sont entièrement isolés et contrôlés (LXC, 2017). Uniquement sous Linux, le logiciel LXC est souvent considéré comme étant entre un *chroot* (commande per-

mettant l'isolation d'une application au sein de Linux) et une VPH. LXC crée un environnement aussi proche que possible d'une installation Linux standard, mais sans avoir besoin d'un noyau distinct. Dans ce système, le logiciel *lxc-tool*, un *chroot* grandement amélioré, permet le contrôle des conteneurs.

Docker existe depuis 2013. Selon Docker Company (2016), il est une plateforme ouverte pour les développeurs et les administrateurs systèmes souhaitant construire, exporter et lancer des applications distribuées à travers des ordinateurs, des centres de données, ou sur l'infonuagique. Tout comme LXD, il présente l'avantage d'être exploitable sur l'ensemble des systèmes d'exploitation. Ce logiciel est orienté vers la mise en conteneur d'applications et supporte de nombreux systèmes de conteneur comme LXC.
Docker présente une grande maturité, possède une grande communauté d'utilisateurs et une très bonne documentation. Il répond parfaitement au besoin de notre objectif pour mettre en place des programmes de SHD au sein de plateformes de flux de travaux comme présenté dans le tableau 1.1. Nous présenterons plus précisément ce logiciel dans la partie 2.2.5.

Depuis févier 2015, **LXD (*LinuX Deamon*)** est un gestionnaire de conteneurs (*container hypervisor*) et peut aussi démarrer des machines virtuelles. Il présente l'avantage de gérer toute sorte de conteneurs de type Linux, mais a été spécifiquement conçu pour la gestion de conteneurs LXC. Il se présente comme le successeur de LXC. Il peut être utilisé conjointement avec Docker.
Selon les concepteurs de LXD celui-ci sécurise par son design et par de nombreux outils de contrôle les conteneurs qu'il orchestre. Il est capable de s'adapter en fonction de l'offre et la demande matérielle en particulier sur l'infonuagique. Il est intuitif grâce à une interface de programmation applicative (API) et des lignes des commandes simples et claires. Il se base sur des images pour la diffusion de ses machines virtuelles. Enfin, il permet également la migration en direct des conte-

neurs (LCD, 2017). Il est accessible depuis 2017 pour les systèmes d'exploitation Windows et Mac.

2.2.4 Comparaison entre la Virtualisation Par Hôte (VPH) et l'Environnement Virtuel (EV)

Nous avons comparé la VPH et l'EV selon des critères proposés dans les articles Felter *et al.* (2015) et Soltesz *et al.* (2007). Ces neuf critères vont de la mise en place de la virtualisation jusqu'à une mise à l'échelle (comme le déploiement sur l'infonuagique) en passant par les aspects d'accès aux machines et aux programmes qu'elles contiennent. Dans le tableau 2.1 et pour qualifier les points, nous utiliserons une échelle allant de --- (très très négatif) à +++ (très très positif) en passant par 0 (neutre ou référence).

Tableau 2.1: Comparaison entre la virtualisation par hôte (VPH) et l'environnement virtuel (EV)

	VPH	EV
Mise en place	0	++
Automatisation	0	+++
Mise à l'échelle, portabilité	0	+++
Accès aux ressources hôtes	--	+++
Sécurité par défaut	++	-
Facilité d'accès aux programmes ou applications de la machine invitée depuis la machine hôte	-	++
Partage des fichiers entre hôte et invité	-	+++
Partage des outils	--	+++
Poids des machines invitées	---	++
Notation : --- (très très négatif) à +++ (très très positif) en passant par 0 (neutre ou référence)		

Bien que la mise en place d'une VPH puisse être facilitée par des outils comme Vagrant ou virt-manager, un déploiement d'une instance EV reste plus rapide et facile (Joy, 2015). De plus, l'automatisation des procédés de lancement d'une

instance EV favorise le transfert entre plateformes facilitant la migration et traitant automatiquement les changements de configurations nécessaires ce qui est plus difficile dans le cas d'une VPH. De même et surtout pour l'EV, on peut noter que l'automatisation d'une installation permet de maximiser la portabilité sur les différentes plateformes. Une bonne gestion de la répartition des capacités informatiques incluant l'aspect portabilité des conteneurs signifie qu'ils peuvent s'exécuter sur un certain nombre de plateformes différentes localement, ou sur l'infonuagique. Cela donne lieu à une optimisation en fonction des coûts et des performances (Felter *et al.*, 2015).

L'EV permet de réduire la complexité de virtualisation. En effet, les conteneurs ne nécessitent pas de dépendances applicatives complexes sur l'infrastructure hôte. Autrement dit, il n'est pas nécessaire d'utiliser une interface native complexe pour traiter les services de la machine invitée.

En revanche, l'aspect sécurité est souvent le point fort de la VPH car celle-ci est souvent mal comprise, mise en place et plus complexe dans sa gestion pour l'EV.

Depuis la machine hôte, l'accessibilité des programmes contenus dans la machine invitée est beaucoup plus facile depuis l'EV que depuis une VPH. Ceci est principalement dû à l'appel de commande optimisée pour le logiciel de gestion de conteneur via l'API et plus complexe pour l'hyperviseur qui gère une VPH.

Pour conclure sur la virtualisation, notre approche d'intégration des programmes de SHD utilisant des conteneurs est confortée par notre comparaison entre la virtualisation par hôte et l'environnement virtuel.

2.2.5 Le logiciel Docker

Dans cette section, nous présenterons le logiciel Docker comme outil de gestion de conteneurs.

Docker est un programme au code source ouvert proposant un ensemble d'outils qui permettent l'installation d'applications Linux. Ce logiciel permet la virtualisation de conteneurs par l'intermédiaire de machines virtuelles. Il utilise le noyau Linux ainsi qu'une interface de programmation applicative (*RESTful API*). Le noyau et l'application programming interface (API) permettent d'exécuter simultanément plusieurs processus. Ceux-ci sont reliés aux CPU, mémoires, réseaux et opérations d'entrées sorties. Docker est donc capable d'utiliser les ressources de la machine hôte en tant qu'environnement virtuel (Ludvigh *et al.*, 2015). Docker assigne les services en fonction d'un ordre de priorité. Il a également pour objectif de minimiser les coûts d'opérations de calcul et d'accès aux ressources.

Docker utilise une architecture de type client-serveur. Le client Docker parle au daemon Docker, qui effectue les tâches de construction, d'exécution et de distribution des conteneurs Docker. Le client et le daemon Docker peuvent s'exécuter sur le même système ou sur un système distant. L'interface de ligne de commande utilise le Docker REST API pour contrôler ou interagir avec le daemon Docker à l'aide de scripts ou de commandes CLI directes. De nombreuses autres applications Docker utilisent l'API et l'interface de ligne de commande sous-jacente. C'est sur cette architecture que se base de Docker Engine présenté dans le schéma 2.4[1].

1. (Source : `https://docs.docker.com/engine/docker-overview/#docker-engine`)

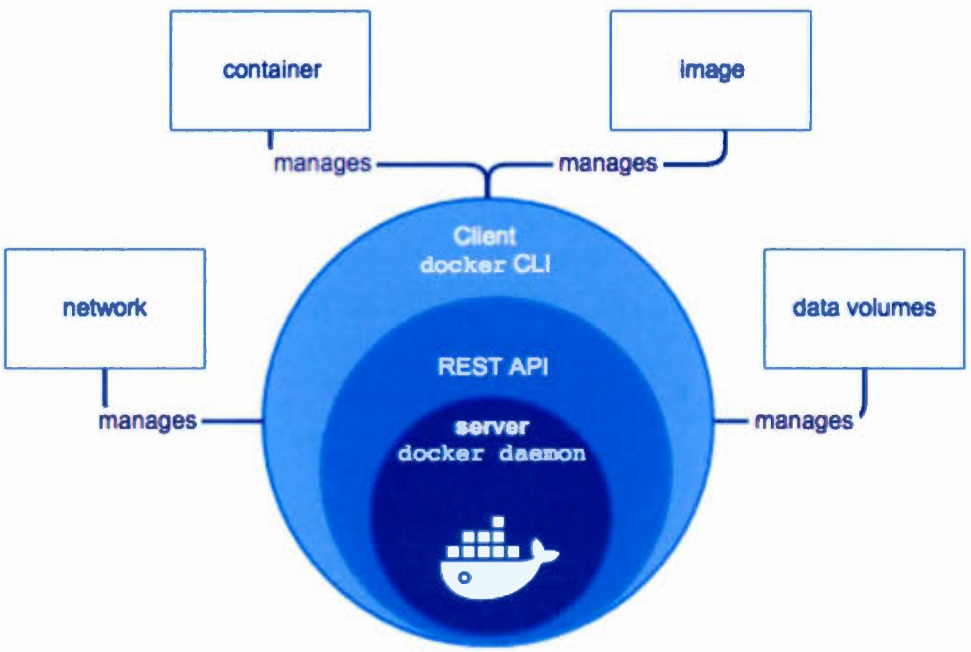


Figure 2.4: Docker Engine

Docker est écrit en langage Go et utilise dans son système principal la librairie *libcontainer*. Cette librairie permet aux conteneurs de travailler avec les espaces de noms Linux *namespaces*, les groupes de contrôle *cGroups*, les fonctionnalités, AppArmor, les profils de sécurité, les interfaces réseau et les règles de pare-feu de manière cohérente et prévisible. Le groupe de contrôle (*cGroups*) permet aux conteneurs l'accès aux ressources physiques du système hôte. Docker utilise le service *Union File Système* (ou *UnionFS*) du système de fichiers en conjonction avec des techniques d'écriture en copie comme la base de construction des fichiers pour les conteneurs, et ce, afin de les alléger et de les rendre plus rapides (Docker Glossary, 2017). La librairie *libcontainer* permet également de gérer le cycle de vie du conteneur en effectuant des opérations supplémentaires après la création

du conteneur.

Docker utilise une image de base des principaux systèmes d'exploitation de type Linux. Le procédé de création de nouvelles images est automatisé et s'appuie sur un fichier nommé *Dockerfile*. C'est un document de type texte qui contient toutes les commandes qui auraient été exécutées manuellement dans l'objectif de créer une nouvelle image Docker.

L'image initiale est modifiée par les instructions du *Dockerfile* en ajoutant une nouvelle couche à la précédente, faisant évoluer l'image de base vers une image finale. L'avantage de cette approche est l'empilement des conteneurs tel que présenté dans la figure 2.5. En effet, lorsque Docker construit une nouvelle image, il s'appuie sur les images déjà présentes si cela est possible.

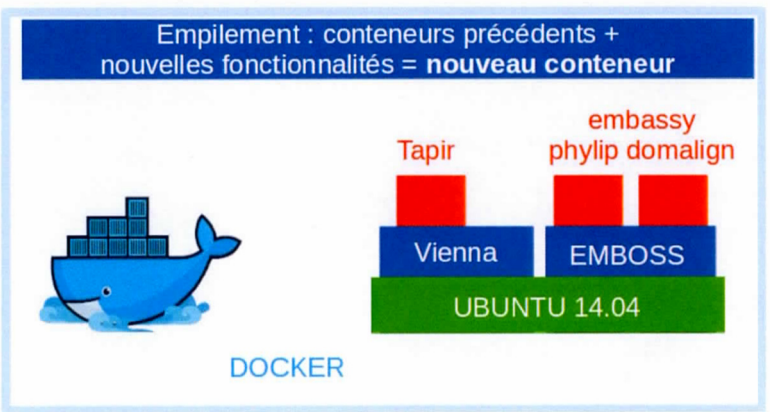


Figure 2.5: Empilement des conteneurs de Docker

L'intégralité de ces actions, formalisées et simplifiées, permet d'effectuer des tests et un déploiement rapide. Il est ensuite possible de partager ce fichier par l'intermédiaire du centre Docker (*Docker hub*) de manière publique ou privée. Le *Docker deamon* se sert de l'image produite par le fichier *Dockerfile* comme base pour le lancement d'un ou plusieurs conteneurs.

Le fichier *Dockerfile* est aussi utilisable quelque soit la plateforme à partir du *Docker deamon* qui est disponible sous différentes versions accessibles nativement ou non pour les plateformes Mac ou Windows. La version la plus commune et la plus ancienne est *Docker Toolbox*. Cet outil passe par Virtualbox pour effectuer la virtualisation du noyau Linux sous ces systèmes. Il utilise donc un hyperviseur de type 2 dans un premier temps. De récentes versions de Docker sont natives pour les Mac utilisant OS X El Capitain ou encore plus récemment pour Windows 10 dont le système supporte Microsoft Hyper-V (Docker Company, 2016).

2.3 La parallélisation et le calcul distribué

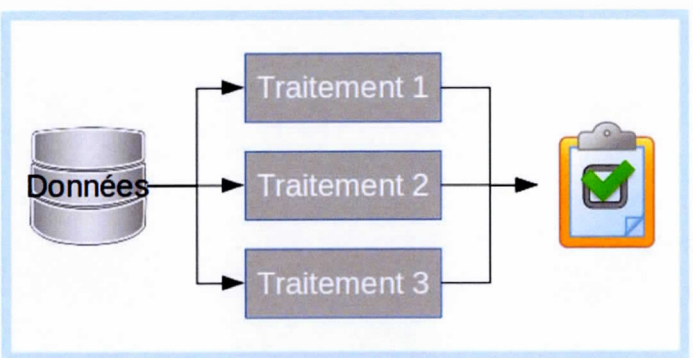
"Ensemble de techniques logicielles et matérielles permettant l'exécution simultanée de séquences d'instructions indépendantes, sur des processeurs différents" Guy Moebs - 2010.

La parallélisation et le calcul distribué permettent, d'une part, un gain de temps en terme de restitution des résultats grâce à la distribution du travail lors de l'exécution d'un programme, et, d'autre part, la résolution de problèmes plus gros grâce à l'accès à de nombreuses ressources matérielles (notamment les mémoires *RAM ROM*, *CPU* ou *GPU*). Or, les analyses des données issues du SHD demandent souvent des résolutions de problèmes importants dans des temps raisonnables, ce qui place la parallélisation et le calcul distribué comme des outils privilégiés.

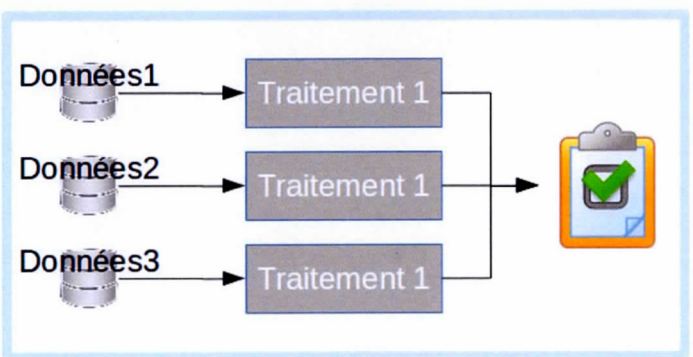
Cette section abordera une définition de la parallélisation et du calcul distribué suivi de son lien avec le SHD et les conteneurs. La dernière partie exposera brièvement les possibilités d'accès aux serveurs de Calcul Québec.

2.3.1 Définitions

Le calcul distribué représente simplement le fait de répartir un calcul ou un traitement sur plusieurs microprocesseurs (Coulouris *et al.*, 2011). Tandis qu'il existe de nombreuses formes de parallélisation. Parmi les types principaux de parallélisations, on retrouve celles de type logiciel et matériel. Dans les parallélisations logicielles, on retrouve deux grandes approches schématisées dans la figure 2.6 : la parallélisation par tâches et la parallélisation des données. La parallélisation par tâches (figure : 2.6a) effectue des tâches en parallèle sur le même type de données. À l'opposé, la parallélisation par données (figure : 2.6b) effectue la même tâche, mais sur des données différentes.



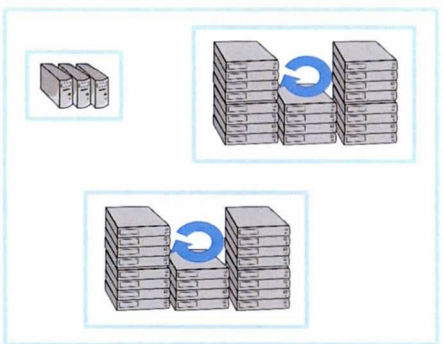
(a) Parallélisation par tâches



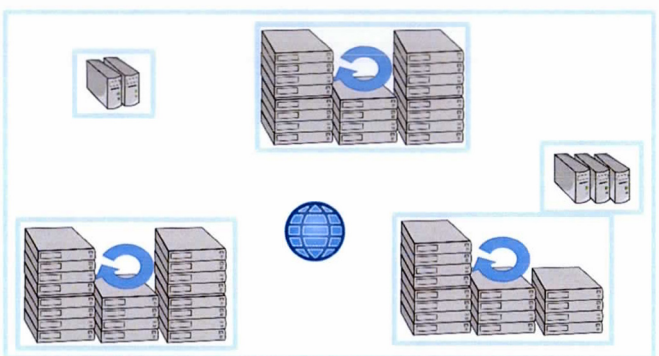
(b) Parallélisme par données

Figure 2.6: Exemples de parallélisation logicielle

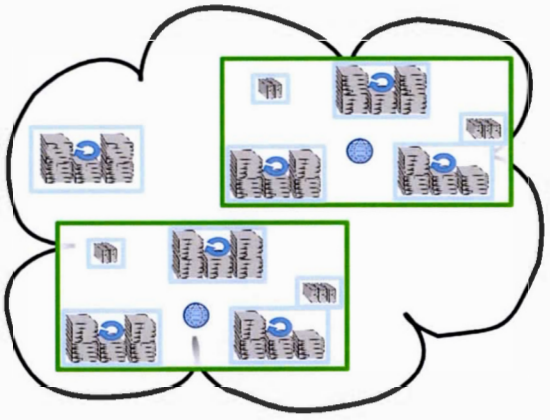
Les parallélisations matérielles, quant à elles, peuvent être définies sommairement selon le niveau de prise en charge de la parallélisation par le matériel. Par exemple, sur des machines multicoeurs présentes au sein d'un *cluster*, on peut définir une parallélisation au niveau des muticoeurs et un second niveau de parallélisation au niveau du *cluster* en répartissant les calculs sur les nombreuses machines. Il est donc possible d'effectuer du parallélisme à différentes échelles, et ce, de manière non exclusive. La figure 2.7 présente les inclusions possibles partant du *cluster* jusqu'à l'infonuagique.



(a) *Clusters*



(b) Grilles de calcul



(c) Infonuagique

Figure 2.7: Schématisation du *cluster* à l'infonuagique

Les ***clusters*** sont des regroupements dans un même lieu de machines identiques (baies de serveurs, postes de travail, etc.). Ils fonctionnent comme s'ils n'étaient qu'une seule machine virtuelle. La **grille de calcul** est une approche plus globale et décentralisée. Une grille peut comprendre plusieurs clusters de types différents. Elle considère et répartit les calculs selon les besoins (Armbrust *et al.*, 2010).

Quant à lui, l'**infonuagique** se construit à partir de grilles de calcul. Il est un modèle pratique, à la demande, qui permet d'établir un accès, par le réseau, à un réservoir partagé de ressources informatiques configurables (réseau, serveurs, stockage, applications et services) qui peuvent être rapidement approvisionnées et libérées en minimisant les efforts de gestion ou les interactions avec le fournisseur de service (Mell et Grance, 2011). De plus, l'infonuagique propose une bonne alternative pour les projets qui requièrent de forts besoins périodiquement, un besoin rapide de prototypage, ou encore des temps d'exécution rapides loin des limites imposées dans un environnement local (Fusaro *et al.*, 2011). L'infonuagique propose un ensemble de services à différents niveaux. Ces services peuvent être regroupés en trois catégories : Logiciel en tant que service (*SaaS*), Plateforme en

tant que service (*PaaS*) et Infrastructure en tant que service (*IaaS*) comme le présente la figure 2.8 issue de l'article de Zhang *et al.* (2010).

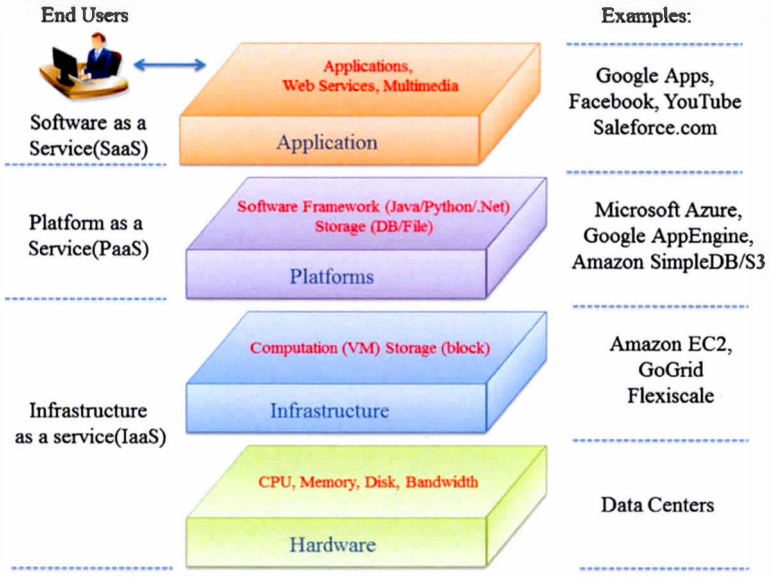


Figure 2.8: Services de l'infonuagique

2.3.2 Calcul distribué, SHD et flux de travaux

Au sein du SHD et des flux de travaux, le choix du calcul distribué a souvent pour objectif de répondre au besoin de mise à l'échelle, de gros volumes de données, d'utilisation de ressources non disponible localement, ou même de reproduire des expérimentations à faible coût. D'une manière générale, on trouve deux approches dans la littérature. La première est le passage par des plateformes comme Galaxy, que l'on essaye d'élargir jusqu'à l'infonuagique, et ce, sans régler complètement les soucis d'installation des différents programmes ou leur compatibilité avec l'infonuagique (Liu *et al.*, 2014). La seconde est la mise à l'échelle de machine virtuelle complète (Afgan *et al.*, 2010), mais cette approche reste difficilement exportable entre les environnements.

2.3.3 Calcul distribué et EV

Il existe une littérature récente autour de l'utilisation des conteneurs dans des calculs distribués ou sur l'infonuagique. Celle-ci porte souvent sur l'importance de la facilité de déploiement, de déplacement ou de mise à l'échelle d'EV (Gentzsch, 2014; Beserra *et al.*, 2015). D'après Hale *et al.* (2016), l'utilisation de conteneurs avec des programmes scientifiques spécifiques sur des systèmes de calculs hautes performances n'entraîne pas de pertes de performances. Les images des conteneurs proposent un environnement complet, stable, régulier et distribuable quelle que soit la plateforme physique (de l'ordinateur portable, à l'infonuagique en passant par les supercalculateurs).

2.3.4 Calcul distribué et Calcul Québec

Calcul Québec, partenaire régional de Calcul Canada, met à disposition des chercheurs canadiens des infrastructures de calcul distribué hébergées dans différentes universités à travers le Québec. Ces infrastructures possèdent des propriétés différentes. L'objectif de Calcul Québec est de proposer et d'utiliser le serveur qui correspond le mieux aux besoins du chercheur.

Le tableau 2.2 reprend les spécificités des différents *clusters* mis à disposition des chercheurs. Nous avons ajouté une annotation particulière pour les clusters possédant des affinités avec la bio-informatique ainsi que des programmes préinstallés et utilisés dans le domaine des SHD. D'après ce tableau, le cluster "Mammouth Parallèle II" semble être le plus prometteur par l'identification rapide des modules à charger lors de l'exécution des programmes.

Tableau 2.2: Spécificités des *clusters* de Calcul Québec

Cluster	Spécificité	Bio-informatique	SHD
Briarée	Biologie, Physique, Chimie	Oui	Oui
Colosse	-	-	-
Cottos	-	-	-
Guillimin	Physique, Chimie, Mathématiques	-	-
Hadès	Réservé pour l'utilisation des cartes graphiques	-	-
Helios	(Présence importante de cartes graphiques)	-	-
Mammouth Parallèle II	Physique, Chimie Bio-informatique, Mathématiques	Oui	Oui (64 [a])
Mammouth Série II	Physique, Chimie Mathématiques	-	Oui (1 [a])

[a] identifié(s) directement

2.4 Conclusion

Dans ce chapitre, nous avons présenté l'environnement virtuel comme une solution pour la portabilité des programmes SHD sur l'ensemble des systèmes d'exploitation. Cet outil sera donc inséré dans Armadillo comme support aux programmes Linux. Dans un second temps, nous avons présenté le calcul distribué comme solution aux besoins de ressources des programmes SHD. Un flux de travail local exportable vers un système de calcul distribué sera nommé **flux nuagique** et présenté dans le chapitre suivant.

CHAPITRE III

ARMADILLO 2.0

3.1 Les programmes SHD dans Armadillo

Comme nous l'avons montré dans le chapitre 1, Armadillo 1.1 est un logiciel de gestion de flux de travaux le plus proche de notre objectif. Il possède déjà la structure nécessaire pour la mise en place de flux de travaux en bio-informatique. Dans ce chapitre et dans la première section, nous verrons les modifications liées à l'intégration des nouveaux programmes de SHD pour répondre à la problématique sur la gestion des fichiers issus des programmes de SHD ainsi que sur l'automatisation de l'ajout d'une nouvelle capsule dans Armadillo. Dans la seconde section, nous présenterons plus en avant Docker comme logiciel de gestion de conteneurs et comme solution de déploiement et de portabilité d'applications SHD au sein d'Armadillo.

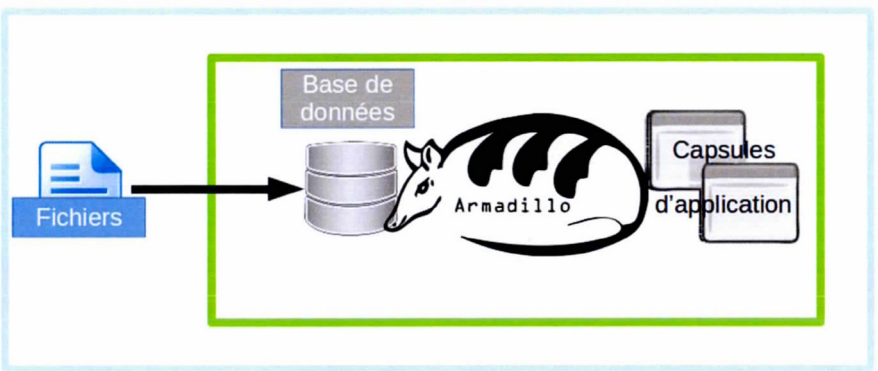
3.1.1 Gestion des programmes SHD dans Armadillo

3.1.1.1 La gestion de fichiers

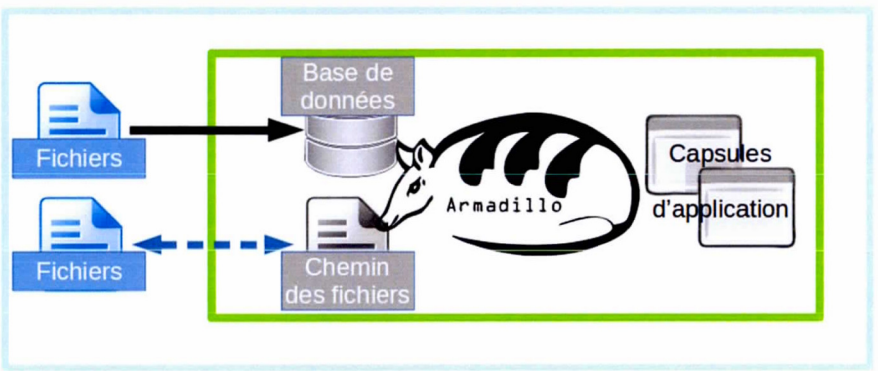
Jusqu'à présent, Armadillo 1.1 intègre les données fournies et produites dans une base de données. Ces données représentent en général de faibles volumes contrairement aux données provenant de SHD comme nous l'avons vu précédemment dans le chapitre 1 sur la biologie et la bio-informatique. L'utilisation de lien vers

les fichiers entre les différentes boites de dialogue des programmes a permis de résoudre la gestion des fichiers comme entité dans le flux de travail sans modifier la gestion des données précédemment utilisée.

La gestion des fichiers s'effectue en utilisant un chemin relatif ou absolu vers les fichiers puis en intégrant cette information dans les capsules qui l'utilisent. Une gestion de l'accès aux fichiers est utilisée avant chaque appel de ces derniers. La figure 3.1 présente les deux versions. La figure 3.1a est issue d'Armadillo 1.1. La figure 3.1b présente l'ajout de la gestion des fichiers et le lien toujours présent entre Armadillo et le fichier dans son intégralité.



(a) Armadillo1.1



(b) Le nouvel Armadillo maintient le lien vers le fichier source

Figure 3.1: Gestion des fichiers dans Armadillo

3.1.1.2 Les étapes de l'insertion d'un programme

Dans Armadillo 1.1, l'ajout d'un programme peut être fait directement dans le flux de données par l'utilisateur via une interface générique ainsi que sous la forme préinstallée à l'aide d'une capsule (interface entre l'utilisateur et d'un programme associé). Nous nous intéresserons ici à la seconde approche qui permet une approche plus complète, stable et transposable d'un utilisateur à l'autre. Concrètement, l'installation préalable de la capsule d'un programme se déroule en plusieurs étapes. La première consiste en l'analyse du programme en question, ce qui permet de comprendre son fonctionnement, ses entrées et sorties, les options disponibles et sa structure. La seconde comprend l'encapsulation et l'insertion de la capsule dans Armadillo.

L'analyse du programme et sa représentation représentent une étape difficilement automatisable, car il n'existe pas d'obligations (seulement des bonnes pratiques et des recommandations) quant à la mise en place d'une aide pour comprendre un programme. Il existe plusieurs sources d'information pour obtenir les arguments. La plus directe est obtenue lors de l'appel de l'aide du programme souvent obtenu avec l'argument *-h* ou *–help*. Une autre source peut être obtenue dans le fichier "lisez-moi" (*readme*) généralement fourni avec le programme. Il est également possible d'obtenir de l'information sur le programme par l'intermédiaire d'Internet sur le site de l'application, l'aide en ligne ou même encore les forums de discussions. Enfin, il est important de comprendre les entrées et sorties du programme.

Une fois cette analyse effectuée, il est possible de créer un fichier YAML (ex. avec EMBOSS chips dans l'annexe A.1) le représentant. Le langage YAML est un format de représentation de données dont l'objet est de représenter des informations élaborées en gardant une grande lisibilité. La hiérarchie de la structure de données

est assurée par une indentation homogène. Dans notre cas, ce fichier YAML se décompose en plusieurs parties :

La partie **"Programme"** a pour objectif de présenter les grandes lignes du programme. Elle se décompose en différentes sections qui sont principalement utilisées dans la création du fichier des propriétés de la capsule. Bien que la liste des modules soit la même commande pour l'ensemble des *clusters* lorsqu'il est disponible, le nom du programme et sa version sont deux points importants qui doivent être renseignés et immuables dans les propriétés de la capsule des programmes d'Armadillo.

La partie **"Docker"** permet de préparer l'intégration de Docker dans Armadillo. Elle contient entre autres les informations de base pour appeler le conteneur et exécuter le programme. Elle permet également de préparer le transfert de fichier entre l'hôte et le conteneur.

La partie **"*Inputs* et *Outputs*"** est une partie importante, car elle présente les entrées et sorties du programme. C'est elle qui va permettre de relier les capsules entre elles et de transférer les informations entre les capsules reliées. Les entrées et sorties dépendent des types de fichiers présents dans le répertoire [chemin vers armadillo]/src/biologic/ dans lesquels sont définis la nature du lien. Ainsi, une capsule dont la sortie est compatible avec l'entrée d'une autre pourra être reliée lors de la création du flux.

La partie **"commandes et menu"** rassemble les options disponibles pour l'application. Ces options seront visibles lors de l'édition de la capsule. L'idée générale est d'offrir les options selon un choix par défaut, suivi d'un mode avancé, et de rendre toutes les options disponibles.
La structure du fichier est présentée dans la figure 3.2. On retrouve le menu découpé en sous-menus dont le premier est le menu par défaut, puis viennent les

autres menus (ici, "Options Avancées" et "Autres Options"). Le menu "Options Avancées", par exemple, propose des options numérotées de 1 à n qui se composent d'un argument et, si besoin, d'une valeur. Chaque Menu peut également comporter des sous-menus comme c'est le cas avec "Autres options" et ses différentes sous-parties "Autres options Part" 1 à M.

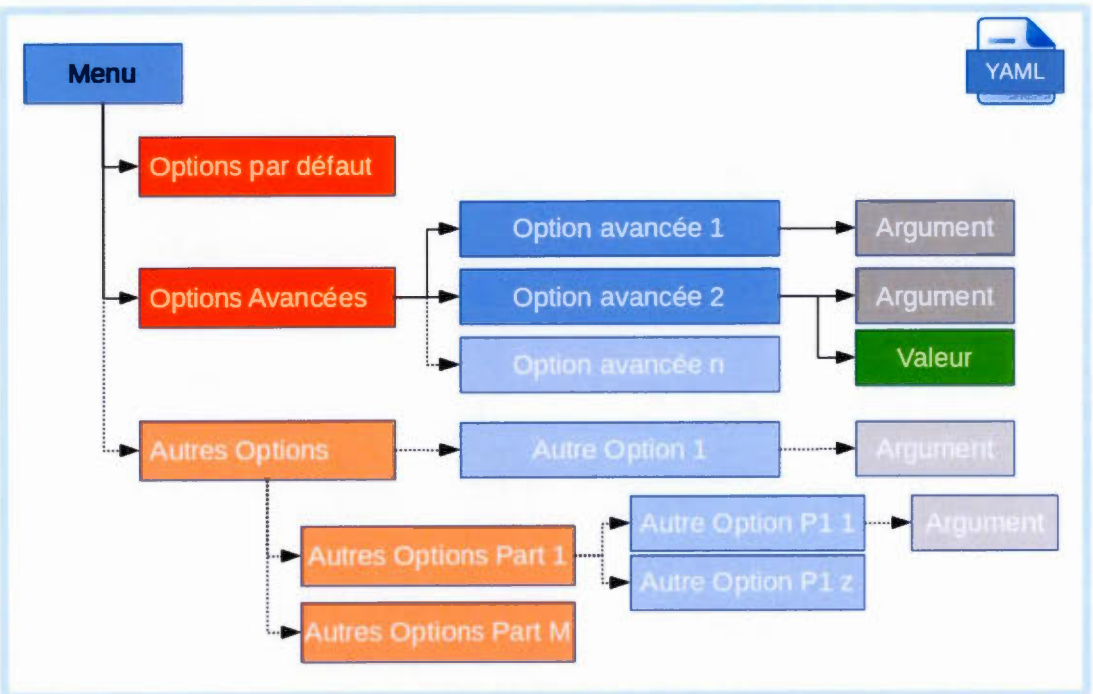


Figure 3.2: Découpage du menu d'un programme

Afin d'expliquer comment un argument est représenté dans Armadillo tout au long de son insertion, nous utiliserons l'exemple du programme *tail* qui sous Unix permet d'obtenir les dernières lignes d'un fichier et deux de ses options *--quiet* (qui ne nécessite pas de valeur) et *-n* (qui nécessite une valeur sous la forme d'un entier positif supérieur à 0 et dont la valeur est 10 par défaut). La figure 3.3 schématise les passages entre les types de représentation à l'aide d'un jeu de couleur dont le gris sera l'argument, l'orange le programme et le menu, et le vert pour les valeurs. Pour un argument simple, c'est-à-dire sans valeur associée, sa représentation dans

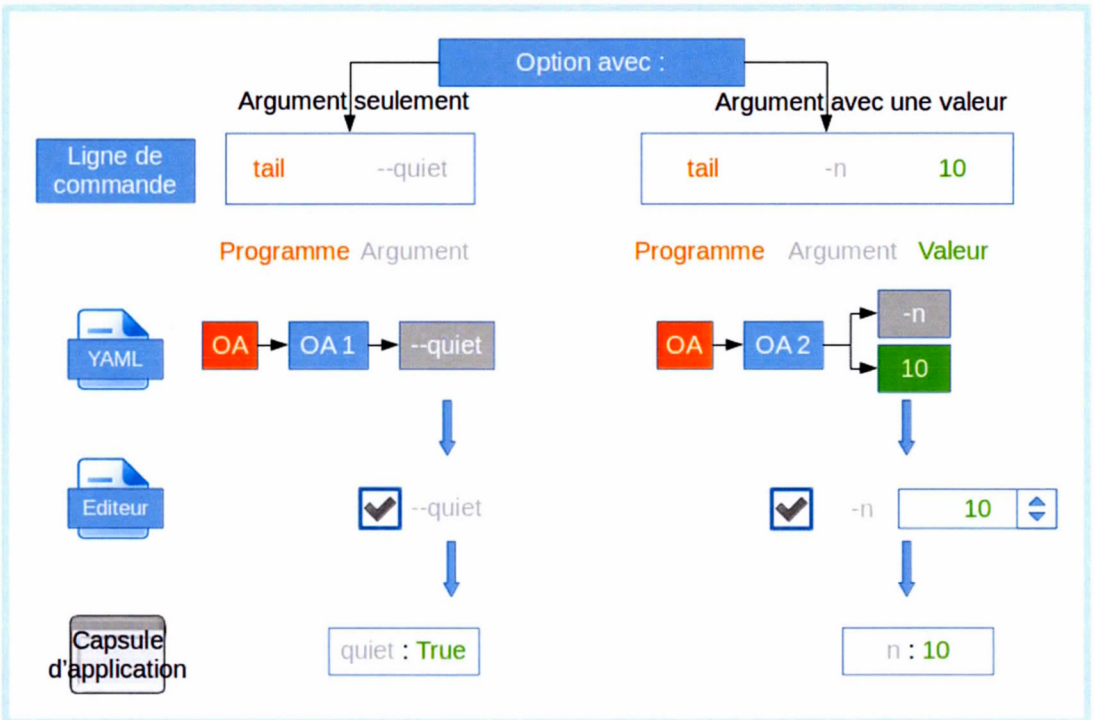
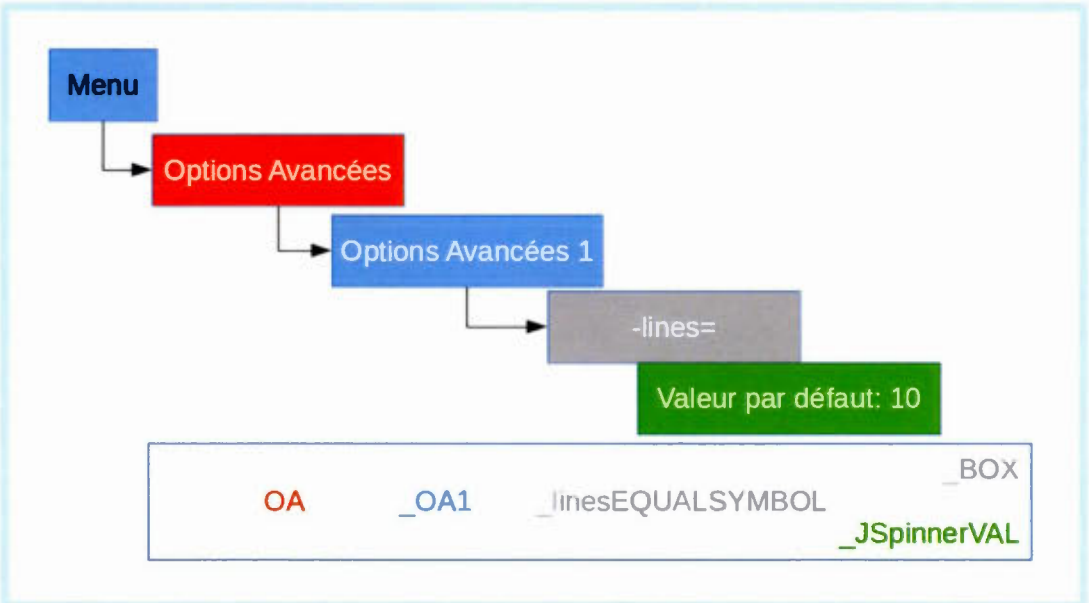


Figure 3.3: Représentation d'un argument du programme jusqu'à l'encapsulation

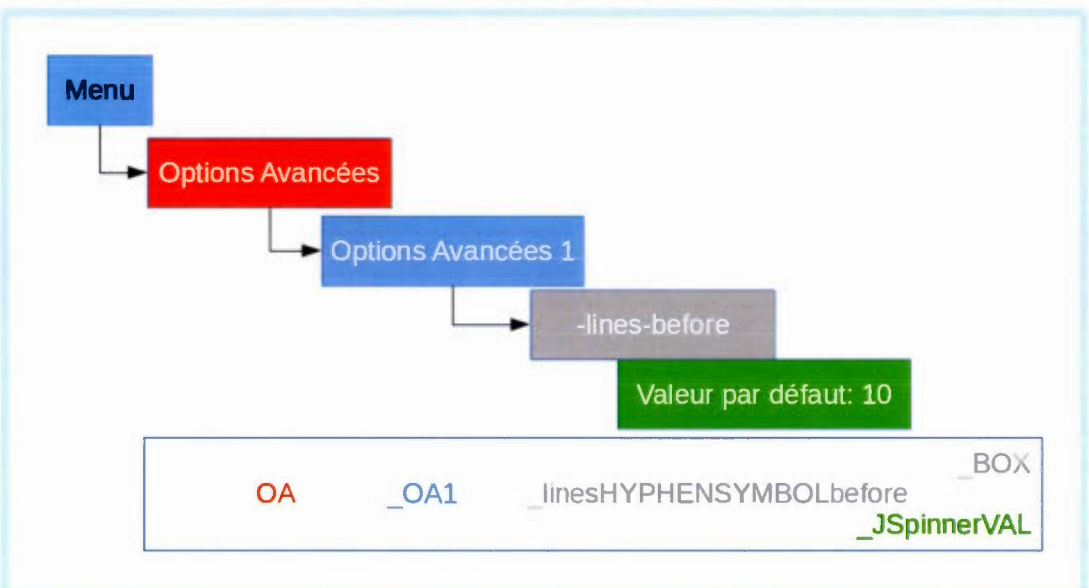
le fichier YAML découle du menu d'options avancées ("*AO*") puis de l'option avancée 1 ("*AO1*") et de l'argument *–quiet*. Sa transformation dans l'éditeur (la partie visuelle/utilisateur d'Armadillo) se fera sous la forme d'une boite à cocher. Si elle est sélectionnée, elle apparaîtra dans les propriétés de la capsule sous la forme d'une clé-valeur avec pour clé l'argument et pour valeur la donnée *True*.
Pour un argument associé à une valeur, sa représentation dans le fichier YAML est proche de celle d'une valeur simple sous une nouvelle option ("*AO2*") et comporte également une valeur associée. Sa représentation dans l'éditeur sera faite à l'aide d'une boite à cocher pour choisir l'argument et d'une liste modifiable de type numérique dans notre exemple pour la valeur associée. Lors de sa sélection, elle sera sauvegardée sous la forme d'une clé-valeur avec pour clé l'argument et pour valeur la donnée issue de la liste (10 dans notre exemple).

Pour aller plus loin, la récupération des informations sauvegardées dans la capsule est primordiale pour rapatrier les données initiales, charger les données entrées par l'utilisateur et fournir les options désirées rapidement.

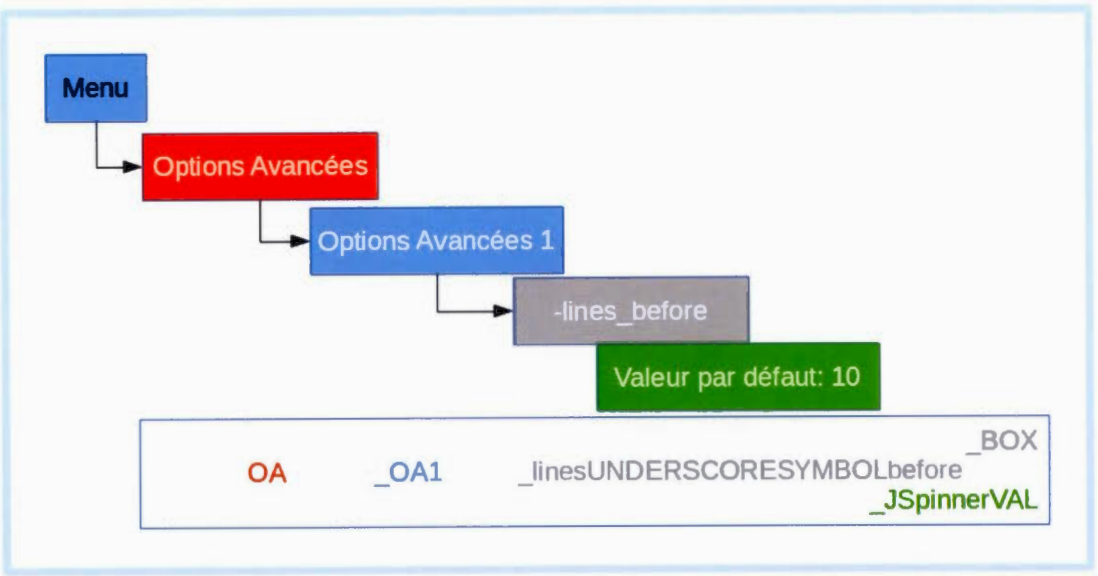
L'exemple présenté à la figure 3.3 propose la sauvegarde des données sous la forme d'une clé-valeur dont la clé est composée du nom de l'argument. Pour faciliter les échanges et homogénéiser les informations entre les différents fichiers sources, nous avons mis en place un moyen de représenter de manière unique ces données.
Ainsi, le nom d'un argument se voit ajouter les initiales des menus et sous-menus dans lesquelles il se trouve. Il se voit également ajouter un identifiant de fin relié à sa nature (BOX pour boite, JSPINNERVALUE pour une valeur présentée dans une liste modifiable de type numérique, etc.).
Si l'argument commence par un simple tiret alors, il comporte un simple tiret bas comme pour les exemples 3.4a, 3.4b et 3.4c. Par contre, s'il est précédé par un double tiret alors, il commencera par deux tirets bas comme pour 3.4d. Si l'argument nécessite le symbole égal, moins ou tiret bas, alors, il modifie le symbole avec le lettrage respectif suivant EQUALSYMBOL, HYPHENSYMBOL et UNDERSCORESYMBOL. Cette modification permettra de construire une variable Java acceptable qui d'un simple inversement reconstruira l'argument lors de la création de la ligne de commande.



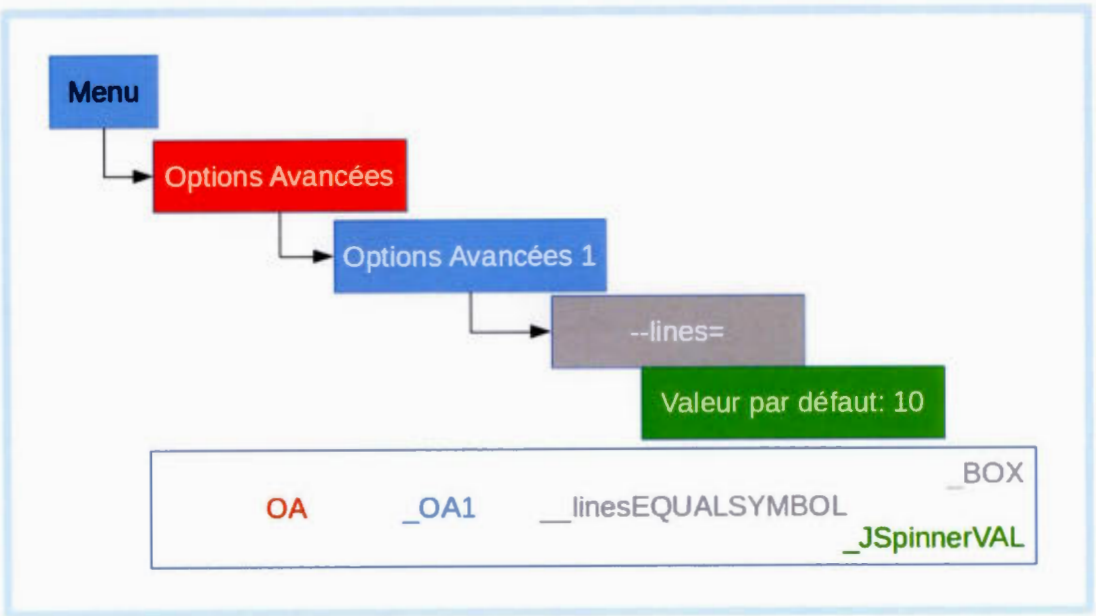
(a) Exemple 1 avec un égal



(b) Exemple 2 avec un tiret



(c) Exemple 3 avec tiret bas



(d) Exemple 4 avec égal et double tirets

Figure 3.4: Construction du nom de l'argument

La partie **"post-traitement"** permet d'enregistrer les étapes qui ont été nécessaires au bon fonctionnement de la capsule après sa création. Il est en effet

possible que les fichiers préparés soient incomplets ou ne prennent pas en compte une spécificité de l'application. La figure 3.5 présente par défaut l'organisation de la ligne de commande appelée dans la capsule par Armadillo. Celle-ci peut ne pas correspondre et devra être modifiée après la création du fichier "*programme*".

```
$ program [options] [inputs] [outputs]
```

Figure 3.5: Ligne de commande par défaut appelée dans la capsule par Armadillo

À partir de l'extraction des informations du programme et son insertion dans un fichier YAML, il suffit d'exécuter le script de création des fichiers. Celui-ci automatise la création des fichiers pour les capsules.

3.1.1.3 Les fichiers sources d'un programme

Les capsules des programmes d'Armadillo sont compilées à partir de trois types de fichiers tels que présentés dans la figure 3.6.

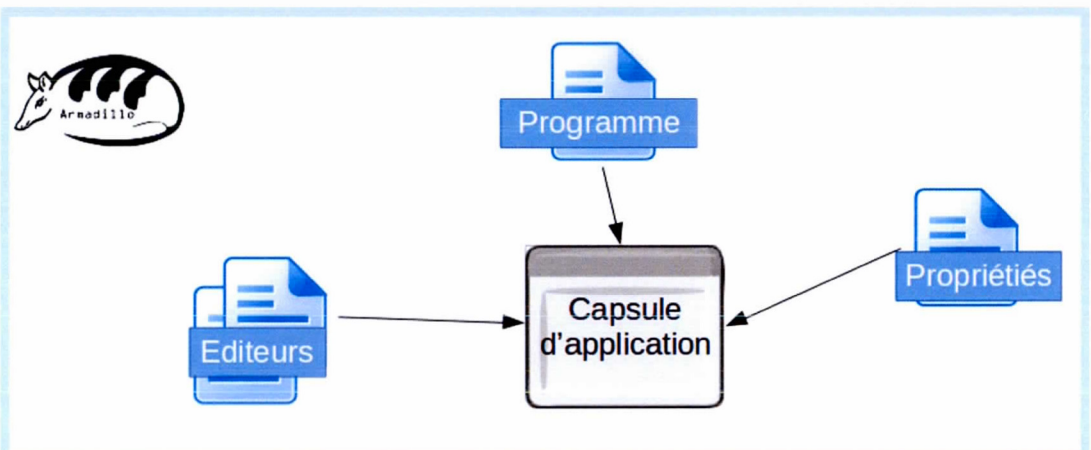


Figure 3.6: Fichiers liés à une capsule

Voici, ci-dessous, les définitions de ces trois fichiers permettant ainsi de mieux les

comprendre.

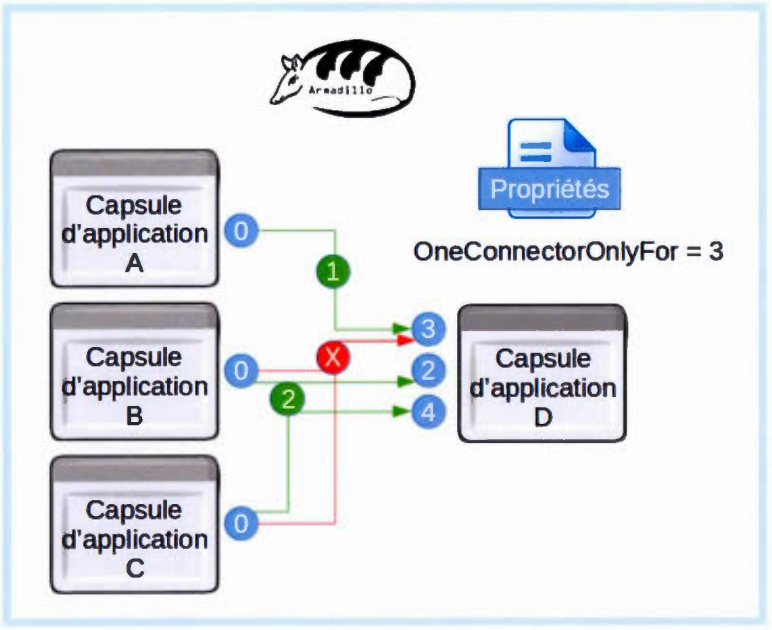
Le fichier "Programme" est un fichier Java qui se compose généralement de trois parties importantes. La première a pour objectif principal de valider les données entrantes. Elle permet également d'évaluer les données sauvegardées et d'ajouter des données par défaut si aucune n'est fournie par l'utilisateur. La seconde permet la préparation de la ligne de commande contenant les options qui vont être utilisées par le programme. La dernière s'occupe des sorties standards du programme, de la gestion des fichiers résultats et de la sauvegarde des données obtenues.

Le fichier "Propriétés" permet de fournir les informations de bases qui serviront au paramétrage par défaut des valeurs utilisées dans l'interface graphique et dans le fichier programme. Pour la bonne utilisation des programmes SHD et une homogénéisation des capsules d'Armadillo, il a été nécessaire d'augmenter le nombre d'entrées possibles, d'ajouter des informations sur les valeurs par défaut pour les options utilisateurs et la possibilité de limiter le nombre de connexions à l'une des entrées OU de n'autoriser qu'une entrée si elle est choisie. Le tableau 3.1 présente les entrées modifiées ou ajoutées dans le fichier, la définition associée ainsi que la construction valeurs par un exemple.

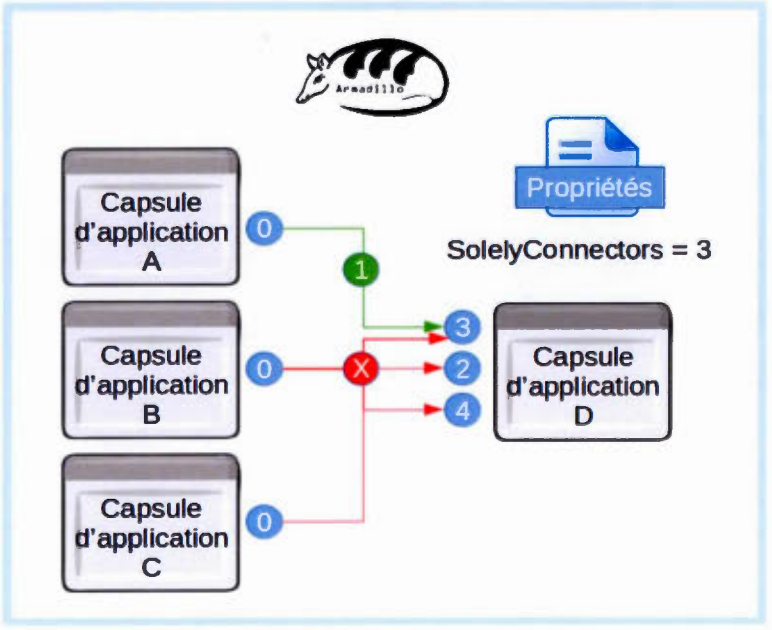
Tableau 3.1: Entrées modifiées ou ajoutées dans les options d'Armadillo

Type	Description	Valeurs ou Exemples
defaultPgrmValues	Valeur par défaut des programmes. Il existe une fonction dans le fichier *util.java* qui permet l'ajout de ces données à une capsule.	ex. nom : valeur ...
nbInput	Augmentation du nombre de ports d'entrées de 2 à 3	val. 2 (PortInputDOWN), 3 (PortInputUP), 4 (PortInputDOWN2)
SolelyConnectors	Connecteurs qui bloquent l'accès aux autres connecteurs s'ils sont sélectionnés.	ex. 2/3/4
OneConnectorOnlyFor	Connecteurs qui n'acceptent qu'une seule connexion.	ex. 2/3/4

La différence entre *SolelyConnector* et *OneConnectorOnlyFor* est schématisée dans la figure 3.7. Dans celle-ci, la figure 3.7a, le fichier "*Propriétés*" n'autorise qu'une seule entrée pour le connecteur 3. Lorsque celle-ci est utilisée par une connexion, elle n'est plus disponible pour d'autres connexions. La figure 3.7b présente un fichier "*Propriétés*" dans lequel seule l'option 3 est disponible si elle est choisie. Les autres options sont alors bloquées.



(a) Une seule entrée pour le connecteur 3



(b) Si choisi, un seul connecteur reste disponible

Figure 3.7: *SolelyConnector* et *OneConnectorOnlyFor*

La table 3.2 présente les modifications apportées au fichier "*Propriétés*" pour

l'accès au calcul distribué. Les options proviennent des informations nécessaires à une connexion sécurisée sur les clusters de Calcul Québec.

Tableau 3.2: Description des propriétés d'un *cluster* de Calcul Québec dans Armadillo

Type	Description	Valeurs ou Exemple
Name	Nom du Cluster pour l'accès ssh	ex. @briaree.calculquebec.ca etc..
userName	Nom de l'utilisateur	Vrai ou Faux
groupName	Nom du groupe de l'utilisateur (selon le *cluster*)	Vrai ou Faux

Dans la table 3.3, les différents éléments présentés correspondent aux informations nécessaires à la création d'un fichier Bash exécutable sur le *cluster* de Calcul Québec.

Tableau 3.3: Description des propriétés du fichier Bash exécuté sur le *cluster* de Calcul Québec

Type	Description	Valeurs ou Exemple
-q Email	Courriel de l'utilisateur possible ou non	Vrai ou Faux
-A rapID	Servant à spécifier le projet (RAP Id)	Vrai ou Faux
-q queue available	Noms des queues utilisables. La première sera celle utilisée par défaut	ex : test/courte/normale/longue/hpcourte/hp
-l walltime	Temps réservé pour effectuer la tâche par défaut (hh :mm :ss)	hh :mm :ss
-l nodes	Nombre de nœuds disponibles dans le cluster	ex : default/min/max/jump
-l ppn	Nombre de processeurs par nœud (" ppn »)	ex : default/min/max/jump
-r	Détermine si la tache peut être interrompue et reprise de zéro sans effets secondaires indésirables. Attention, sur certains cluster de Calcul Québec, Torque (*PBS*) considère qu'une tache peut redémarrer par défaut !	Vrai ou Faux
-N Add name	Ajoute un titre à l'exécution du Bash	Vrai ou Faux

Les fichiers "Éditeurs" sont aux nombres de deux. Le premier se présente sous la forme de fichier au format Java et le second sous un format XML. Par défaut, ils sont générés lors de la création de la capsule par le conteneur JFrame dans l'environnement de développement "intégré" utilisé pour la compilation d'Armadillo. Le fichier Java contient entre autres de nombreuses fonctions pour tester, capturer, sauvegarder et restituer les informations fournies par l'utilisateur. Il propose également un accès vers l'aide ou la gestion des données utilisées dans Docker. Le fichier XML comprend les informations de l'organisation de la structure de l'interface créée.

3.1.1.4 L'automatisation de l'insertion

Afin de faciliter leur mise en place, leur maintenance et le suivi des programmes encapsulés dans Armadillo, nous avons automatisé la création des fichiers d'encapsulation comme le représente la figure 3.8. Cette automatisation s'effectue à partir de la représentation du programme sous la forme d'un fichier YAML, puis d'un script qui le transforme en trois fichiers sources à l'origine de la capsule. L'automatisation du procédé d'encapsulation permet également de limiter les erreurs humaines et devrait permettre d'augmenter rapidement le nombre de capsules possibles dans Armadillo. Enfin, elle permet également de présenter des capsules avec une interface constante et non dispersive en homogénéisant les interfaces.

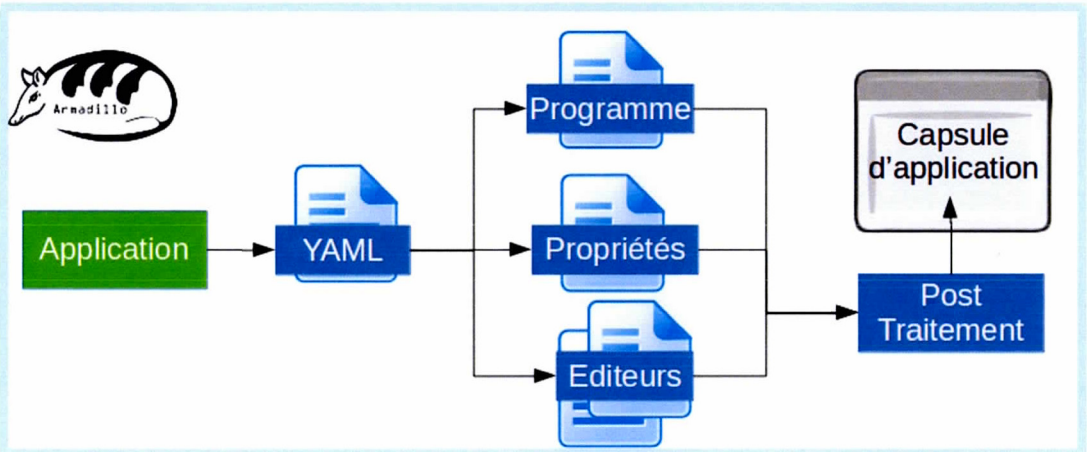


Figure 3.8: Schématisation des étapes pour l'encapsulation d'un programme

3.1.2 Les programmes ajoutés

Les programmes choisis ont été en priorité ceux utilisés dans la publication Agharbaoui *et al.* (2015). Puis, afin de répondre aux besoins du laboratoire de bioinformatique de l'UQAM, d'autres programmes de SHD ont été ajoutés. 19 nouvelles extensions possibles pour les entrées et sorties ont été ajoutées à Armadillo. Le tableau 3.4 liste les nouveaux programmes disponibles grâce au nom du programme (et/ou de la suite dans laquelle il se trouve), sa version, ainsi que les options disponibles (partielles ou toutes présentes). En résumé, les programmes SHD suivants ont directement été encapsulés dans Armadillo 2.0 :

— Bowtie 1.x et Bowtie 2.x (map, index, inspect)
— Suite de programmes miRcheck
— BWA (map et index)
— Cutadapt
— FastQC
— LoadFiles2.0

3.1.3 Conclusion

Dans ce chapitre, nous avons présenté les améliorations apportées au gestionnaire de flux de travaux Armadillo pour l'insertion de nouveaux programmes.

Dans le chapitre 1, nous avons présenté un logiciel **Objectif** qui avait de nombreux buts et à l'issue de cette section, les points suivants sont maintenant acquis par Armadillo :

— a une installation facile de nouveaux programmes,

— propose une interface utilisateur facile d'utilisation.

3.2 Docker : un complément pour Armadillo

Dans le chapitre 2, nous avons présenté Docker comme environnement virtuel et gestionnaire de conteneur. Celui-ci est un excellent moyen de rendre portable, multiplateforme et transparent l'utilisation de logiciels SHDs. Dans cette section, nous présenterons l'intégration de Docker au sein d'Armadillo et les différents défis que cela a représentés. Nous terminerons par la présentation des programmes Dockerisés.

3.2.1 Les intérêts de l'utilisation de Docker dans Armadillo

De manière synthétique et afin de seconder Armadillo dans l'utilisation de programmes SHD, Docker présente l'intérêt :

- de garder l'usage multi plate-forme d'Armadillo,
- de faciliter les échanges,
- de privilégier l'expérience utilisateur par la transparence d'accès aux programmes et sans les difficultés de leurs mises en place,
- de faciliter les mises à jour,
- de faciliter l'intégration de nouveaux programmes,
- de gérer l'accès aux ressources avec la notion de limitation d'accès,
- d'adapter les conteneurs aux besoins et aux disponibilités de programmes SHD,
- d'autoriser ou d'interdire l'accès en mémoire SWAP (options à implémenter),
- de gérer des dossiers partagés de la même manière, et ce, quel que soit le système d'exploitation,
- d'exporter les conteneurs vers l'infonuagique, les services PaaS et les services distribués.

Il est à noter que sur les systèmes d'exploitation Windows ou Mac, Docker passe

par le programme *DockerToolbox* qui utilise une virtualisation par hôte par l'intermédiaire du logiciel *Oracle VM Virtualbox*. Bien que cette machine virtuelle soit minimale, cette approche impose une couche supplémentaire (par l'hyperviseur de *Oracle VM Virtualbox*) comme nous l'avons présenté dans la section 2.2.5 à propos du logiciel Docker.

Récemment, on trouve de nombreuses intégrations de Docker dans des flux de travaux ou des applications de bio-informatique dont l'inspiration vient principalement de l'article de Boettiger (2015). Dans celui-ci, Boettiger explique l'importance des conteneurs (et plus particulièrement Docker) pour la reproductibilité et la transmission de code de programme de bio-informatique.

3.2.2 Intégration et relations avec Armadillo

Même si l'intégration de Docker au sein d'Armadillo est entièrement transparente pour l'utilisateur, ce dernier doit tout de même installer préalablement le programme Docker (ou le *DockerToolbox* selon le système d'exploitation). Dans cette section, nous présenterons l'intégration de Docker, la création d'un conteneur et sa mise en place. Enfin, cette section se terminera avec les modalités d'intégration des conteneurs.

3.2.2.1 L'intégration de Docker dans Armadillo

Il existe deux librairies JAVA disponibles qui permettent l'accès direct au RESTful API de Docker. Ces deux librairies sont des implémentations Java du client Docker. La première spotify/docker-client a été développée en fichier source ouvert par la compagnie Spotify. La seconde docker-java/docker-java est développée par la communauté, possède un wiki et un forum. La courbe d'apprentissage de ces deux librairies est relativement lente, car faiblement documentée pour des actions précises comme le partage de répertoires multiples par exemple. Dans le cadre

de ce mémoire, nous avons utilisé la librairie docker-java. Cette librairie a requis l'installation de 48 nouvelles librairies connexes pour un bon fonctionnement. Une fois les conflits de version validés, l'installation s'est avérée relativement simple mais elle n'a, au final, augmenté la taille de l'application Armadillo que de quelque 20 Mo.

3.2.2.2 Création, mise en place et utilisations des conteneurs

Pour faciliter la mise à disposition des programmes SHD, Docker vient en aide à Armadillo et permet l'accès à des programmes sans que l'utilisateur ait à les installer ou les paramétrer. En effet comme nous l'avons vu dans la présentation de ce logiciel, il a été possible de créer des fichiers (*Dockerfiles*) préparant l'installation de ces programmes et les rendants accessibles dans Armadillo. Les programmes peuvent être personnalisés pour répondre au besoin de l'environnement. Il est possible, comme proposer dans l'annexe A.2, de modifier la zone d'échange de fichiers ou encore de modifier les paramètres pour ajouter plus d'options. Les possibilités sont nombreuses et le langage est suffisamment souple pour laisser place à toutes les opérations possibles.

3.2.2.3 Modalité d'intégration des conteneurs

La mise en place des conteneurs est gérée intégralement par Docker grâce à la librairie Docker-Java. La figure 3.9 présente les communications entre la capsule d'Armadillo, Docker et le centre des applications de Docker (*Docker Hub*). Une fois que la capsule a transmis le conteneur qu'elle souhaite, Docker se charge de l'appel du conteneur localement s'il est présent (cas 1) ou depuis le *Docker Hub* s'il n'existe pas sur la machine de l'utilisateur (cas 2).

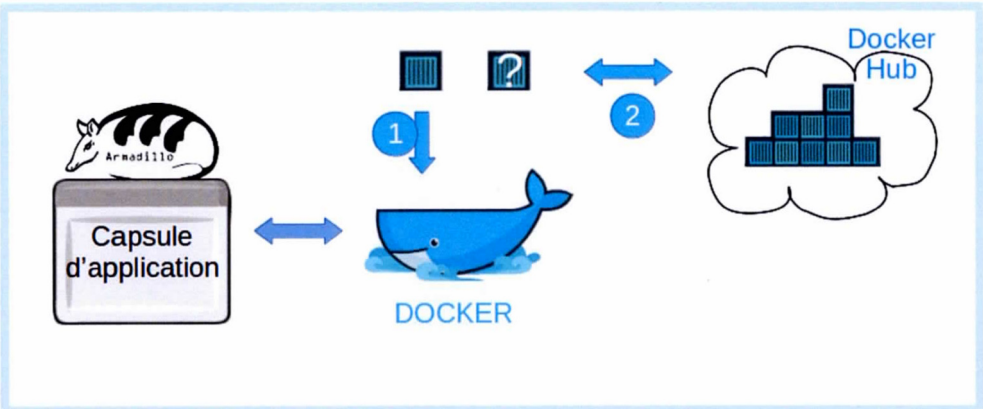


Figure 3.9: Armadillo appelant un conteneur Docker

Lors de son lancement, le conteneur est défini avec un nom spécifique unique afin d'obtenir une seule instance du conteneur par application. Des dossiers de partage sont également mis en place pour faciliter les échanges entre Armadillo et le conteneur (comme la présente la figure 3.10). Les fichiers créés lors de l'exécution du programme changent de propriétaire pour retrouver celui de l'utilisateur-hôte. Le conteneur ayant terminé son travail, il est stoppé et enfin supprimé. Il libère ainsi la place et le nom pour d'autres conteneurs. Il est possible de charger un grand nombre de conteneurs de la même application.

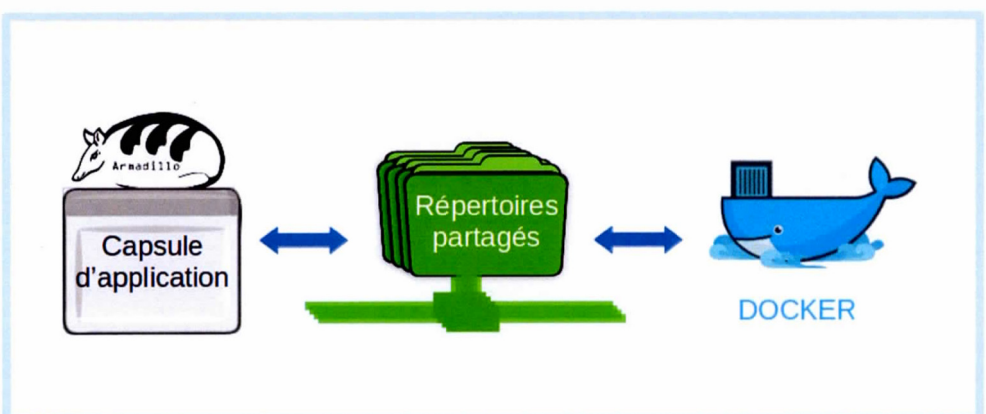


Figure 3.10: Répertoire partagé entre Armadillo et Docker

3.2.3 Les programmes intégrés dans Docker

Certains programmes comme Bowtie sont directement disponibles dans les dépôts de programmes sous Linux et en téléchargement compilés pour Windows et Mac. Cependant, c'est un cas de figure assez rare et souvent lié à des logiciels maintenant standards en SHD. D'autres ne sont disponibles que sur certaines plateformes (Ubuntu, Debian, Redhat). Les programmes compatibles uniquement sous Linux ont été les premiers encapsulés dans des conteneurs Docker. Les programmes associés sont les suivants (voir également dans le tableau 3.4) :

- Suite de programmes Emboss (aaindexextract, banana, chips, diffseq, einverted, est2genome, sizeseq)
- miRdup
- tapir
- Suite de programmes miRcheck
- Suite de programmes Vienna
- Suite de programmes Samtools (Partiellement)
- Suite de programmes BCFtools (Partiellement)
- embassy domainatrix
- embassy domalign
- embassy domsearch
- embassy phylip

3.2.4 Conclusion

Dans cette section, nous avons présenté l'intégration de Docker comme conteneur d'applications Linux en multisystèmes.

Dans le chapitre 1, nous avons présenté un logiciel **Objectif** qui avait de nombreux buts et à l'issue de cette section, les points suivants sont maintenant acquis par

Armadillo :

— supporte de nombreux systèmes d'exploitation sans restriction,

— propose nativement des programmes SHD et d'autres programmes de bio-informatique,

— propose de déploiement d'applications par l'intermédiaire de conteneurs.

Tableau 3.4: Liste des nouveaux Programmes SHD

Programme			OS				Options [op]
Nom	Version	Nom dans package	Win	OSX	Linux	Docker	
Bowtie 1.x	1.1.2	- map	*	*	*		T
		- index	*	*	*		T
		- inspect	*	*	*		T
Bowtie 2.x	2.2.5	- map	*	*	*		T
		- index	*	*	*		T
		- inspect	*	*	*		T
Emboss	6.6.0	- aaindexextract				*	T
		- banana				*	T
		- chips				*	T
		- diffseq				*	T
		- einverted				*	T
		- est2genome				*	T
		- sizeseq				*	T
Embassy	0.1.650	- domainatrix				*	T
		- domalign				*	T
		- domsearch				*	T
		- phylip				*	T
miRcheck	1.0	- einverted				*	T
		- evaluate miRNA candidates				*	T
		- extract einverted 20mers				*	T
		- fold inverted repeats				*	T
		- patscan				*	T
		- retrieve genomic regions				*	T
Samtools	1.3.1	- index				*	P
		- mpileup				*	P
		- sort				*	P
		- view				*	P
BCFtools	1.3.1	- call				*	P
BWA	1.3.1	- mem				*	P
Cutadapt	1.8.3				*		T
miRdup	1.4					*	T
FastQC	0.11.3				*		T
tapir	1.2					*	T
ViennaRNA	2.2.0a	- RNAFold				*	T

[op] : Les options disponibles pour l'utilisateur : T=Toutes,P=Partielles

3.3 Les flux de travaux

Dans cette section, nous présenterons les travaux de flux locaux et le flux nuagique comme preuve de concept. Pour ce dernier, la présentation au chapitre 2 sur le calcul distribué a montré l'importance du calcul distribué dans l'analyse de questions de biologie pour les SHD. Ceci est d'autant plus vrai lorsque les données sont importantes et déjà hébergées sur un nuage comportant les outils informatiques pour traiter ces données.

3.3.1 Les flux de travaux locaux

Lors de la création d'un flux de travail, les capsules sont reliées les unes aux autres grâce à un système de *Drag and Drop* comme décrit dans le paragraphe 3.1.1.2. Les tests, actuellement implémentés sur les fichiers entrants d'une capsule, portent uniquement sur l'extension et sur la nature des liens entre les types de sorties et entrées de chaque capsule (qui permet la connexion). Il serait intéressant d'ajouter un test d'intégrité des fichiers connus pour savoir si cela correspond aux besoins. Malheureusement, cette approche a aussi ses propres limites, car il est possible que certaines formes de fichiers soient trop libres dans leur format pour être efficacement traitées et avec un résultat certain.

L'insertion des programmes est en général suivie par des tests utilisant les données fournies par les concepteurs des programmes. Lorsque les capsules sont installées, elles sont testées dans l'environnement Armadillo et les résultats obtenus sont comparés avec ceux attendus.

La figure 3.11 présente un flux de travail exécuté pour les applications Bowtie 1 et 2. Il comprend l'analyse qualité de fichiers fastq, l'indexation de génome par Bowtie 1 et 2 à partir de fichiers fasta, puis l'alignement des fichiers fastq sur le génome indexé.

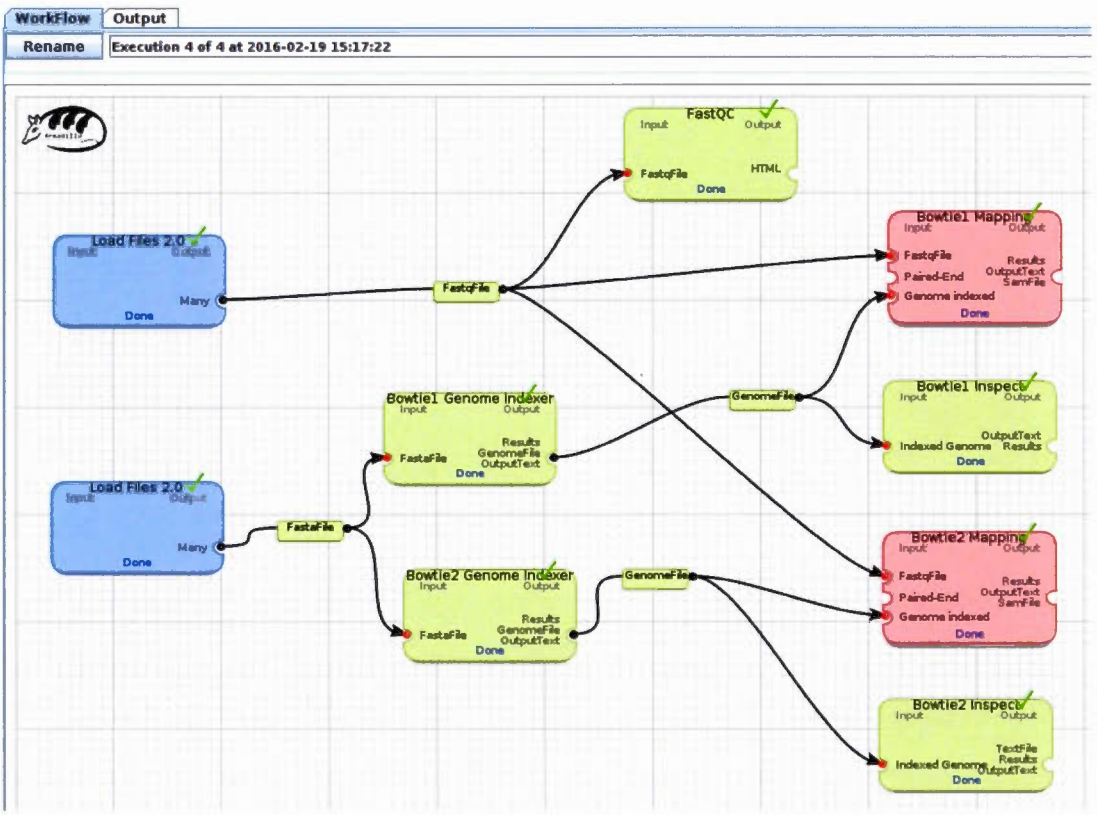


Figure 3.11: Exemple d'un flux contenant des programmes SHD

3.3.2 Le flux nuagique d'Armadillo

Un des objectifs pratiques de ce mémoire a été d'offrir à Armadillo une preuve de concept qui consiste à permettre à ce dernier d'effectuer des tâches sur l'infonuagique, sans toutefois remettre en cause le flux de travail, et ce, quelle que soit sa zone d'exécution (locale ou nuagique). Pour des raisons pratiques, ce travail a été effectué sous Linux.

3.3.2.1 Un travail local et sur un système de calcul distribué

Armadillo2.0 offre la possibilité d'exécuter des programmes sur des *clusters*, groupe de calcul ou tout autre type de services (infonuagique, etc.). Le choix est dispo-

nible par l'intermédiaire d'un menu déroulant dans la section en haut à droite d'Armadillo (cf. figure 3.12 (1)). Une fois sélectionnée, une nouvelle boite donne accès aux paramétrages du cluster (cf. figure 3.12 (2)). Les options de connexion proposées (cf. figure 3.12 (3)) sont actuellement uniquement orientées vers l'utilisation sur les *clusters* de Calcul Québec. Deux boutons se trouvent à la base de la boite. Le premier permet de paramétrer la première tabulation et le second intervient pour la seconde tabulation (cf. figure 3.12 (4)). La première tabulation est présentée dans la figure 3.13a. Il y est possible d'accéder aux *clusters* en définissant le nom d'utilisateur, le nom de groupe si nécessaire et l'identificateur de projet (RAP-ID).

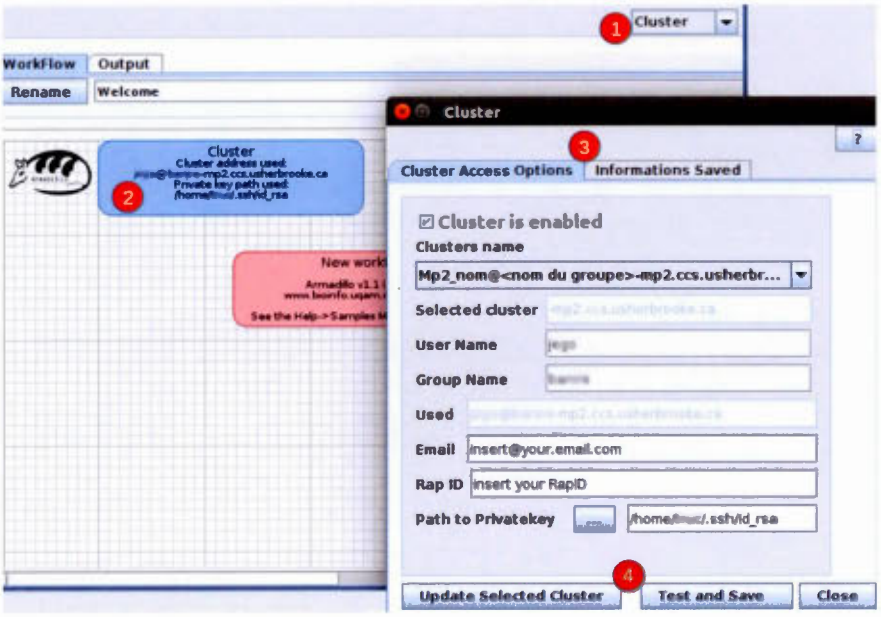
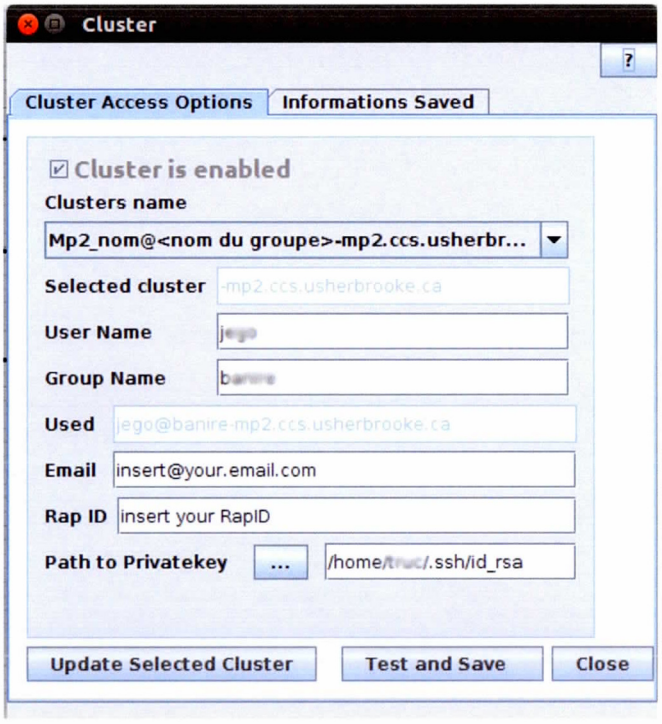


Figure 3.12: Vue utilisateur du flux nuagique dans Armadillo

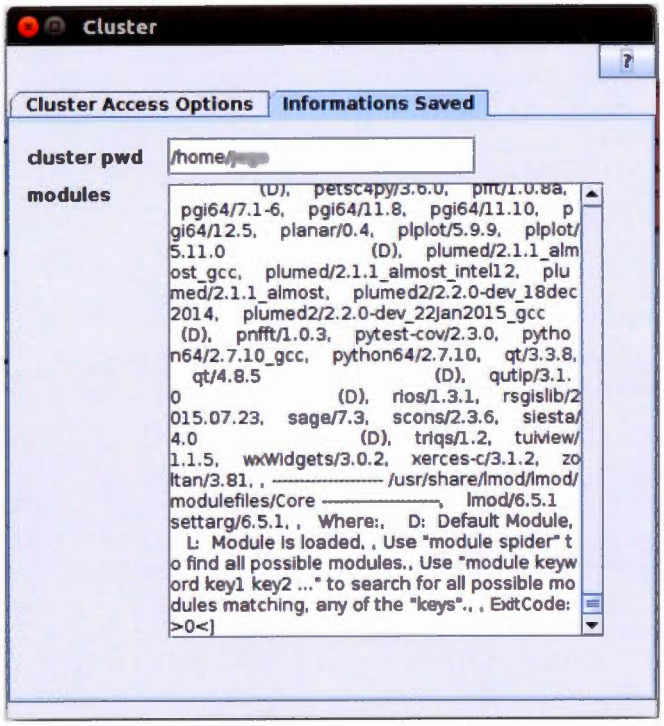
Les protocoles d'échange entre Armadillo et l'infonuagique utilisent des accès à l'aide de clés privées et publiques pour effectuer les échanges entre la machine hôte (qui utilise Armadillo) et le cluster distant. Ces protocoles permettent une communication sécurisée entre les machines, que ce soit pour l'exécution de lignes de

commande ou pour l'envoi de fichiers. La figure 3.14 présente les quatre principales étapes du protocole d'échanges entre Armadillo et l'infonuagique.

La seconde dans la figure 3.13b rassemble les informations du chemin absolu au sein du cluster et des modules présents disponibles dans celui-ci.



(a) Vue de la première tabulation



(b) Vue de la seconde tabulation

Figure 3.13: Les options pour le flux nuagique

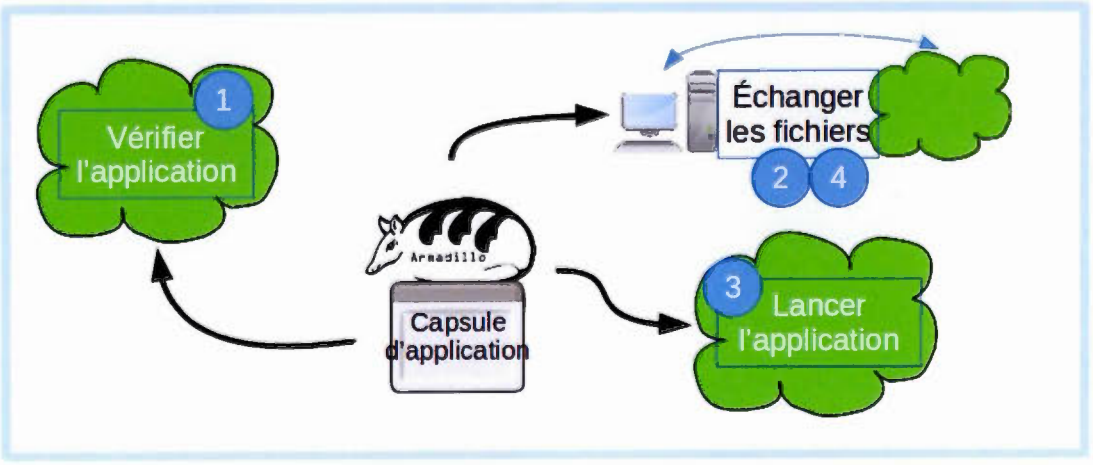


Figure 3.14: Étapes du flux nuagique dans Armadillo

(1) La première étape consiste à tester l'existence du programme que l'on souhaite exécuter sur l'infonuagique. Sur les *clusters* de Calcul Québec cela se traduit sous la forme de module à charger lors du lancement de la tâche comme présentée précédemment. Le test s'effectue grâce à la ligne de commande *module avail* (ou *module avail list*). Par exemple pour Mammouth Parallèle II et parmi l'ensemble des modules, 64 sont identifiés comme des logiciels de bio-informatiques grâce à la mention "bioinformatics/" précédent le nom du logiciel et sa version (ex. *"bioinformatics/bowtie/0.12.7"*). Il est donc possible de tester la présence du programme souhaité dans cette liste. Pour l'instant, les préalables des modules ne sont pas pris en compte pour le chargement des modules ce qui limite le flux nuagique à l'utilisation de programmes n'ayant pas de dépendances. Il est possible de tester la connexion et de pré-enregistrer les informations issues du cluster et utilisées pendant l'exécution d'un flux. Si ces informations ne sont pas disponibles directement, une nouvelle tentative sera effectuée lors du lancement du programme.

(2) Lorsque la première étape est validée, Armadillo prépare un fichier *BASH* pour l'exécution du programme et envoie l'ensemble des fichiers sur l'infonuagique (seconde étape).

(3) À l'étape trois, Armadillo lance l'exécution du fichier *BASH* puis teste régulièrement la fin de l'exécution de la tâche grâce à l'identifiant de tâche sur le cluster.

(4) Lorsque les fichiers résultats sont disponibles, Armadillo les récupère et les réintègre au flux de travail ce qui correspond à la quatrième étape.

3.3.2.2 Les *clusters* disponibles

Au cours de nos travaux, les calculs ont été effectués sur le supercalculateur "Mammouth Parallèle II" de "Université de Sherbrooke", sous la gouverne de Calcul

Québec. L'exploitation de ce supercalculateur est financée par la Fondation canadienne pour l'innovation (FCI), le ministère de l'Économie, de la Science et de l'Innovation du Québec (MESI), et le Fonds de recherche du Québec - Nature et technologies (FRQ-NT). Les autres *clusters* bientôt disponibles et paramétrés dans Armadillo sont :

— Briaree calculquebec.ca

— Colosse calculquebec.ca

— Cottos calculquebec.ca

— Guillimin hpc.mcgill.ca

— Ms2 ms.ccs.usherbrooke.ca

3.3.3 Exemple d'un flux de travail SHD avec les différentes options de traitement d'Armadillo 2.0

3.3.3.1 Présentation du flux de travail Snakemake

Le flux de travail présenté ici a pour objectif de trouver les variants ou *single nucleotide polymorphisms (SNPs)* d'un génome de référence. Il comprend les étapes pour mapper les *SNPs* de séquences sur un génome de référence et d'appeler les variants des *SNPs* alignés. Ce flux est originellement proposé comme exemple pour l'apprentissage du fichier de configuration du logiciel Snakemake (Köster et Rahmann, 2012) pour la bio-informatique. L'annexe B.1.1 présente sous la forme de code la première partie de l'exercice.

Ce flux de travail utilise trois logiciels principaux : BWA pour l'indexation, la suite Samtools pour indexer le génome, voir et trier variants, et enfin la suite BFCTools pour l'analyse de variants. La figure 3.15 schématise le flux de travail tel qu'il sera effectué sous Snakemake.

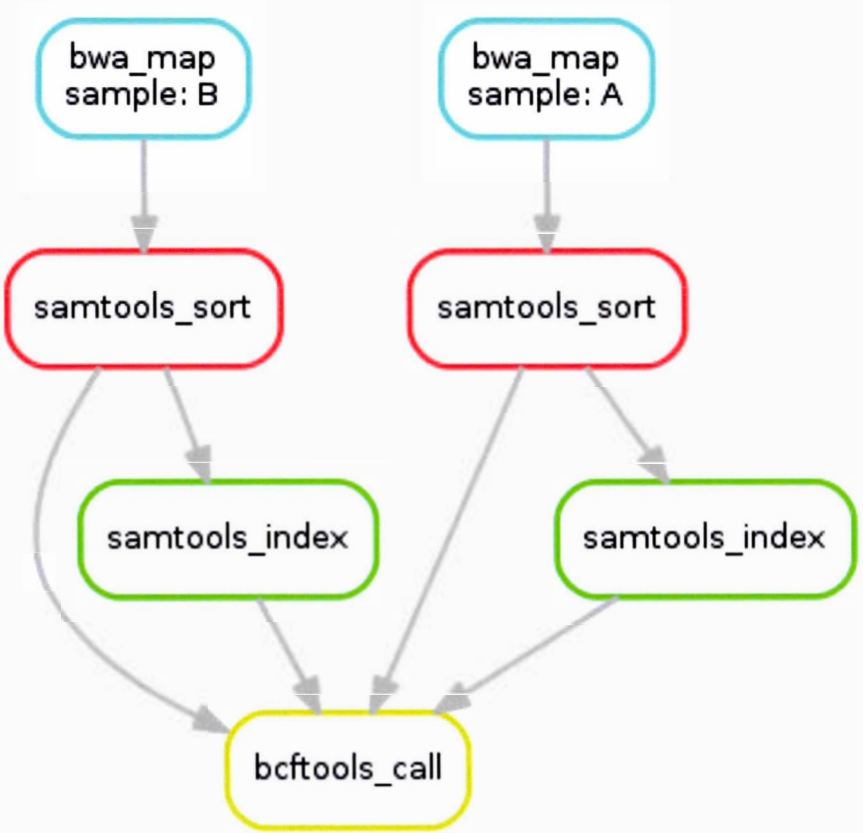


Figure 3.15: Exemple d'un flux selon Snakemake
(Source: https ://snakemake.readthedocs.io/)

La partie avancée de l'exercice, proposée par Snakemake et présentée dans l'annexe B.1.2, porte principalement sur l'accès aux fonctions avancées des programmes comme le support multicoeur et sur la gestion des logs.

3.3.3.2 Analyse du flux de travail et modifications du flux de travail Snakemake

On remarque ici que les données d'entrées proposées pour le test dans le flux de travail de Snakemake ont été préparées avec soin. En effet, le flux de travail réel derrière cet exercice est légèrement plus complexe. La figure 3.16 montre l'ensemble des étapes nécessaires à l'accomplissement du flux de travail tel qu'il doit

être pour répondre à la question biologique initiale. On peut noter que la première étape (#1) en orange n'est pas présentée dans le flux de travail de Snakemake 3.15. Elle est pourtant une étape préalable pour la préparation des données. L'indexation par "*samtools faidx*" peut être, également, effectuée juste avant la dernière étape pour trouver les variants et tout en restant préalable à l'utilisation de son résultat par le logiciel "samtools mpileup" (#4).

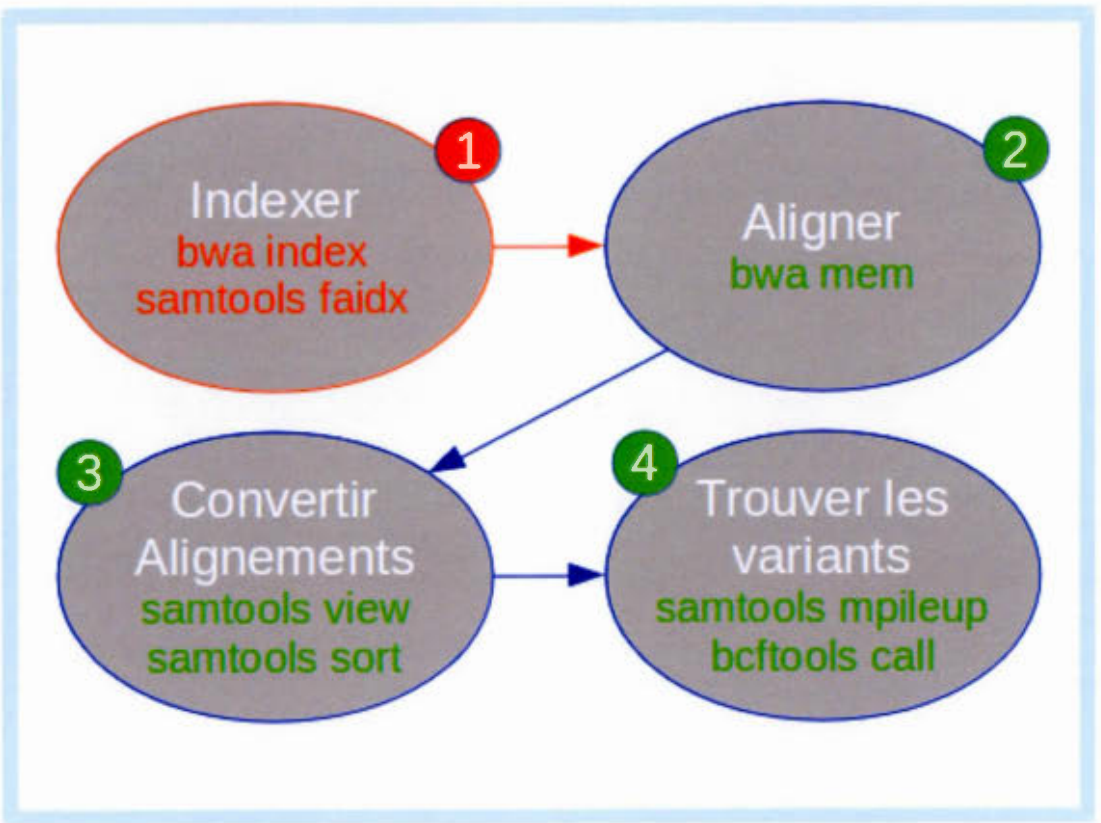


Figure 3.16: Les étapes réelles du flux de travail

Par ailleurs, il semble que la figure 3.15 ne soit pas tout à fait juste. En effet, dans la partie #4 de l'annexe B.1.1, les données entrantes sont au nombre de 3, le génome de référence (fa), les fichiers bam (*.bam*) et les fichiers bai (*.bai*). Or les fichiers bai sont d'une part absents de la ligne de commande (après le "*shell :*"

de la ligne 9 dans la figure 3.17) et ils ne semblent pas être utilisés par le logiciel "*samtools mpileup*". Il n'y a donc pas de lien direct entre l'indexation faite par le logiciel "samtools index" à l'étape #3 et l'étape #4. Cette remarque s'appuie également sur différentes autres sources externes ([1] ou [2]) qui proposent des flux de travaux d'apprentissage plus ou moins complexes autour des *SNPs*. Dans la présentation de *samtools index*, on retrouve également un lien entre ce programme et *samtools view* utilisé précédemment ([3]).

```
1  #4
2  rule bcftools_call:
3      input:
4          fa="data/genome.fa",
5          bam=expand("sorted_reads/{sample}.bam", sample=SAMPLES),
6          bai=expand("sorted_reads/{sample}.bam.bai", sample=SAMPLES)
7      output:
8          "calls/all.vcf"
9      shell:
10         "samtools mpileup -g -f {input.fa} {input.bam} | "
11         "bcftools call -mv - > {output}"
```

Figure 3.17: Le code de configuration de la partie 4 du flux de travail Snakemake

Les lignes 3 à 6 représentent les fichiers d'entrée, les lignes 7 et 8 le fichier de sortie, et les lignes 9 à 11 la ligne de commande *shell* à effectuer pour obtenir le résultat souhaité

1. `http://snp-pipeline.readthedocs.io/en/latest/dataflow.html`

2. `https://www.slideshare.net/danbolser/pipeline03x`

3. *This index is needed when region arguments are used to limit samtools view and similar commands to particular regions of interest.*

3.3.3.3 Le flux de travail Snakemake sous Armadillo

On peut donc reformuler le pipeline de Snakemake selon la figure 3.19, pour laquelle on retrouve les étapes réelles de l'exercice proposé par le logiciel Snakemake.

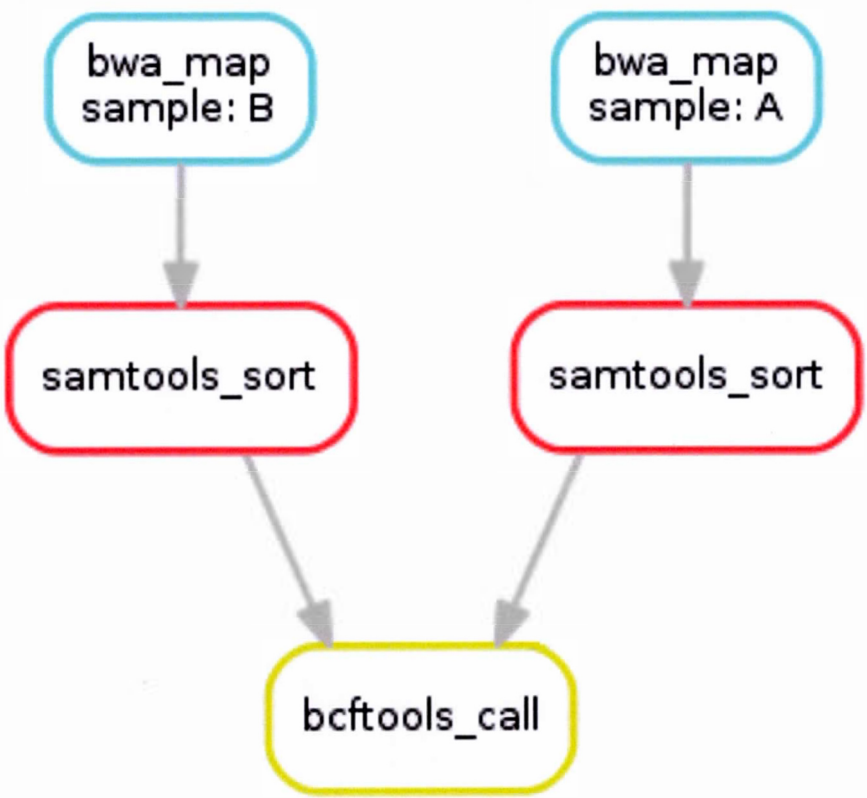
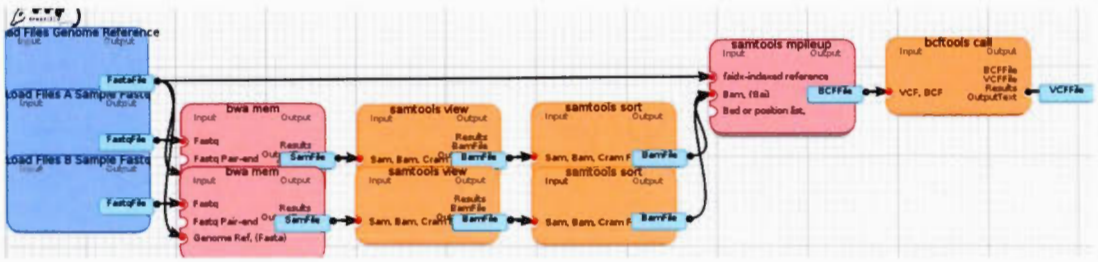


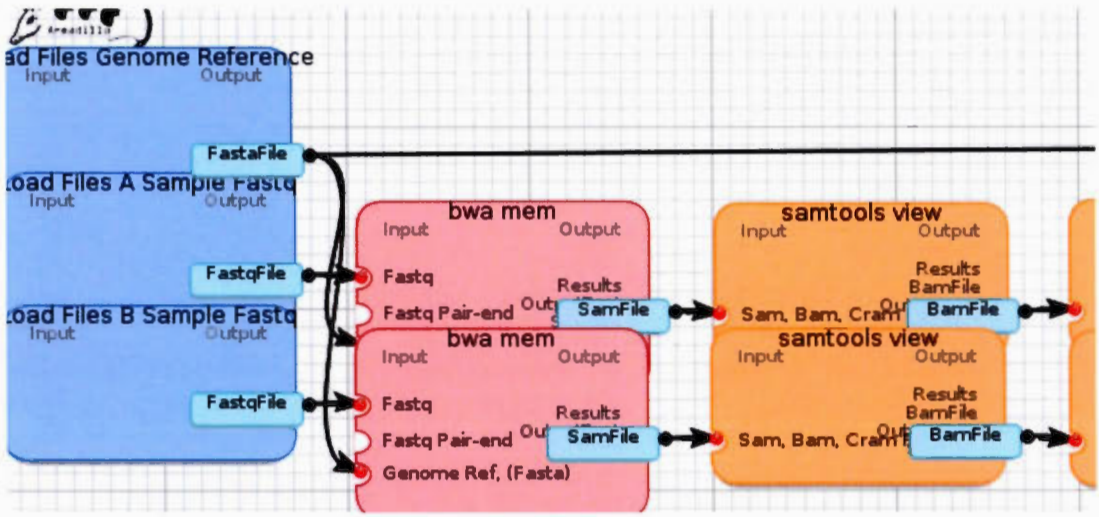
Figure 3.19: Correction du flux proposé dans le tutoriel Snakemake

Enfin, la partie rapport du logiciel Snakemake semble être propre au logiciel. Elle reste très succincte dans cet exemple, car elle se limite à offrir le lien vers les résultats obtenus.

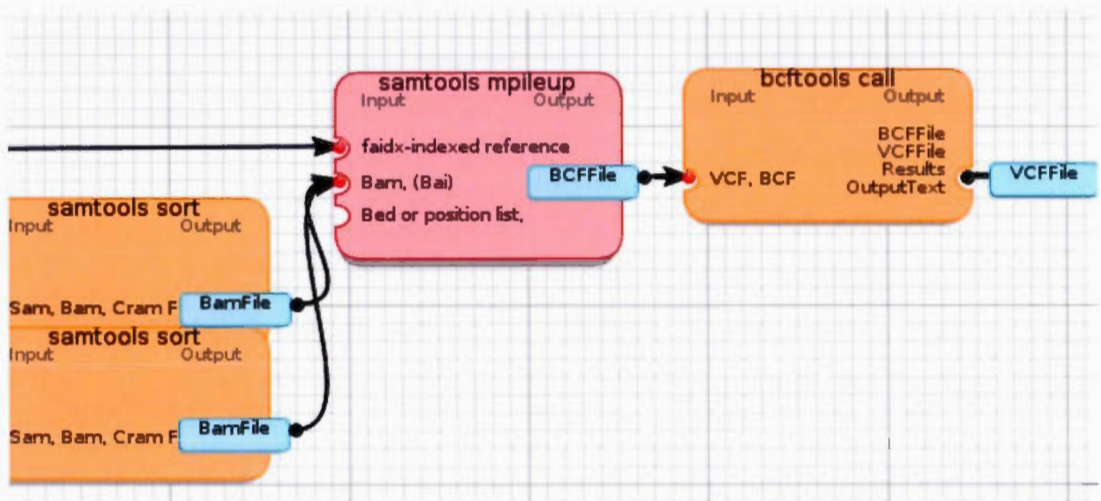
La figure 3.20 représente, quant à elle, le flux de travail sous Armadillo avec une vision globale en 3.20a, une vue de la partie gauche 3.20b et une vue de la partie droite 3.20c.



(a) Vue complète



(b) Vue partie gauche



(c) Vue partie droite

Figure 3.20: Représentation du flux Snakemake dans Armadillo

Sous Armadillo, l'accès aux fonctions avancées des programmes comme le support multicoeur et sur la gestion des logs sont disponibles et accessibles par défaut dans Armadillo sans modifications particulières du flux de travail. Il est simplement possible d'indiquer directement dans le logiciel utilisant le multicoeur l'option comme dans la figure 3.21.

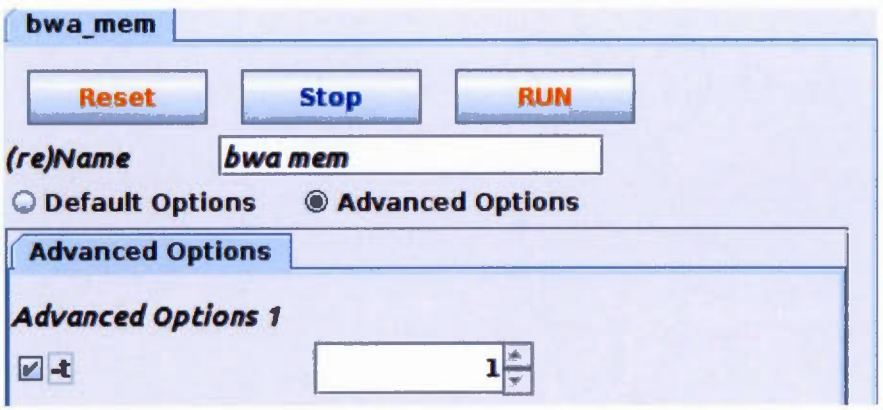


Figure 3.21: Sélection du multicoeur pour BWA

3.3.3.4 Exécution du flux de travail localement

Pour **comparer** et montrer la **reproductibilité** des flux de travaux sous Armadillo ou Snakemake, les logs des différentes approches sont présentés dans l'annexe B.2.1 pour le log du flux dans Armadillo et l'annexe B.2.2 pour la sortie standard du flux du programme Snakemake.

Les lignes de commandes explicitement décrites dans l'exemple permettent une comparaison entre celles originellement définies dans Snakemake et celles produites par Armadillo. Les arguments des programmes dans Armadillo proposent en option visible pour les utilisateurs les deux types de commandes, mais seule la commande la plus longue est utilisée comme argument lors de la création de la ligne de commande. En terme de construction, la mise en place du flux de travail sous Armadillo n'a pas demandé de connaissances particulières ou d'apprentissage

pour sa création, et ce, sans connaître les lignes de commandes sous Linux. Une bonne compréhension des options des programmes utilisés reste un point important pour le flux de travail et de ses résultats. De la même manière, Armadillo permet une relation entre ces capsules sans besoin de connaître les extensions ou les spécificités des fichiers de sortie.

La comparaison des résultats présentés dans l'annexe B.4 montre qu'ils sont identiques, car seules les lignes de commentaire sont différentes. Il est possible aussi de noter une différence de la librairie *htslib* dont la version est 1.3.2 sous Armadillo/-Docker et 1.3.1 sous SnakeMake. Ce flux de travail est donc bien reproductible avec le même jeu de données et les mêmes lignes de commande quelque soit le support pour créer le flux de travail. La reproductibilité est également dans le sens inverse. Il est possible d'effectuer un flux de travail comme présenté dans Armadillo avec le logiciel SnakeMake. On y retrouve les mêmes étapes et des résultats identiques (voir annexe B.1.3). Dans un premier temps, nous avons testé le flux avec les entrées (A et B). Nous avons ensuite travaillé avec l'ensemble des données proposées dans le jeu de données proposé (A,B et C). Le flux de travail sous Armadillo reste tout aussi fonctionnel.

Docker et Armadillo : cela fonctionne ! Dans cet exemple, il a été démontré qu'il était maintenant possible d'effectuer un flux de travail SHD dans Armadillo en utilisant Docker comme outil de conteneurs d'application localement. Cette approche est d'autant plus intéressante qu'elle permet à l'utilisateur de ne jamais avoir à installer le logiciel et pourtant en avoir l'usage. Cette solution de portabilité entre les systèmes d'exploitation est également porteuse d'avenir pour la reproductibilité des flux de travaux et leur échange au sein de la communauté scientifique.

3.3.3.5 Exécution du flux sur le cluster Mammouth Parallèle II

Le flux de travail précédent a été exécuté en local et sur le cluster Mammouth Parallèle II. Malheureusement, le flux n'est pas intégralement transférable sur le cluster, car les modules par défaut disponibles ne sont pas tous présents. Il nous a été possible de modifier la version de *SAMTools* utilisé ici pour une version légèrement supérieure (de 1.3.1 à 1.3.2). Par contre, le changement de version pour le logiciel *BCFTools*, qui est sous la version 1.3.1 dans Armadillo et sous la version 1.2 sur le cluster, n'a pas été possible. Il existe une trop grande différence entre ces versions. Le fichier log présentant le flux sur le cluster est disponible dans l'annexe B.3. Ce log présente les différentes étapes du flux et le travail qui est effectué sur le cluster.

À titre de comparaisons, le tableau 3.5 reprend l'ensemble des programmes utilisés dans ce flux de travail ainsi qu'un comparatif sur les temps d'exécution de chaque étape pour les deux approches Cluster et Docker. Il comprend les données moyennes pour 5 flux complets de 7 applications pour le Cluster et de 8 applications sur Docker.

Tableau 3.5: Table des temps par étapes d'un flux de travail local et nuagique

Étapes	Cluster		Docker	
	val (s)	%	val (s)	%
créer les répertoires ou lancer le conteneur	2	~0	1	~8
envoyer les fichiers	188	57	-	-
créer et envoyer le fichier de commande	4	1	~0	~0
exécuter la tâche	4	1	1	~8
attente de la fin du programme	126	38	-	-
télécharger les fichiers	7	2	-	-
arrêt et suppression du conteneur	-	-	10	~84

La création des répertoires pour la partie cluster et le lancement des conteneurs

Docker en local ont des temps relativement faibles. Il est important de noter que les images docker utilisées lors de ces essais étaient déjà présentes. Il n'y a donc pas eu de téléchargement des images depuis le *Docker Hub*. Les flux ont été effectués localement avec une connexion internet de type câble aux débits de 10 Mbit/s en téléchargement et de 1.5 Mbit/s en envoi. Bien que les séquences utilisées ici soient de taille très petite, leur transfert vers le Cluster représente 57% du temps total dans les conditions des essais de flux. Sous Docker, les répertoires où se trouvent les fichiers sont partagés. Il n'y a donc pas de transfert de données. Par contre, l'attente dans la file de traitement du cluster représente un temps important. Ce temps est en partie du à l'espacement entre nos tests pour vérifier l'état du travail. Celui-ci s'échelonne de la manière suivante : un test toutes les minutes pour les 6 premières minutes puis un test après 2 puis 4 puis 8 puis 16 puis 32 minutes. Soit un total de 68 minutes d'attente potentielle. Dans notre cas, le travail est généralement fini au second test. Le traitement de la tâche localement ou sur le cluster ne montre que peu de différence (4 secondes pour le Cluster et 1 seconde sous Docker). Ceci s'explique par la faible taille des jeux de données et le faible besoin en ressource système pour répondre à l'exécution des tâches. Enfin, l'arrêt et la suppression du conteneur restent une tâche homogène à 10 secondes.

Dans cet exemple, il a été possible d'effectuer un flux de travail en local et sur le cluster sans modifications du flux. Cette approche est d'autant plus intéressante qu'elle permet à l'utilisateur de tester et d'obtenir des résultats plus rapidement avec l'aide de la parallélisation et du calcul distribué. Cette solution alternative reste peu disponible dans d'autres applications. Seule la queue *qwork* a été utilisée. Il existe néanmoins d'autres queues de travail qui pourront mieux répondre au besoin de l'utilisateur.

3.3.4 Conclusion

Dans cette section, nous avons présenté la mise en place des flux de travaux au niveau local et distant sur un *cluster*. Nous avons démontré qu'il était maintenant possible d'effectuer un flux de travail en local et sur un système de calcul distribué avec des programmes SHD.

Dans le chapitre 1, nous avons présenté un logiciel **Objectif** qui avait de nombreux buts et à l'issue de cette section, le point suivant est maintenant acquis par Armadillo :

— peut être exécuté localement ou sur un serveur et sur un système de calcul distribué en ligne de commande.

L'ensemble des explications et fichiers pour effectuer les tests SNPs localement et sur le cluster est disponible sur le dépôt : `https://github.com/JeGoi/Armadillo_SNPs_WF_test`.

3.4 Discussions sur Armadillo 2.0

3.4.1 L'intégration de nouveaux logiciels : avancées et limites

L'**intégration de nouveaux logiciels** a d'abord été faite de manière manuelle pour ensuite se tourner vers une automatisation de l'interface comme nous l'avons montré dans la section 3.1.1.4. Il existe néanmoins plusieurs limites à l'utilisation de Docker. Certains programmes nécessitent une réécriture pour la gestion des options ou l'accès aux fichiers (chemin relatif ou absolu). De plus, il est parfois nécessaire de bien identifier les licences d'utilisation (comme pour le logiciel miRcheck) ce qui complexifie la création du *Dockerfile* et son déploiement (le temps de préparation de l'image finale peut être important). Néanmoins, ces dernières étapes restent transparentes pour l'utilisateur.

Un **accès aux images** reste nécessaire pour une première utilisation du conteneur (ou après sa suppression sur la machine hôte). Il peut se faire par l'intermédiaire du centre de services de Docker (*DockerHub*) ou tout autre moyen équivalent permettant le chargement de l'image sur la machine hôte. Pour les programmes installés dans ce travail, une connexion à l'Internet est nécessaire, car les dépôts sont présents sur le Docker Hub. L'accès à ce dépôt facilite grandement l'utilisation de Docker.

L'intégration de Docker, des programmes de SHD et la gestion de fichiers légèrement différente **n'ont pas modifiés le fonctionnement initial** d'Armadillo 1.1.

Bien que Docker se prévaut d'une légèreté dans ses conteneurs, **la taille des conteneurs** reste relative et demande parfois des temps de chargement relativement longs. Dans ce projet, un programme ou une suite de programmes pour les progiciels correspondent à une image et un conteneur. Cette approche par com-

partimentation permet de tester les applications sans interactions ou interférences potentielles entre elles. Si seules quelques applications sont régulièrement utilisées par l'utilisateur, elle présente aussi l'avantage d'avoir une taille globale inférieure à celui issu d'un conteneur contenant toutes les applications encapsulées dans Armadillo. L'intégration de l'ensemble des applications dans une seule image peut aussi devenir complexe du fait des dépendances, versions de programmes différents ou même des noyaux Linux différents utilisés pour deux applications distinctes.

3.4.2 Docker : implémentation et limites

L'implémentation actuelle de Docker au sein d'Armadillo **utilise les options par défaut**. Ces options permettent notamment d'activer un certain nombre de restrictions, mais ne sont pas toujours utiles en local, car les composantes matérielles et techniques sont simples. L'utilisation de Docker sur l'infonuagique devra offrir plus d'options et permettre à l'utilisateur d'adapter son application s'il le souhaite.

La sécurité est une problématique souvent soulevée dans le cadre des conteneurs et de leur accessibilité tant au niveau de la mémoire que l'accès aux services qui le composent. On peut noter qu'il est possible d'offrir une meilleure sécurité et gouvernance, à l'extérieur des conteneurs. Placer ses services à l'extérieur du conteneur réduit considérablement les risques. Cette approche peut être jumelée à la possibilité de monter les systèmes de fichiers en lecture seule. Ainsi, les services de sécurité et de gouvernance sont propres à une plateforme et non à une application. De plus, l'exécution de conteneurs par défaut est relativement sécuritaire, surtout si les actions et accès au conteneur ne se font pas en tant qu'administrateur (Bui, 2015). La **sécurité** est particulièrement importante dans le cadre de conteneurs nécessitant des interactions avec d'autres conteneurs ou même Internet. Dans le cas des applications utilisées jusqu'à présent dans Armadillo, il est relativement

simple d'offrir une sécurité importante en bloquant toute interaction avec l'extérieur. Les applications sont pour le moment à usage unique pour le temps limité de l'action du programme.
Cette sécurité nécessitera d'être élevée lors du lancement des applications sur l'infonuagique. Il sera alors préférable lors de la création du conteneur de supprimer les accès inutiles aux machines virtuelles.

Les **librairies Docker-Java et Docker-Client** sont parfois construites d'une manière peu intuitive ou éloignées de la couche classique du client Docker. Ceci rend leur intégration complexe et déstabilisante les premiers temps.

Le **choix de la librairie Docker-Java** n'a peut-être pas été le meilleur. En effet, cette librairie bien que stable semble maintenant peu suivi par la communauté. Elle s'est également arrêtée à une ancienne version de l'*API* de Docker. De plus, il ne semble plus y avoir de leader dans la démarche. Il serait peut-être bon d'intégrer et d'utiliser la librairie Docker-Client qui semble plus dynamique maintenant.

Nous avons vu dans le chapitre 2, que l'utilisation de Docker passait par des images souvent présentes sur l'Internet. Un des prochains développements serait la **création d'un dépôt local d'images** permettant le lancement en local des programmes dockerisés directement depuis Armadillo sans avoir besoin d'un accès internet.

Enfin, il est aussi notable que la solution actuelle **stoppe et supprime les conteneurs après chaque utilisation**. Or, permettre un usage multiple du même conteneur dans un même flux de travail optimise les étapes de lancement puis de suppression des conteneurs. En effet, l'arrêt d'un conteneur entraîne un ralentissement dans l'exécution du flux de travail. Il serait souhaitable de laisser le conteneur actif après utilisation et de ne l'arrêter qu'en fin d'exécution du flux ou bien s'il n'est plus appelé par la suite.

3.4.3 Les flux de travaux, intérêts et limites

Le **flux local** permet, avec de simples moyens, de créer et d'**obtenir facilement de premiers résultats**. Le recours au **flux nuagique** a quant à lui pour but de **faciliter la parallélisation des calculs et d'accélérer l'obtention de résultats** à travers l'infonuagique. Pour résumer, l'intérêt principal réside donc dans une première obtention de résultats localement avec peu de données afin de valider le flux et les premiers résultats pour ensuite extrapoler avec des données plus importantes et l'aide de l'infonuagique, et ce, d'un simple clic.

Le **concept de pipe sous Unix**, utilisé dans le pipeline du tutoriel Snakemake, pour transférer les résultats d'un programme à un autre n'est envisageable, sous Armadillo, qu'avec la création d'une capsule comprenant les deux programmes et travaillant conjointement au sein de la même étape. Cette approche évite pourtant le passage par l'écriture sur un fichier qui ralentit l'exécution générale du flux, mais reste trop spécifique à certains types de flux et d'enchaînement de capsules. Il reste donc complexe à envisager son déploiement sous Armadillo.

La **gestion des erreurs des flux de travail nuagique** nécessitera sûrement une longue étape de développement pour passer à l'intégration complète de ce concept. En effet, elle devra se faire sur l'ensemble des étapes proposées pour les échanges entre l'infonuagique et Armadillo sur l'ordinateur local. Elle devra aussi prendre en compte l'intégration de nouveaux systèmes dont les messages d'erreurs peuvent être différents.

La **gestion des résultats obtenus** sur l'infonuagique reste encore sommaire. Elle se base sur l'identifiant fourni par le cluster pour le travail en cours. Une fois le travail terminé, le fichier est ensuite téléchargé pour continuer le flux de travail. Cette approche implique de nombreux transferts de fichiers. Ceci n'est pas toujours indispensable surtout si les étapes suivantes s'effectuent également sur

le même cluster. Une solution serait de sauvegarder l'emplacement du fichier sur le cluster pour continuer le travail et réutiliser ce lien afin d'éviter de nouveaux échanges ou transferts.

L'**analyse du flux avant son exécution (ou une prévision applicative)** sera sûrement d'une grande aide pour éviter les flux de travaux demandant un très grand nombre d'échanges. Par exemple, un flux de travail irrégulier avec une application en local suivie d'une application en ligne et ainsi de suite, sera vraiment dommageable pour une exécution en temps raisonnable du flux. Il serait donc intéressant de proposer l'option du flux nuagique uniquement selon certains critères proposés par défaut et modulables par l'utilisateur avancé. Cette notion de prévision applicative peut être étendue au principe de limitation des échanges, de tests de la présence des fichiers en ligne, etc.

Parmi les **limites au travail en local et sur l'infonuagique** se trouve la difficulté d'appréhender tous les types possibles d'infonuagique. Nous nous sommes restreints aux *clusters* proposés par Calcul Québec et ceux-ci ne permettent pas d'utiliser des conteneurs. Lorsque cette restriction disparaîtra, il sera alors possible d'ouvrir plus facilement l'accès en testant non plus la présence des applications, mais simplement en chargeant les conteneurs sur l'infonuagique et en les exécutant de la même manière qu'utilisé en local.

Dans la version actuelle d'Armadillo, la mise en place de **la connexion** entre la station locale et la station distante (le nuage) **n'est pas automatisée**. L'utilisateur a donc une étape additionnelle avant l'utilisation du flux nuagique.

Enfin, une autre limite importante réside dans **le transfert des données**. Dans l'implémentation actuelle, les données transitent de la machine locale jusqu'aux clusters. Ce transport n'est pas optimisé et pourrait être grandement amélioré

grâce à l'utilisation de systèmes de compression préalable ou l'utilisation du service web *Globus* ou *BBCP* par exemple.

3.5 Conclusion

Dans ce chapitre, nous avons présenté Armadillo2.0 et son encapsulation des programmes, son intégration de Docker comme gestionnaire de conteneur et ses différents flux de travaux. Nous avons également présenté un exemple qui reprend toutes les options possibles de flux proposés par Armadillo. Enfin, nous avons également présenté une discussion autour des nouvelles possibilités d'Armadillo 2.0.

CONCLUSION

Armadillo2.0 se place désormais comme une très bonne alternative dans l'exécution de flux de travaux scientifiques y compris ceux comprenant des SHD. Il est maintenant possible d'effectuer également toutes sortes de flux à la fois en local et sur un système distribué.

Armadillo2.0 remplit l'ensemble des critères décrits dans le programme Objectif du chapitre 1 soit :

- a une installation facile,
- a une installation facile de nouveaux programmes,
- propose une interface utilisateur facile d'utilisation,
- supporte de nombreux systèmes d'exploitation sans restriction,
- peut être exécuté localement ou sur un serveur et sur un système de calcul distribué en ligne de commande,
- propose nativement des programmes SHD et d'autres programmes de bioinformatique,
- peut être exécuté nativement sur un système de calcul distribué,
- propose de déploiement d'applications par l'intermédiaire de conteneurs.

Le programme Armadillo2.0 est disponible sur un dépôt GitHub à l'adresse : `https://github.com/JeGoi/armadillo2`. Le programme d'aide à l'installation de capsules dans Armadillo est présent à cette adresse : `https://github.com/JeGoi/IPa2`

Les principales contributions

Ce travail permet d'offrir aux prochains programmeurs d'Armadillo une nouvelle approche de création et de maintien des capsules pour les programmes déployés dans Armadillo. Il propose également l'intégration de Docker afin de déployer des applications nativement développées sous Linux pour d'autres systèmes d'exploitation sans installations de la part des utilisateurs. Enfin, il a été mis en place une preuve de concept du flux nuagique se basant sur un flux de travail local et exécutant les applications sur l'infonuagique.

Pour le côté utilisateur, ce travail a permis une homogénéisation des capsules, ainsi qu'une gestion des fichiers différente sans alourdir la base de données d'Armadillo.

Perspectives

Pour la partie Docker, il reste quelques points à améliorer parmi lesquels on peut citer l'installation automatique de Docker selon le système d'exploitation hôte, l'intégration des options de Docker, une réflexion sur l'optimisation des conteneurs Docker pour les applications de types SHD et une réflexion sur l'approche multiconteneur ou uniconteneur pour les applications.

La mise en place du flux de travail nuagique reste préliminaire et certains points peuvent être améliorés. Parmi ceux-ci, on peut citer la gestion des erreurs, les conditions des applications sur l'infonuagique, la création par les utilisateurs d'une gestion des clusters, la mise en place de clés (publiques et privées) via l'interface utilisateur et la gestion du transfert de fichier.

Le déploiement de conteneurs temporaire sur le nuage reste souvent problématique pour des questions de sécurité. Il serait intéressant de se pencher sur de nouvelles solutions de conteneur comme Singularity (Kurtzer *et al.*, 2017) ou encore CoreOS rtk [4].

4. CoreOS rtk container (https ://coreos.com/rkt/) is an application container engine developed for modern production cloud-native environments.

APPENDICE A

INTÉGRATION DES PROGRAMMES SHD DANS ARMADILLO

A.1 Exemple de fichier YAML pour le programme EMBOSS chips

```
Program:
    name                : EMBOSS chips
    exitValue           : 0
    executablePaths:
      ExecutableLinux   : /usr/bin/docker
      ExecutableMacOSX  : docker
      Executable        : docker
    menu                : NGS<>EMBOSS
    numImputs           : 1
    outputPath          : ./results/EMBOSS/chips/
    helpSupplementary   : Codon usage statistics Nucleotide sequence(s) filename
    and optional format, or reference (input USA)
    website             : http://emboss.sourceforge.net/apps/cvs/emboss/apps/
    chips.html
Docker:
    imageName       : jego/emboss
    cmd             : chips --auto
    sharedFolder    : /data
Inputs:
  - type:                   FastaFile
    connector:              2
    connectorText:          Sequence
    OneConnectorOnlyFor:    2
    command2Call:           --seqall
    extention:              .fasta
Outputs:
  - type:           ChipsFile
```

```
    connectorText:  ChipsFile
    command2Call:   -outfile
    extention:      .chips
Menus:
  - name    : Default Options
    isMenu  : true
    isTab   : false
    help    : Default Options
  - name    : Advanced Options
    isMenu  : true
    isTab   : false
    help    : Advanced Options
    Panel:
      - tab     : Advanced Options 1
        Arguments:
          - name        : -nosum
            cType       : box
            tooltip     : Not Sum codons over all sequences
            oppositeTo  :
                - sum
          - name        : -sum
            cType       : box
            tooltip     : Sum codons over all sequences
            oppositeTo  :
                - nosum
AfterProcess:
    - modifications:
        - None
```

A.2 Exemple de Dockerfile pour le programme miRcheck

```
1  ##################################################################
2  # Dockerfile
3  # Software :          miRcheck
4  # Software Version :
5  # Description :       miRcheck       and various perl scripts useful in identfying plant
       miRNA genes .
6  # Website :           http          :// bartellab . wi . mit . edu / softwareDocs /
7  # Files :             http            :// bartellab . wi . mit . edu / softwareDocs / miRcheck . tar
8  # Tags :              Proteomics              | Genomics | General
9  # Provides :          miRcheck
10 #
11 # Run :                perl              ~/ script_you_need [ options ]
12 # Run :
13 # If needed , copy new models in model folder
14 # Copy :
15 # Modifications :     Few     files have been modified during installation to add
       option or to update files with current OS .
16 ##################################################################
17
18 # Base docker image
19 FROM ubuntu:14.04
20 MAINTAINER test@armadillo.uqam
21 ARG DEBIAN_FRONTEND=noninteractive
22
23 # Install miRcheck and dependencies
24 RUN apt-get clean
25
26 # Prepare installation
27 RUN apt-get -qq update \
28  && apt-get -qqy install \
29     --no-install-recommends \
30     software-properties-common \
31     wget \
32     g++ \
33     emboss=6.6.0-1 \
34  && apt-get clean \
35  && apt-get purge \
36  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
37
38 #*/
```

```
39 # Install vienna for fold_inverted_repeats .pl
40 RUN apt-add-repository -y ppa:j-4/vienna-rna > /dev/null 2>&1 \
41  && apt-get -qq update \
42  && apt-get -qqy install \
43     vienna-rna \
44  && apt-get clean \
45  && apt-get purge \
46  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
47
48 #*/
49 # Get miRcheck
50 RUN cd ~ \
51  && wget --quiet http://bartellab.wi.mit.edu/softwareDocs/miRcheck.tar \
52  && tar -xf miRcheck.tar \
53  && mv ./_miRcheck_scripts ./miRcheck \
54  && rm -f miRcheck.tar
55
56 # Install scan_for_matches
57 RUN cd ~ \
58  && wget --quiet http://www.theseed.org/servers/downloads/scan_for_matches.tgz \
59  && tar -xzf scan_for_matches.tgz \
60  && cd scan_for_matches \
61  && gcc -O -o scan_for_matches ggpunit.c scan_for_matches.c \
62  && chmod +x scan_for_matches \
63  && chmod 777 scan_for_matches \
64  && ln -sv scan_for_matches /usr/bin/scan_for_matches \
65  && rm -f ../scan_for_matches.tgz
66
67 USER root
68 ENV PATH="$PATH:/root/scan_for_matches"
69
70 # Update miRcheck lib localization
71 RUN cd ~/miRcheck \
72  && sed -i 's/\/home2\/mjonesrh/\/root/g' evaluate_miRNA_candidates.pl \
73  && sed -i 's/\/home2\/mjonesrh/\/root/g' extract_einverted_20mers.pl \
74  && sed -i 's/\/home2\/mjonesrh/\/root/g' fold_inverted_repeats.pl \
75  && sed -i 's/\/home2\/mjonesrh\/_miRNA_search\/_scripts\///g'
       fold_inverted_repeats.pl \
76  && sed -i 's/\/home2\/mjonesrh/\/root/g' retrieve_genomic_regions.pl \
77  && sed -i 's/\/home2\/mjonesrh/\/root/g' run_einverted.pl \
78  && sed -i 's/\/home2\/mjonesrh\/exe\/linuxexe\/RNAfold_original/RNAfold/g'
       miRcheck.pm \
```

```
 && sed -i ' s /\/ home2\/ mjonesrh /\/ root / g ' test_mircheck.pl

# Add options in extract_einverted_20mers . pl
RUN cd ~/miRcheck \
  && sed -i ' 26 i \
while ( @ARGV){\n\
    $thisarg = shift @ARGV ;\n\
    if ( $thisarg eq " - unpair " ) { $MAX_UNPAIR=shift @ARGV ;} \ n\
    if ( $thisarg eq " - star_unpair " ) { $MAX_STAR_UNPAIR=shift @ARGV ;} \ n\
    if ( $thisarg eq " - size_diff " ) { $MAX_SIZEDIFFERENCE=shift @ARGV ;} \ n\
    if ( $thisarg eq " - mir_bulge ") { $MAX_MIR_GAP=shift @ARGV ;} \ n\
    if ( $thisarg eq " - star_bulge ") { $MAX_STAR_GAP=shift @ARGV ;} \ n\
    if ( $thisarg eq " - fback_min ") { $MIN_FBACK_SIZE=shift @ARGV ;} \ n\
    if ( $thisarg eq " - ass ") { $MAX_MIR_AS_BULGE=shift @ARGV ;} \ n\
    if ( $thisarg eq " - min_unpair ") { $MIN_UNPAIR=shift @ARGV ;} \ n\
    if ( $thisarg eq " - bp_ext ") { $BP_EXTENSION=shift @ARGV ;} \ n\
    if ( $thisarg eq " - max_one_nt ") { $MAX_ONE_NT=shift @ARGV ;} \ n\
    if ( $thisarg eq " - max_two_nt ") { $MAX_TWO_NT=shift @ARGV ;} \ n\
    if ( $thisarg eq " - block ") { $BLOCK = shift @ARGV ;} \ n\
} \ n\n\
'  extract_einverted_20mers.pl

# Add options in fold_inverted_repeats . pl
RUN cd ~/miRcheck \
  && sed -i ' 14 i \
while ( @ARGV){\n\
    $thisarg = shift @ARGV ;\n\
    if ( $thisarg eq " - flanking_nt " ) { $FLANKING_NT=shift @ARGV ;} \ n\
} \ n\n\
'  fold_inverted_repeats.pl

# Add options in run_einverted . pl
RUN cd ~/miRcheck \
  && sed -i ' 28 i \
\n\
while ( @ARGV){\n\
    $thisarg = shift @ARGV ;\n\
    if ( $thisarg eq " - win " ) { $WIN=shift @ARGV ;} \ n\
    if ( $thisarg eq " - step " ) { $step=shift @ARGV ;} \ n\
    if ( $thisarg eq " - min_pct " ) { $MIN_PCT=shift @ARGV ;} \ n\
    if ( $thisarg eq " - min_arm ") { $MIN_ARM=shift @ARGV ;} \ n\
    if ( $thisarg eq " - gap ") { $GAP=shift @ARGV ;} \ n\
```

```
121     if ( $thisarg eq " -tresh ") { $THRESH=shift @ARGV ;} \ n\
122     if ( $thisarg eq " -m") { $M=shift @ARGV ;} \ n\
123     if ( $thisarg eq " -mm") { $MM=shift @ARGV ;} \ n\
124     if ( $thisarg eq " -dist ") { $DIST=shift @ARGV ;} \ n\
125 } \ n\n\
126 ' run_einverted.pl
127
128 # Add options in run_patscan .pl
129 RUN cd ~/miRcheck \
130   && sed -i".bak" ' 8,10d' run_patscan.pl \
131   && sed -i ' s/ARGV\[5\]/ARGV\[2\]/' run_patscan.pl \
132   && sed -i".bak" ' 10d' run_patscan.pl \
133   && sed -i ' 10i \
134 $mismatches = 0; \ n\
135 $deletions = 0; \ n\
136 $insertions = 0; \ n\
137 \n\
138 while ( @ARGV){\n\
139     $thisarg = shift @ARGV ;\n\
140     if ( $thisarg eq " -mismatches" ) { $mismatches=shift @ARGV ;} \ n\
141     if ( $thisarg eq " -deletions" ) { $deletions = shift @ARGV ;} \ n\
142     if ( $thisarg eq " -insertions" ) { $insertions=shift @ARGV ;} \ n\
143 } \ n\n\
144 if ( not ( $outfile ) ){print " usage: simple_patscan .pl query .fa database .fa outfile -
      mismatches 0 - deletions 0 - insertions 0 \\ n"; exit ;} \ n\
145 \n\
146 ' run_patscan.pl
147
148 # Add options in retrieve_genomic_regions .pl
149 RUN cd ~/miRcheck \
150   && sed -i".bak" ' 4,12d' retrieve_genomic_regions.pl \
151   && sed -i ' 4i \
152 # usage : retrieve_genomic_regions .pl sample_miRNA_matches sample_genome .fa
      sample_miRNA_matches_genomic - win window_size \ n\
153 \n\
154 $hitfile = $ARGV[0]; # list of sequences and genomic coorinates \n\
155 $genome = $ARGV [1]; # Fasta file of genomic sequence \n\
156 $outfile = $ARGV [2]; # name of outputfile \n\
157 $WIN = 0; # number of flanking nucleotides to add to each side of putative miRNA
      \n\
158 \n\
159 while ( @ARGV){\n\
```

```
160     $thisarg = shift @ARGV ;\n\
161     if ( $thisarg eq "-win" ) { $WIN=shift @ARGV ;} \n\
162 } \n\n\
163 \n\
164 if ( not($outfile)){print "usage: retrieve_genomic_regions.pl sample_miRNA_matches
        sample_genome.fa sample_miRNA_matches_genomic -win window_size \n"; exit ;}\n\
165 ' retrieve_genomic_regions.pl
166
167 # Create rep
168 RUN mkdir ~/data
169
170 RUN cd ~ \
171  && chmod -R 777 ./miRcheck
172
173 WORKDIR /root/data
```

APPENDICE B

EXEMPLE DE FLUX SHD DANS ARMADILLO

B.1 Le flux de travail de Snakemake

B.1.1 Code pour le tutoriel débutant

```
SAMPLES = ["A", "B"]

rule all:
    input:
        "report.html"

#1
rule bwa_map:
    input:
        "data/genome.fa",
        "data/samples/{sample}.fastq"
    output:
        "mapped_reads/{sample}.bam"
    shell:
        "bwa mem {input} | samtools view -Sb - > {output}"

#2
rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam"
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
        "-O bam {input} > {output}"
```

```
26
27 #3
28 rule samtools_index:
29     input:
30         "sorted_reads/{sample}.bam"
31     output:
32         "sorted_reads/{sample}.bam.bai"
33     shell:
34         "samtools index {input}"
35
36 #4
37 rule bcftools_call:
38     input:
39         fa="data/genome.fa",
40         bam=expand("sorted_reads/{sample}.bam",sample=SAMPLES),
41         bai=expand("sorted_reads/{sample}.bam.bai",sample=SAMPLES)
42     output:
43         "calls/all.vcf"
44     shell:
45         "samtools mpileup -g -f {input.fa} {input.bam} | "
46         "bcftools call -mv - > {output}"
47
48 #5
49 rule report:
50     input:
51         "calls/all.vcf"
52     output:
53         "report.html"
54     run:
55         from snakemake.utils import report
56         with open(input[0]) as vcf:
57             n_calls = sum(1 for l in vcf if not l.startswith("#"))
58
59         report("""
60         An example variant calling workflow
61         ===================================
62
63         Reads were mapped to the Yeast
64         reference genome and variants were called jointly with
65         SAMtools/BCFtools.
66         This resulted in {n_calls} variants (see Table T1_).
67         """, output[0], T1=input[0])
```

B.1.2 Code pour le tutoriel avancé

```
configfile: "config.yaml"

rule all:
    input:
        "report.html"

#1
rule bwa_map:
    input:
        "data/genome.fa",
        lambda wildcards: config["samples"][wildcards.sample]
    output:
        temp("mapped_reads/{sample}.bam")
    params:
        rg="@RG\tID:{sample}\tSM:{sample}"
    log:
        "logs/bwa_mem/{sample}.log"
    threads: 8
    shell:
        "(bwa mem -R '{params.rg}' -t {threads} {input} | "
        "samtools view -Sb - > {output}) 2> {log}"

#2
rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        protected("sorted_reads/{sample}.bam")
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
        "-O bam {input} > {output}"

#3
rule samtools_index:
    input:
        "sorted_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam.bai"
    shell:
        "samtools index {input}"
```

```
41
42 #4
43 rule bcftools_call:
44     input:
45         fa="data/genome.fa",
46         bam=expand("sorted_reads/{sample}.bam",
47           sample=config["samples"]),
48         bai=expand("sorted_reads/{sample}.bam.bai",
49           sample=config["samples"])
50     output:
51         "calls/all.vcf"
52     shell:
53         "samtools mpileup -g -f {input.fa} {input.bam} | "
54         "bcftools call -mv - > {output}"
55
56 #5
57 rule report:
58     input:
59         "calls/all.vcf"
60     output:
61         "report.html"
62     run:
63         from snakemake.utils import report
64         with open(input[0]) as vcf:
65             n_calls = sum(1 for l in vcf if not l.startswith("#"))
66
67         report("""
68         An example variant calling workflow
69         ===================================
70
71         Reads were mapped to the Yeast
72         reference genome and variants were called jointly with
73         SAMtools/BCFtools.
74
75         This resulted in {n_calls} variants (see Table T1_).
76         """, output[0], T1=input[0])
```

B.1.3 Code SnakemakeFile équivalent au flux sous Armadillo

```
SAMPLES = ["A", " B"]

rule all:
    input:
        "report.html"

#1
rule bwa_map:
    input:
        "data/genome.fa", " data/samples/{sample}.fastq"
    output:
        "mapped_reads/{sample}.sam"
    shell:
        "bwa mem {input} > {output}"

#2
rule samtools_view:
    input:
        "mapped_reads/{sample}.sam"
    output:
        "mapped_reads/{sample}.bam"
    shell:
        "samtools view -Sb {input} > {output}"

#3
rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam"
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
        "-O bam {input} > {output}"

#4
rule samtools_index:
    input:
        "sorted_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam.bai"
```

```
41      shell:
42          "samtools index {input}"
43
44  #5
45  rule samtools_mpileup:
46      input:
47          fa="data/genome.fa", bam=expand("sorted_reads/{sample}.bam",
48          sample=SAMPLES),
49      output:
50          "calls/all.bcf"
51      shell:
52          "samtools mpileup -g -f {input.fa} {input.bam} --output
53          {output}"
54
55  #6
56  rule bcftools_call:
57      input:
58          bcf="calls/all.bcf"
59      output:
60          "calls/all.vcf"
61      shell:
62          "bcftools call -mv {input.bcf} --output {output}"
63
64  #7
65  rule report:
66      input:
67          "calls/all.vcf"
68      output:
69          "report.html"
70      run:
71          from snakemake.utils import report with open(input[0])
72          as vcf:
73              n_calls = sum(1 for l in vcf if not l.startswith("#"))
74
75          report("""An example variant calling workflow
76          ==========================================
77
78          Reads were mapped to the Yeast reference genome and variants
79          were called jointly with SAMtools/BCFtools.
80
81          This resulted in {n_calls} variants (see Table T1_).
82          """, output[0], T1=input[0])
```

B.2 Les logs de flux de travaux

B.2.1 Log d'un flux de travail local avec Armadillo et Docker

```
1  ********************************************************************************
2  Armadillo v2.0 New Execution started -Running
3   ./armadillo2-master/projects/New_Untitled_WF.db workflow. -Started at 2017-08-28
       01:18:44
4  ********************************************************************************
5
6  Running Load Files...
7      Initialization...
8      Running [Load Files Genome Reference]
9      Checking program requirements...
10     Creating inputs...
11     Creating commandline...
12 <-Program Output->
13 <-End Program Output ->
14     Program Exit Value: 0
15     Parsing outputs...
16
17 ******************************
18
19 Running Load Files...
20     Initialization...
21     Running [Load Files C Sample Fastq]
22     Checking program requirements...
23     Creating inputs...
24     Creating commandline...
25 <-Program Output->
26 <-End Program Output ->
27     Program Exit Value: 0
28     Parsing outputs...
29
30 ******************************
31
32 Running Load Files...
33     Initialization...
34     Running [Load Files B Sample Fastq]
35     Checking program requirements...
36     Creating inputs...
37     Creating commandline...
```

```
38 <-Program Output->
39 <-End Program Output ->
40     Program Exit Value: 0
41     Parsing outputs...
42
43 ******************************
44
45 Running Load Files...
46     Initialization...
47     Running [Load Files A Sample Fastq]
48     Checking program requirements...
49     Creating inputs...
50     Creating commandline...
51 <-Program Output->
52 <-End Program Output ->
53     Program Exit Value: 0
54     Parsing outputs...
55
56 ******************************
57
58 Running bwa mem...
59     Initialization...
60     Running [bwa mem]
61     Checking program requirements...
62     DockerInitCommandLine: $ /usr/bin/docker run  -v ./inputs:/data/inputs/1/ -v
       ./armadillo2-master/results/BWA/mem/2017-7-28_bwa_mem:/data/outputs/ -v ./
       inputs/samples:/data/inputs/3/ --name bwa_mem_armadilloWF_0 -di jego/bwa
63     Creating inputs...
64     Creating commandline...
65     DockerRunningCommandLine: $ bwa mem /data/inputs/1/genome.fa /data/inputs/3/A
       .fastq > /data/outputs/OutputOf_A.fastq.sam
66     DockerExecution: $ /usr/bin/docker exec -i bwa_mem_armadilloWF_0 sh -c './
       dockerBash.sh'
67     Running program...
68 <-Program Output->
69     [M::bwa_idx_load_from_disk] read 0 ALT contigs
70     [M::process] read 25000 sequences (2525000 bp)...
71     [M::mem_process_seqs] Processed 25000 reads in 0.948 CPU sec, 0.952 real sec
       [main] Version: 0.7.12-r1039 [main] CMD: bwa mem /data/inputs/1/genome.fa /
       data/inputs/3/A.fastq [main] Real time: 1.001 sec; CPU: 0.964 sec <-End
       Program Output ->
72     Program Exit Value: 0
```

```
73      Parsing outputs...
74
75  ******************************
76
77  Running samtools view...
78      Initialization...
79      Running [samtools view]
80      Checking program requirements...
81      DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        BWA/mem/2017-7-28_bwa_mem:/data/inputs/0/ -v ./armadillo2-master/results/
        SAMTOOLS/view/2017-7-28_samtools_view:/data/outputs/ --name samtools_view_
        armadilloWF_2 -di jego/samtools
82      Creating inputs...
83      Creating commandline...
84      DockerRunningCommandLine: $ samtools view -b -S  /data/inputs/0/OutputOf_A.
        fastq.sam  >/data/outputs/OutputOf_A.fastq.sam.bam
85      DockerExecution: $ /usr/bin/docker exec -i samtools_view_armadilloWF_2 sh -c
        './dockerBash.sh'
86      Running program...
87  <-Program Output->
88  <-End Program Output ->
89      Program Exit Value: 0
90      Parsing outputs...
91
92  ******************************
93
94  Running samtools sort...
95      Initialization...
96      Running [samtools sort]
97      Checking program requirements...
98      DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        SAMTOOLS/sort/2017-7-28_samtools_sort:/data/outputs/ -v ./armadillo2-master/
        results/SAMTOOLS/view/2017-7-28_samtools_view:/data/inputs/1/ --name samtools
        _sort_armadilloWF_0 -di jego/samtools
99      Creating inputs...
100     Creating commandline...
101     DockerRunningCommandLine: $ samtools sort -O bam  /data/inputs/1/OutputOf_A.
        fastq.sam.bam -o /data/outputs/OutputOf_A.fastq.sam.bam
102     DockerExecution: $ /usr/bin/docker exec -i samtools_sort_armadilloWF_0 sh -c
        './dockerBash.sh'
103     Running program...
104 <-Program Output->
```

```
105 <-End Program Output ->
106     Program Exit Value: 0
107     Parsing outputs...
108
109 ****************************
110
111 Running bwa mem...
112     Initialization...
113     Running [bwa mem]
114     Checking program requirements...
115     DockerInitCommandLine: $ /usr/bin/docker run  -v ./inputs:/data/inputs/1/ -v
        ./armadillo2-master/results/BWA/mem/2017-7-28_bwa_mem:/data/outputs/ -v ./
        inputs/samples:/data/inputs/3/ --name bwa_mem_armadilloWF_0 -di jego/bwa
116     Creating inputs...
117     Creating commandline...
118     DockerRunningCommandLine: $ bwa mem   /data/inputs/1/genome.fa /data/inputs
        /3/C.fastq > /data/outputs/OutputOf_C.fastq.sam
119     DockerExecution: $ /usr/bin/docker exec -i bwa_mem_armadilloWF_0 sh -c './
        dockerBash.sh'
120     Running program...
121 <-Program Output->
122     [M::bwa_idx_load_from_disk] read 0 ALT contigs
123     [M::process] read 25000 sequences (2525000 bp)...
124     [M::mem_process_seqs] Processed 25000 reads in 0.960 CPU sec, 0.963 real sec
        [main] Version: 0.7.12-r1039 [main] CMD: bwa mem /data/inputs/1/genome.fa /
        data/inputs/3/C.fastq [main] Real time: 1.011 sec; CPU: 0.976 sec <-End
        Program Output ->
125     Program Exit Value: 0
126     Parsing outputs...
127
128 ****************************
129
130 Running samtools view...
131     Initialization...
132     Running [samtools view]
133     Checking program requirements...
134     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        SAMTOOLS/view/2017-7-28_samtools_view:/data/outputs/ -v ./armadillo2-master/
        results/BWA/mem/2017-7-28_bwa_mem:/data/inputs/0/ --name samtools_view_
        armadilloWF_2 -di jego/samtools
135     Creating inputs...
136     Creating commandline...
```

```
137     DockerRunningCommandLine: $ samtools view -b -S  /data/inputs/0/OutputOf_C.
        fastq.sam  >/data/outputs/OutputOf_C.fastq.sam.bam
138     DockerExecution: $ /usr/bin/docker exec -i samtools_view_armadilloWF_2 sh -c
        './dockerBash.sh'
139     Running program...
140 <-Program Output->
141 <-End Program Output ->
142     Program Exit Value: 0
143     Parsing outputs...
144
145 *****************************
146
147 Running samtools sort...
148     Initialization...
149     Running [samtools sort]
150     Checking program requirements...
151     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        SAMTOOLS/view/2017-7-28_samtools_view:/data/inputs/1/ -v ./armadillo2-master/
        results/SAMTOOLS/sort/2017-7-28_samtools_sort:/data/outputs/ --name samtools_
        sort_armadilloWF_0 -di jego/samtools
152     Creating inputs...
153     Creating commandline...
154     DockerRunningCommandLine: $ samtools sort -O bam  /data/inputs/1/OutputOf_C.
        fastq.sam.bam  -o /data/outputs/OutputOf_C.fastq.sam.bam
155     DockerExecution: $ /usr/bin/docker exec -i samtools_sort_armadilloWF_0 sh -c
        './dockerBash.sh'
156     Running program...
157 <-Program Output->
158 <-End Program Output ->
159     Program Exit Value: 0
160     Parsing outputs...
161
162 *****************************
163
164 Running bwa mem...
165     Initialization...
166     Running [bwa mem]
167     Checking program requirements...
168     DockerInitCommandLine: $ /usr/bin/docker run  -v ./inputs:/data/inputs/1/ -v
        ./inputs/samples:/data/inputs/3/ -v ./armadillo2-master/results/BWA/mem/2017-
        7-28_bwa_mem:/data/outputs/ --name bwa_mem_armadilloWF_0 -di jego/bwa
169     Creating inputs...
```

```
170     Creating commandline...
171     DockerRunningCommandLine: $ bwa mem   /data/inputs/1/genome.fa /data/inputs
        /3/B.fastq > /data/outputs/OutputOf_B.fastq.sam
172     DockerExecution: $ /usr/bin/docker exec -i bwa_mem_armadilloWF_0 sh -c './
        dockerBash.sh'
173     Running program...
174 <-Program Output->
175     [M::bwa_idx_load_from_disk] read 0 ALT contigs
176     [M::process] read 25000 sequences (2525000 bp)...
177     [M::mem_process_seqs] Processed 25000 reads in 0.956 CPU sec, 0.959 real sec
        [main] Version: 0.7.12-r1039 [main] CMD: bwa mem /data/inputs/1/genome.fa /
        data/inputs/3/B.fastq [main] Real time: 1.008 sec; CPU: 0.976 sec <-End
        Program Output ->
178     Program Exit Value: 0
179     Parsing outputs...
180
181 ******************************
182
183 Running samtools view...
184     Initialization...
185     Running [samtools view]
186     Checking program requirements...
187     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        BWA/mem/2017-7-28_bwa_mem:/data/inputs/0/ -v ./armadillo2-master/results/
        SAMTOOLS/view/2017-7-28_samtools_view:/data/outputs/ --name samtools_view_
        armadilloWF_2 -di jego/samtools
188     Creating inputs...
189     Creating commandline...
190     DockerRunningCommandLine: $ samtools view   -b -S  /data/inputs/0/OutputOf_B.
        fastq.sam  > /data/outputs/OutputOf_B.fastq.sam.bam
191     DockerExecution: $ /usr/bin/docker exec -i samtools_view_armadilloWF_2 sh -c
        './dockerBash.sh'
192     Running program...
193 <-Program Output->
194 <-End Program Output ->
195     Program Exit Value: 0
196     Parsing outputs...
197
198 ****************************
199
200 Running samtools sort...
201     Initialization...
```

```
202     Running [samtools sort]
203     Checking program requirements...
204     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        SAMTOOLS/sort/2017-7-28_samtools_sort:/data/outputs/ -v ./armadillo2-master/
        results/SAMTOOLS/view/2017-7-28_samtools_view:/data/inputs/1/ --name samtools
        _sort_armadilloWF_0 -di jego/samtools
205     Creating inputs...
206     Creating commandline...
207     DockerRunningCommandLine: $ samtools sort   -O bam  /data/inputs/1/OutputOf_B
        .fastq.sam.bam  -o /data/outputs/OutputOf_B.fastq.sam.bam
208     DockerExecution: $ /usr/bin/docker exec -i samtools_sort_armadilloWF_0 sh -c
        './dockerBash.sh'
209     Running program...
210 <-Program Output->
211 <-End Program Output ->
212     Program Exit Value: 0
213     Parsing outputs...
214
215 ******************************
216
217 Running samtools mpileup...
218     Initialization...
219     Running [samtools mpileup]
220     Checking program requirements...
221     No BedFile found.
222     DockerInitCommandLine: $ /usr/bin/docker run  -v ./inputs:/data/inputs/1/ -v
        ./armadillo2-master/results/SAMTOOLS/sort/2017-7-28_samtools_sort:/data/
        inputs/2/ -v ./armadillo2-master/results/SAMTOOLS/mpileup/2017-7-28_samtools_
        mpileup:/data/outputs/ -v ./armadillo2-master/results/SAMTOOLS/sort/2017-7-
        28_samtools_sort:/data/inputs/5/ -v ./armadillo2-master/results/SAMTOOLS/sort
        /2017-7-28_samtools_sort:/data/inputs/4/ --name samtools_mpileup_armadilloWF
        _0 -di jego/samtools
223     Creating inputs...
224     Creating commandline...
225     DockerRunningCommandLine: $ samtools mpileup --illumina1.3+ --BCF --fasta-ref
         /data/inputs/1/genome.fa /data/inputs/2/OutputOf_A.fastq.sam.bam /data/
        inputs/4/OutputOf_C.fastq.sam.bam /data/inputs/5/OutputOf_B.fastq.sam.bam  --
        output /data/outputs/OutputOf_A.fastq.sam.bam.bcf
226     DockerExecution: $ /usr/bin/docker exec -i samtools_mpileup_armadilloWF_0 sh
        -c './dockerBash.sh'
227     Running program...
228 <-Program Output->
```

```
229 [mpileup] 3 samples in 3 input files <mpileup>
230 Set max per-file depth to 2666
231 <-End Program Output ->
232     Program Exit Value: 0
233     Parsing outputs...
234
235 ******************************
236
237 Running bcftools call...
238     Initialization...
239     Running [bcftools call]
240     Checking program requirements...
241     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        BCFTOOLS/call/2017-7-28_bcftools_call:/data/outputs/ -v ./armadillo2-master/
        results/SAMTOOLS/mpileup/2017-7-28_samtools_mpileup:/data/inputs/1/ --name
        bcftools_call_armadilloWF_0 -di jego/samtools
242     Creating inputs...
243     Creating commandline...
244     DockerRunningCommandLine: $ bcftools call --variants-only  --output-type v  -
        -multiallelic-caller /data/inputs/1/OutputOf_A.fastq.sam.bam.bcf  --output /
        data/outputs/OutputOf_A.fastq.sam.bam.bcf.vcf
245     DockerExecution: $ /usr/bin/docker exec -i bcftools_call_armadilloWF_0 sh -c
        './dockerBash.sh'
246     Running program...
247 <-Program Output->
248 Note: Neither --ploidy nor --ploidy-file given,
249 assuming all sites are diploid
250 <-End Program Output ->
251     Program Exit Value: 0
252     Parsing outputs...
253
254 ******************************
255
256 ********************************************************************************
257 Computation finished in 2m 16s Ended at 2017-08-28 01:21:00
258 ********************************************************************************
```

B.2.2 Output du flux avec Snakemake

```
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
  count jobs
  1 all
  1 bcftools_call
  2 bwa_map
  1 report
  1 samtools_mpileup
  2 samtools_sort
  2 samtools_view
  10

#1
rule bwa_map:
    input: data/genome.fa, data/samples/B.fastq output:
    mapped_reads/B.sam jobid: 9 wildcards: sample=B

[M::bwa_idx_load_from_disk] read 0 ALT contigs [M::process] read
25000 sequences (2525000 bp)...  [M::mem_process_seqs] Processed
25000 reads in 0.928 CPU sec, 0.949 real sec [main] Version:
0.7.12-r1039 [main] CMD: bwa mem data/genome.fa data/samples/B.fastq
[main] Real time : 0.988 sec; CPU: 0.956 sec Finished job 9.  1 of
10 steps (10%) done

#1
rule bwa_map:
    input: data/genome.fa, data/samples/A.fastq output:
    mapped_reads/A.sam jobid: 8 wildcards: sample=A

[M::bwa_idx_load_from_disk] read 0 ALT contigs [M::process] read
25000 sequences (2525000 bp)...  [M::mem_process_seqs] Processed
25000 reads in 0.924 CPU sec, 0.945 real sec [main] Version:
0.7.12-r1039 [main] CMD: bwa mem data/genome.fa data/samples/A.fastq
[main] Real time : 1.002 sec; CPU: 0.960 sec Finished job 8.  2 of
10 steps (20%) done

#2
rule samtools_view:
    input: mapped_reads/B.sam output: mapped_reads/B.bam jobid:
```

```
41      7 wildcards: sample=B
42
43  Finished job 7.  3 of 10 steps (30%) done
44
45  #2
46  rule samtools_view:
47      input: mapped_reads/A.sam output: mapped_reads/A.bam jobid:
48      6 wildcards: sample=A
49
50  Finished job 6.  4 of 10 steps (40%) done
51
52  #3
53  rule samtools_sort:
54      input: mapped_reads/B.bam output: sorted_reads/B.bam jobid:
55      5 wildcards: sample=B
56
57  Finished job 5.  5 of 10 steps (50%) done
58
59  #3
60  rule samtools_sort:
61      input: mapped_reads/A.bam output: sorted_reads/A.bam jobid:
62      4 wildcards: sample=A
63
64  Finished job 4.  6 of 10 steps (60%) done
65
66  #4
67  rule samtools_mpileup:
68      input: data/genome.fa, sorted_reads/A.bam, sorted_reads/B.bam
69      output: calls/all.bcf jobid: 3
70
71  [mpileup] 2 samples in 2 input files <mpileup> Set max  per- file
72  depth to 4000 Finished job 3.  7 of 10 steps (70%) done
73
74  #5
75  rule bcftools_call:
76      input: calls/all.bcf output: calls/all.vcf jobid: 2
77
78  Note: Neither --ploidy nor --ploidy-file  given, assuming all sites
79  are diploid Finished job 2.  8 of 10 steps (80%) done
80
81  rule report:
82      input: calls/all.vcf output: report.html jobid: 1
```

```
83
84 Finished job 1.  9 of 10 steps (90%) done
85
86 localrule all:
87     input: report.html jobid: 0
88
89 Finished job 0.  10 of 10 steps (100%) done
```

B.3 Output du flux sur le cluster

```
1  ********************************************************************************
2   Armadillo v2.0 New Execution started -Running
3   ./armadillo2-master/projects/New_Untitled.db workflow.  -Started at 2017-09-04
       11:13:08
4  ********************************************************************************
5
6  Running Load Files...
7      Initialization...
8      Running [Load Files Genome Reference]
9      Checking program requirements...
10     Creating inputs...
11     Creating commandline...
12     Not enough information to run on Cluster
13     Running will done on the local machine...
14 <-Program Output->
15 <-End Program Output ->
16     Program Exit Value: 0
17     Parsing outputs...
18
19 ****************************
20
21 Running Load Files...
22     Initialization...
23     Running [Load Files B Sample Fastq]
24     Checking program requirements...
25     Creating inputs...
26     Creating commandline...
27     Not enough information to run on Cluster
28     Running will done on the local machine...
29 <-Program Output->
30 <-End Program Output ->
31     Program Exit Value: 0
32     Parsing outputs...
33
34 ****************************
35
36 Running Load Files...
37     Initialization...
38     Running [Load Files A Sample Fastq]
39     Checking program requirements...
```

```
40      Creating inputs...
41      Creating commandline...
42      Not enough information to run on Cluster
43      Running will done on the local machine...
44  <-Program Output->
45  <-End Program Output ->
46      Program Exit Value: 0
47      Parsing outputs...
48
49  ******************************
50
51  Running bwa mem...
52      Initialization...
53      Running [bwa mem]
54      Checking program requirements...
55      DockerInitCommandLine: $ /usr/bin/docker run  -v ./Armadillo_SNPs_WF_test/
        inputs:/data/inputs/190/ -v ./armadillo2-master/results/bwa/mem/bwa_mem:/data
        /outputs/ -v ./Armadillo_SNPs_WF_test/inputs/samples:/data/inputs/192/ --name
         bwa_mem_armadilloWF_2 -di jego/bwa
56      Creating inputs...
57      Creating commandline...
58      DockerRunningCommandLine: $ bwa mem   /data/inputs/190/genome.fa /data/inputs
        /192/B.fastq  > /data/outputs/OutputOf_B.fastq.sam
59      DockerExecution: $ /usr/bin/docker exec -i bwa_mem_armadilloWF_2 sh -c './
        dockerBash.sh'
60
61      Test to access and get modules on cluster already done! Congratulation.
62      Can access to the cluster
63      Start Test if module is here
64      Start to create directories.
65          <-Directories created on the cluster->
66          <TIME> Time to create directories on cluster is >2 s
67      Start to send file(s)
68          <-Files sended->
69          <TIME> Time to create directories on cluster is >407 s
70      Start create PBS file , send and execute
71          <TIME> Time to create pbs file and launch it on cluster is >3 s
72          <-Program Cluster Status->
73          Running program on cluster...
74          >> PBS File on cluster contains
75              #!/bin/bash
76              #PBS -l walltime=00:05:00
```

```
77              #PBS -l nodes=1:ppn=1
78              #PBS -q qwork@mp2
79              #PBS -o ~/bwa_0.7.12/bwa_mem/stdOutFile
80              #PBS -e ~/bwa_0.7.12/bwa_mem/stdErrFile
81              module load bioinformatics/bwa/0.7.12
82              bwa mem ~/bwa_0.7.12/bwa_mem/genome.fa ~/bwa_0.7.12/bwa_mem/B.fastq >
        ~/bwa_0.7.12/bwa_mem/outputs/OutputOf_B.fastq.sam
83              a="ExitStatus>"$?"<"
84              echo $a
85          >> Command line running on cluster is
86              bwa mem ~/bwa_0.7.12/bwa_mem/genome.fa ~/bwa_0.7.12/bwa_mem/B.fastq
       > ~/bwa_0.7.12/bwa_mem/outputs/OutputOf_B.fastq.sam
87          >> The CLuster task number is 138017.mp2.m
88      Wait until job is done or time is up (after 68 minutes) In seconds, the time
        between two tests is 60,60,60,60,60,60,120,240,480,960,1920 So it means a
        test each minute during the first six minutes, then a test in 2, 4, 8, 16, 32
         minutes
89          <TIME> Time to wait until tasks done on the cluster is >125 s
90          <-Results downloaded from cluster->
91          <TIME> Time to donwload files from the cluster is >11 s
92
93      <-Program Output->
94      <-End Program Output ->
95      Program Exit Value: 0
96      Parsing outputs...
97
98  ****************************
99
100 Running samtools view...
101     Initialization...
102     Running [samtools view]
103     Checking program requirements...
104     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        samtools/view/samtools_view:/data/outputs/ -v ./armadillo2-master/results/bwa
        /mem/bwa_mem:/data/inputs/196/ --name samtools_view_armadilloWF_0 -di jego/
        samtools
105     Creating inputs...
106     Creating commandline...
107     DockerRunningCommandLine: $ samtools view   -b -S  /data/inputs/196/OutputOf_
        B.fastq.sam  >/data/outputs/OutputOf_B.fastq.sam.bam
108     DockerExecution: $ /usr/bin/docker exec -i samtools_view_armadilloWF_0 sh -c
        './dockerBash.sh'
```

```
109
110     Test to access and get modules on cluster already done! Congratulation.
111     Can access to the cluster
112     Start Test if module is here
113     Start to create directories.
114         <-Directories created on the cluster->
115         <TIME> Time to create directories on cluster is >1 s
116     Start to send file(s)
117         <-Files sended->
118         <TIME> Time to create directories on cluster is >128 s
119     Start create PBS file, send and execute
120         <TIME> Time to create pbs file and launch it on cluster is>4 s
121         <-Program Cluster Status->
122         Running program on cluster...
123         >> PBS File on cluster contains
124             #!/bin/bash
125             #PBS -l walltime=00:05:00
126             #PBS -l nodes=1:ppn=1
127             #PBS -q qwork@mp2
128             #PBS -o ~/samtools_1.3.2/samtools_view/stdOutFile
129             #PBS -e ~/samtools_1.3.2/samtools_view/stdErrFile
130             module load bioinformatics/samtools/1.3.2
131             samtools view -b -S  ~/samtools_1.3.2/samtools_view/OutputOf_B.fastq.
    sam  >~/samtools_1.3.2/samtools_view/outputs/OutputOf_B.fastq.sam.bam
132             a="ExitStatus>"$?"<"
133             echo $a
134         >> Command line running on cluster is
135             samtools view   -b -S  ~/samtools_1.3.2/samtools_view/OutputOf_B.
    fastq.sam  >~/samtools_1.3.2/samtools_view/outputs/OutputOf_B.fastq.sam.bam
136         >> The CLuster task number is 138018.mp2.m
137     Wait until job is done or time is up (after 68 minutes) In seconds, the time
        between two tests is 60,60,60,60,60,60,120,240,480,960,1920 So it means a
        test each minute during the first six minutes, then a test in 2, 4, 8, 16, 32
         minutes
138         <TIME> Time to wait until tasks done on the cluster is>125 s
139         <-Results downloaded from cluster->
140         <TIME> Time to donwload files from the cluster is >6 s
141
142     <-Program Output->
143     <-End Program Output ->
144     Program Exit Value: 0
145     Parsing outputs...
```

```
146
147 ****************************
148
149 Running samtools sort...
150     Initialization...
151     Running [samtools sort]
152     Checking program requirements...
153     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        samtools/view/samtools_view:/data/inputs/199/ -v ./armadillo2-master/results/
        samtools/sort/samtools_sort:/data/outputs/ --name samtools_sort_armadilloWF_0
         -di jego/samtools
154     Creating inputs...
155     Creating commandline...
156     DockerRunningCommandLine: $ samtools sort -O bam /data/inputs/199/OutputOf_B.
        fastq.sam.bam -o /data/outputs/OutputOf_B.fastq.sam.bam
157     DockerExecution: $ /usr/bin/docker exec -i samtools_sort_armadilloWF_0 sh -c
        './dockerBash.sh'
158
159     Test to access and get modules on cluster already done! Congratulation.
160     Can access to the cluster
161     Start Test if module is here
162     Start to create directories.
163         <-Directories created on the cluster->
164         <TIME> Time to create directories on cluster is >1 s
165     Start to send file(s)
166         <-Files sended->
167         <TIME> Time to create directories on cluster is >51 s
168     Start create PBS file, send and execute
169         <TIME> Time to create pbs file and launch it on cluster is>3 s
170         <-Program Cluster Status->
171         Running program on cluster...
172         >> PBS File on cluster contains
173             #!/bin/bash
174             #PBS -l walltime=00:05:00
175             #PBS -l nodes=1:ppn=1
176             #PBS -q qwork@mp2
177             #PBS -o ~/samtools_1.3.2/samtools_sort/stdOutFile
178             #PBS -e ~/samtools_1.3.2/samtools_sort/stdErrFile
179             module load bioinformatics/samtools/1.3.2
180             samtools sort -O bam ~/samtools_1.3.2/samtools_sort/OutputOf_B.fastq.
        sam.bam  -o ~/samtools_1.3.2/samtools_sort/outputs/OutputOf_B.fastq.sam.bam
181             a="ExitStatus>"$?"<"
```

```
182             echo $a
183         >> Command line running on cluster is
184             samtools sort -O bam ~/samtools_1.3.2/samtools_sort/OutputOf_B.fastq.
    sam.bam  -o ~/samtools_1.3.2/samtools_sort/outputs/OutputOf_B.fastq.sam.bam
185         >> The CLuster task number is 138020.mp2.m
186     Wait until job is done or time is up (after 68 minutes) In seconds, the time
    between two tests is 60,60,60,60,60,60,120,240,480,960,1920 So it means a
    test each minute during the first six minutes, then a test in 2, 4, 8, 16, 32
     minutes
187         <TIME> Time to wait until tasks done on the cluster is>126 s
188         <-Results downloaded from cluster->
189         <TIME> Time to donwload files from the cluster is >6 s
190
191     <-Program Output->
192     <-End Program Output ->
193     Program Exit Value: 0
194     Parsing outputs...
195
196 ******************************
197
198 Running bwa mem...
199     Initialization...
200     Running [bwa mem]
201     Checking program requirements...
202     DockerInitCommandLine: $ /usr/bin/docker run  -v ./Armadillo_SNPs_WF_test/
    inputs:/data/inputs/190/ -v ./armadillo2-master/results/bwa/mem/bwa_mem:/data
    /outputs/ -v ./Armadillo_SNPs_WF_test/inputs/samples:/data/inputs/194/ --name
     bwa_mem_armadilloWF_2 -di jego/bwa
203     Creating inputs...
204     Creating commandline...
205     DockerRunningCommandLine: $ bwa mem   /data/inputs/190/genome.fa /data/inputs
    /194/A.fastq  > /data/outputs/OutputOf_A.fastq.sam
206     DockerExecution: $ /usr/bin/docker exec -i bwa_mem_armadilloWF_2 sh -c './
    dockerBash.sh'
207
208     Test to access and get modules on cluster already done! Congratulation.
209     Can access to the cluster
210     Start Test if module is here
211     Start to create directories.
212         <-Directories created on the cluster->
213         <TIME> Time to create directories on cluster is >1 s
214     Start to send file(s)
```

```
215         <-Files sended->
216         <TIME> Time to create directories on cluster is >406 s
217     Start create PBS file , send and execute
218         <TIME> Time to create pbs file and launch it on cluster is>4 s
219         <-Program Cluster Status->
220         Running program on cluster...
221         >> PBS File on cluster contains
222             #!/bin/bash
223             #PBS -l walltime=00:05:00
224             #PBS -l nodes=1:ppn=1
225             #PBS -q qwork@mp2
226             #PBS -o ~/bwa_0.7.12/bwa_mem/stdOutFile
227             #PBS -e ~/bwa_0.7.12/bwa_mem/stdErrFile
228             module load bioinformatics/bwa/0.7.12
229             bwa mem ~/bwa_0.7.12/bwa_mem/genome.fa ~/bwa_0.7.12/bwa_mem/A.fastq >
       ~/bwa_0.7.12/bwa_mem/outputs/OutputOf_A.fastq.sam
230             a="ExitStatus>"$?"<"
231             echo $a
232         >> Command line running on cluster is
233             bwa mem ~/bwa_0.7.12/bwa_mem/genome.fa ~/bwa_0.7.12/bwa_mem/A.fastq
      > ~/bwa_0.7.12/bwa_mem/outputs/OutputOf_A.fastq.sam
234         >> The CLuster task number is 138022.mp2.m
235     Wait until job is done or time is up (after 68 minutes) In seconds, the time
        between two tests is 60,60,60,60,60,60,120,240,480,960,1920 So it means a
        test each minute during the first six minutes, then a test in 2, 4, 8, 16, 32
         minutes
236         <TIME> Time to wait until tasks done on the cluster is>126 s
237         <-Results downloaded from cluster->
238         <TIME> Time to donwload files from the cluster is >11 s
239
240     <-Program Output->
241     <-End Program Output ->
242     Program Exit Value: 0
243     Parsing outputs...
244
245 ****************************
246
247 Running samtools view...
248     Initialization...
249     Running [samtools view]
250     Checking program requirements...
251     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
```

```
    bwa/mem/bwa_mem:/data/inputs/205/ -v ./armadillo2-master/results/samtools/
    view/samtools_view:/data/outputs/ --name samtools_view_armadilloWF_0 -di jego
    /samtools
252 Creating inputs...
253 Creating commandline...
254 DockerRunningCommandLine: $ samtools view   -b -S  /data/inputs/205/OutputOf_
    A.fastq.sam  > /data/outputs/OutputOf_A.fastq.sam.bam
255 DockerExecution: $ /usr/bin/docker exec -i samtools_view_armadilloWF_0 sh -c
    './dockerBash.sh'
256
257 Test to access and get modules on cluster already done! Congratulation.
258 Can access to the cluster
259 Start Test if module is here
260 Start to create directories.
261     <-Directories created on the cluster->
262     <TIME> Time to create directories on cluster is >5 s
263 Start to send file(s)
264     <-Files sended->
265     <TIME> Time to create directories on cluster is >130 s
266 Start create PBS file, send and execute
267     <TIME> Time to create pbs file and launch it on cluster is>4 s
268     <-Program Cluster Status->
269     Running program on cluster...
270     >> PBS File on cluster contains
271         #!/bin/bash
272         #PBS -l walltime=00:05:00
273         #PBS -l nodes=1:ppn=1
274         #PBS -q qwork@mp2
275         #PBS -o ~/samtools_1.3.2/samtools_view/stdOutFile
276         #PBS -e ~/samtools_1.3.2/samtools_view/stdErrFile
277         module load bioinformatics/samtools/1.3.2
278         samtools view -b -S  ~/samtools_1.3.2/samtools_view/OutputOf_A.fastq.
    sam  > ~/samtools_1.3.2/samtools_view/outputs/OutputOf_A.fastq.sam.bam
279         a="ExitStatus>"$?"<"
280         echo $a
281     >> Command line running on cluster is
282         samtools view -b -S  ~/samtools_1.3.2/samtools_view/OutputOf_A.fastq.
    sam  > ~/samtools_1.3.2/samtools_view/outputs/OutputOf_A.fastq.sam.bam
283     >> The CLuster task number is 138024.mp2.m
284 Wait until job is done or time is up (after 68 minutes) In seconds, the time
    between two tests is 60,60,60,60,60,60,120,240,480,960,1920 So it means a
    test each minute during the first six minutes, then a test in 2, 4, 8, 16, 32
```

```
     minutes
285        <TIME> Time to wait until tasks done on the cluster is >126 s
286        <-Results downloaded from cluster->
287        <TIME> Time to donwload files from the cluster is >6 s
288
289     <-Program Output->
290     <-End Program Output ->
291     Program Exit Value: 0
292     Parsing outputs...
293
294 ****************************
295
296 Running samtools sort...
297     Initialization...
298     Running [samtools sort]
299     Checking program requirements...
300     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        samtools/sort/samtools_sort:/data/outputs/ -v ./armadillo2-master/results/
        samtools/view/samtools_view:/data/inputs/208/ --name samtools_sort_
        armadilloWF_0 -di jego/samtools
301     Creating inputs...
302     Creating commandline...
303     DockerRunningCommandLine: $ samtools sort   -O bam  /data/inputs/208/OutputOf
        _A.fastq.sam.bam   -o /data/outputs/OutputOf_A.fastq.sam.bam
304     DockerExecution: $ /usr/bin/docker exec -i samtools_sort_armadilloWF_0 sh -c
        './dockerBash.sh'
305
306     Test to access and get modules on cluster already done! Congratulation.
307     Can access to the cluster
308     Start Test if module is here
309     Start to create directories.
310        <-Directories created on the cluster->
311        <TIME> Time to create directories on cluster is >1 s
312     Start to send file(s)
313        <-Files sended->
314        <TIME> Time to create directories on cluster is >51 s
315     Start create PBS file , send and execute
316        <TIME> Time to create pbs file and launch it on cluster is>3 s
317        <-Program Cluster Status->
318        Running program on cluster...
319        >> PBS File on cluster contains
320            #!/bin/bash
```

```
321             #PBS -l walltime=00:05:00
322             #PBS -l nodes=1:ppn=1
323             #PBS -q qwork@mp2
324             #PBS -o ~/samtools_1.3.2/samtools_sort/stdOutFile
325             #PBS -e ~/samtools_1.3.2/samtools_sort/stdErrFile
326             module load bioinformatics/samtools/1.3.2
327             samtools sort -O bam ~/samtools_1.3.2/samtools_sort/OutputOf_A.fastq.
    sam.bam  -o ~/samtools_1.3.2/samtools_sort/outputs/OutputOf_A.fastq.sam.bam
328             a="ExitStatus>"$?"<"
329             echo $a
330         >> Command line running on cluster is
331             samtools sort -O bam ~/samtools_1.3.2/samtools_sort/OutputOf_A.fastq.
    sam.bam  -o ~/samtools_1.3.2/samtools_sort/outputs/OutputOf_A.fastq.sam.bam
332         >> The CLuster task number is 138025.mp2.m
333     Wait until job is done or time is up (after 68 minutes) In seconds, the time
    between two tests is 60,60,60,60,60,60,120,240,480,960,1920 So it means a
    test each minute during the first six minutes, then a test in 2, 4, 8, 16, 32
     minutes
334         <TIME> Time to wait until tasks done on the cluster is >125 s
335         <-Results downloaded from cluster->
336         <TIME> Time to donwload files from the cluster is >6 s
337
338     <-Program Output->
339     <-End Program Output ->
340     Program Exit Value: 0
341     Parsing outputs...
342
343 ****************************
344
345 Running samtools mpileup...
346     Initialization...
347     Running [samtools mpileup]
348     Checking program requirements...
349     No BedFile found.
350     DockerInitCommandLine: $ /usr/bin/docker run  -v ./Armadillo_SNPs_WF_test/
    inputs:/data/inputs/190/ -v ./armadillo2-master/results/samtools/sort/
    samtools_sort:/data/inputs/211/ -v ./armadillo2-master/results/samtools/sort/
    samtools_sort:/data/inputs/202/ -v ./armadillo2-master/results/samtools/
    mpileup/samtools_mpileup:/data/outputs/ --name samtools_mpileup_armadilloWF_2
     -di jego/samtools
351     Creating inputs...
352     Creating commandline...
```

```
353 DockerRunningCommandLine: $ samtools mpileup --BCF --fasta-ref /data/inputs
    /190/genome.fa /data/inputs/202/OutputOf_B.fastq.sam.bam /data/inputs/211/
    OutputOf_A.fastq.sam.bam --output /data/outputs/OutputOf_B.fastq.sam.bam.bcf
354 DockerExecution: $ /usr/bin/docker exec -i samtools_mpileup_armadilloWF_2 sh
    -c './dockerBash.sh'
355
356 Test to access and get modules on cluster already done! Congratulation.
357 Can access to the cluster
358 Start Test if module is here
359 Start to create directories.
360     <-Directories created on the cluster->
361     <TIME> Time to create directories on cluster is >1 s
362 Start to send file(s)
363     <-Files sended->
364     <TIME> Time to create directories on cluster is >97 s
365 Start create PBS file, send and execute
366     <TIME> Time to create pbs file and launch it on cluster is>4 s
367     <-Program Cluster Status->
368     Running program on cluster...
369     >> PBS File on cluster contains
370         #!/bin/bash
371         #PBS -l walltime=00:05:00
372         #PBS -l nodes=1:ppn=1
373         #PBS -q qwork@mp2
374         #PBS -o ~/samtools_1.3.2/samtools_mpileup/stdOutFile
375         #PBS -e ~/samtools_1.3.2/samtools_mpileup/stdErrFile
376         module load  bioinformatics/samtools/1.3.2
377         samtools mpileup --BCF --fasta-ref ~/samtools_1.3.2/samtools_mpileup/
    genome.fa ~/samtools_1.3.2/samtools_mpileup/OutputOf_B.fastq.sam.bam ~/
    samtools_1.3.2/samtools_mpileup/OutputOf_A.fastq.sam.bam --output ~/samtools
    _1.3.2/samtools_mpileup/outputs/OutputOf_B.fastq.sam.bam.bcf
378         a="ExitStatus>"$?"<"
379         echo $a
380     >> Command line running on cluster is
381         samtools mpileup --BCF --fasta-ref ~/samtools_1.3.2/samtools_mpileup/
    genome.fa ~/samtools_1.3.2/samtools_mpileup/OutputOf_B.fastq.sam.bam ~/
    samtools_1.3.2/samtools_mpileup/OutputOf_A.fastq.sam.bam --output ~/samtools
    _1.3.2/samtools_mpileup/outputs/OutputOf_B.fastq.sam.bam.bcf
382     >> The CLuster task number is 138027.mp2.m
383 Wait until job is done or time is up (after 68 minutes) In seconds, the time
    between two tests is 60,60,60,60,60,60,120,240,480,960,1920 So it means a
    test each minute during the first six minutes, then  a test in 2, 4, 8, 16,
```

```
    32 minutes
384         <TIME> Time to wait until tasks done on the cluster is >126 s
385         <-Results downloaded from cluster->
386         <TIME> Time to donwload files from the cluster is >4 s
387
388     <-Program Output->
389     <-End Program Output ->
390     Program Exit Value: 0
391     Parsing outputs...
392
393 ******************************
394
395 Running bcftools call...
396     Initialization...
397     Running [bcftools call]
398     Checking program requirements...
399     DockerInitCommandLine: $ /usr/bin/docker run  -v ./armadillo2-master/results/
        bfctools/call/bcftools_call:/data/outputs/ -v ./armadillo2-master/results/
        samtools/mpileup/samtools_mpileup:/data/inputs/214/ --name bcftools_call_
        armadilloWF_0 -di jego/samtools
400     Creating inputs...
401     Creating commandline...
402     DockerRunningCommandLine: $ bcftools call --variants-only  --output-type v  -
        -multiallelic-caller /data/inputs/214/OutputOf_B.fastq.sam.bam.bcf  --output
        /data/outputs/OutputOf_B.fastq.sam.bam.bcf.vcf
403     DockerExecution: $ /usr/bin/docker exec -i bcftools_call_armadilloWF_0 sh -c
        './dockerBash.sh'
404
405     Test to access and get modules on cluster already done! Congratulation.
406     Can access to the cluster
407     Start Test if module is here
408         The program and it's version has not been found on local. We will check
        online
409         The program and it's version has not been found online.
410         Check the program properties
411     Running will run on the local machine...
412
413     Running program...
414     <-Program Output->
415         Note: Neither --ploidy nor --ploidy-file given, assuming all sites are
        diploid groupadd: group 'truc' already exists
416     <-End Program Output ->
```

```
417     Program Exit Value: 0
418     Parsing outputs...
419
420 ****************************
421
422 ********************************************************************************
423 Computation finished in 39m 53s Ended at 2017-09-04 11:53:01
424 ********************************************************************************
```

B.4 Comparatif des fichiers résultats entre SnakeMake et Docker

```
$ diff armadillo2-master/results/bcftools_call/OutputOf_A.fastq.sam.bam.bcf.vcf
    SnakeMakeTuto/calls/all.vcf

3,5c3,5
< ##samtoolsVersion=1.3.1+ htslib - 1.3.2
< ##samtoolsCommand=samtools mpileup - -BCF - -fasta-ref / data/inputs/1/genome.fa -
    -output / data/outputs/OutputOf_A.fastq.sam.bam.bcf / data/inputs/2/OutputOf_A.
    fastq.sam.bam / data/inputs/4/OutputOf_B.fastq.sam.bam
< ##reference=file:/// data/inputs/1/genome.fa
---
> ##samtoolsVersion=1.3.1+ htslib - 1.3.1
> ##samtoolsCommand=samtools mpileup - g - f data /genome.fa - -output calls / all.bcf
    sorted_reads/A.bam sorted_reads /B.bam
> ##reference=file:// data/genome.fa

27,29c27,29
< ##bcftools_callVersion=1.3.1+ htslib - 1.3.2
< ##bcftools_callCommand=call - -variants-only - -output-type v - -multiallelic -
    caller - -output / data/outputs/OutputOf_A.fastq.sam.bam.bcf.vcf / data/inputs
    /1/OutputOf_A.fastq.sam.bam.bcf
< #CHROM POS ID REF ALT QUAL FILTER INFO FORMA
    inputs/2/OutputOf_A.fastq.sam.bam / data/inputs/4/OutputOf_B.fastq.sam.bam
---
> ##bcftools_callVersion=1.3.1+ htslib - 1.3.1
> ##bcftools_callCommand=call - mv - -output calls /all.vcf calls /all.bcf
> #CHROM POS ID REF ALT QUAL FILTER INFO FORMA
    sorted_reads/A.bam sorted_reads /B.bam
```

BIBLIOGRAPHIE

Afgan, E., Baker, D., Coraor, N., Chapman, B., Nekrutenko, A. et Taylor, J. (2010). Galaxy cloudman : delivering cloud compute clusters. *BMC Bioinformatics*, *11*(12), S4. http://dx.doi.org/10.1186/1471-2105-11-S12-S4

Agharbaoui, Z., Leclercq, M., Remita, M., Badawi, M., Lord, E., Houde, M., Danyluk, J., Diallo, A. B. et Sarhan, F. (2015). An integrative approach to identify hexaploid wheat mirnaome associated with development and tolerance to abiotic stress. *BMC Genomics*, *16*(1), 339. http://dx.doi.org/10.1186/s12864-015-1490-8

Anderson, C. (2015). Docker [software engineering]. *IEEE Software*, *32*(3), 102–c3. http://dx.doi.org/10.1109/MS.2015.62

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. et Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, *53*(4), 50–58. http://dx.doi.org/10.1145/1721654.1721672

Bahassi, E. M. et Stambrook, P. (2014). Next-generation sequencing technologies : breaking the sound barrier of human genetics. *Mutagenesis*, *29*(5), 303–310. http://dx.doi.org/10.1093/mutage/geu031

Belmann, P., Dröge, J., Bremges, A., McHardy, A. C., Sczyrba, A. et Barton, M. D. (2015). Bioboxes : standardised containers for interchangeable bioinformatics software. *GigaScience*, *4*(1), 1–4.

Bernstein, D. (2014). Containers and cloud : From lxc to docker to kubernetes. *IEEE Cloud Computing*, *1*(3), 81–84. http://dx.doi.org/10.1109/MCC.2014.51

Beserra, D., Moreno, E. D., Endo, P. T., Barreto, J., Sadok, D. et Fernandes, S. (2015). Performance analysis of lxc for hpc environments. Dans *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, 358–363. http://dx.doi.org/10.1109/CISIS.2015.53

Biederman, E. W. et Networx, L. (2006). Multiple instances of the global linux namespaces. Dans *Proceedings of the Linux Symposium*, volume 1, 101–112.

Blankenberg, D., Kuster, G. V., Bouvier, E., Baker, D., Afgan, E., Stoler, N., Taylor, J. et Nekrutenko, A. (2014). Dissemination of scientific software with galaxy toolshed. *Genome Biology*, *15*(2), 1–3. http://dx.doi.org/10.1186/gb4161

Boettiger, C. (2015). An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79.

Bui, T. (2015). Analysis of docker security. *arXiv preprint arXiv :1501.02967*.

Chen, J., Cohen-Boulakia, S., Froidevaux, C., Goble, C., Missier, P. et Williams, A. (2014). DistillFlow : removing redundancy in scientific workflows. Dans *SSDBM '14 Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, Aalborg, Denmark. http://dx.doi.org/10.1145/2618243.2618287

Collberg, C. et Proebsting, T. A. (2016). Repeatability in computer systems research. *Commun. ACM*, *59*(3), 62–69. http://dx.doi.org/10.1145/2812803

Coulouris, G., Dollimore, J., Kindberg, T. et Blair, G. (2011). *Distributed Systems : Concepts and Design (5th Edition)* (5 éd.). Addison Wesley.

Docker Company (2016). https ://www.docker.com/ (Visited : 2016-03-31).

Docker Glossary (2017). https ://docs.docker.com/engine/reference/glossary/ (Visited : 2017-01-28).

Felter, W., Ferreira, A., Rajamony, R. et Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. Dans *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 171–172. http://dx.doi.org/10.1109/ISPASS.2015.7095802

Field, D., Tiwari, B., Tim, T. B., Houten, S., Swan, D., Bertrand, N. et Thurston, M. (2006). Open software for biologists : from famine to feast. *Nat Biotech*, *24*(7), 801–803. http://dx.doi.org/10.1038/nbt0706-801

Folarin, A., Dobson, R. J. et Newhouse, S. J. (2015). Ngseasy : a next generation sequencing pipeline in docker containers. *F1000Research*, *4*(4 :997). http://dx.doi.org/doi:10.12688/f1000research.7104.1

Fusaro, V. A., Patil, P., Gafni, E., Wall, D. P. et Tonellato, P. J. (2011). Biomedical cloud computing with amazon web services. *PLOS Computational Biology*, *7*(8), 1–6. http://dx.doi.org/10.1371/journal.pcbi.1002147

Galaxy-Docker (2016). https ://wiki.galaxyproject.org/Admin/Tools/Docker (Visited : 2016-03-31).

Gentzsch, W. (2014). Linux containers simplify engineering and scientific simulations in the cloud. Dans *2014 Annual Global Online Conference on Information and Computer Technology*, 22–26. http://dx.doi.org/10.1109/GOCICT.2014.11

Gerlach, W., Tang, W., Wilke, A., Olson, D. et Meyer, F. (2015). Container orchestration for scientific workflows. Dans *2015 IEEE International Conference on Cloud Engineering*, 377–378. http://dx.doi.org/10.1109/IC2E.2015.87

Goecks, J., Nekrutenko, A. et Taylor, J. (2010). Galaxy : a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, *11*(8), R86. http://dx.doi.org/10.1186/gb-2010-11-8-r86

Guedes, E. A. C., Silva, L. E. T. et Maciel, P. R. M. (2014). Performability analysis of i/o bound application on container-based server virtualization cluster. Dans *2014 IEEE Symposium on Computers and Communications (ISCC)*, 1–7. http://dx.doi.org/10.1109/ISCC.2014.6912556

Hale, J. S., Li, L., Richardson, C. N. et Wells, G. N. (2016). Containers for portable, productive and performant scientific computing. *CoRR*, *abs/1608.07573v2*. http://dx.doi.org/10.1109/MCSE.2017.2421459

Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M. R., Li, P. et Oinn, T. (2006). Taverna : a tool for building and running workflows of services. *Nucleic acids research*, *34*(suppl 2), W729–W732.

Hung, L.-H., Meiss, T., Keswani, J., Xiong, Y., Sobie, E. et Yeung, K. (2017). Building containerized workflows for rna-seq data using the biodepot-workflow-builder (bwb). *bioRxiv*. http://dx.doi.org/10.1101/099010

Joy, A. M. (2015). Performance comparison between linux containers and virtual machines. Dans *2015 International Conference on Advances in Computer Engineering and Applications*, 342–346. http://dx.doi.org/10.1109/ICACEA.2015.7164727

Kivity, A., Kamay, Y., Laor, D., Lublin, U. et Liguori, A. (2007). Kvm : the linux virtual machine monitor. Dans *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*.

Kurtzer, G. M., Sochat, V. et Bauer, M. W. (2017). Singularity : Scientific containers for mobility of compute. *PLOS ONE*, *12*(5), 1–20. http://dx.doi.org/10.1371/journal.pone.0177459

Köster, J. et Rahmann, S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, *28*(19), 2520–2522. http://dx.doi.org/10.1093/bioinformatics/bts480

LCD (2017). https ://linuxcontainers.org/lxd/ (Visited : 2017-01-30).

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G. et Durbin, R. (2009). The sequence alignment/map format and samtools. *Bioinformatics*, *25*(16), 2078. http://dx.doi.org/10.1093/bioinformatics/btp352

Lin, C., Lu, S., Fei, X., Chebotko, A., Pai, D., Lai, Z., Fotouhi, F. et Hua, J. (2009). A reference architecture for scientific workflow management systems and the view soa solution. *IEEE Transactions on Services Computing*, *2*(1), 79–92. http://dx.doi.org/10.1109/TSC.2009.4

Linux Kernel Organization (2017). https ://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (Visited : 2017-01-28).

Lipman, D. J. et Pearson, W. R. (1985). Rapid and sensitive protein similarity searches. *Science*, *227*, 1435–41. http://dx.doi.org/10.1126/science.2983426

Liu, B., Madduri, R. K., Sotomayor, B., Chard, K., Lacinski, L., Dave, U. J., Li, J., Liu, C. et Foster, I. T. (2014). Cloud-based bioinformatics workflow platform for large-scale next-generation sequencing analyses. *Journal of Biomedical Informatics*, *49*, 119 – 133. http://dx.doi.org/10.1016/j.jbi.2014.01.005

Liu, L., Li, Y., Li, S., Hu, N., He, Y., Pong, R., Lin, D., Lu, L. et Law, M. (2012). Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology, 2012*, 11. http://dx.doi.org/10.1155/2012/251364

Lord, E., Leclercq, M., Boc, A., Diallo, A. B. et Makarenkov, V. (2012). Armadillo 1.1 : An original workflow platform for designing and conducting phylogenetic analysis and simulations. *PLoS ONE*, *7*(1), 1–9. http://dx.doi.org/10.1371/journal.pone.0029903

Ludvigh, R., Rebok, T., Tunka, V. et Nguyen, F. (2015). Ruby benchmark tool using docker. Dans *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 947–952. http://dx.doi.org/10.15439/2015F99

Ludäscher, B., Altintas, I., Bowers, S., Cummings, J., Critchlow, T., Deelman, E., Roure, D., Freire, J., Goble, C., Jones, M. *et al.* (2009). Scientific process automation and workflow management. *Scientific Data Management : Challenges, Existing Technology, and Deployment, Computational Science Series*, *230*, 476–508.

LXC (2017). https ://github.com/lxc/lxc (Visited : 2017-01-28).

Mell, P. M. et Grance, T. (2011). *The NIST Definition of Cloud Computing.* Rapport technique, Gaithersburg, MD, United States

Merkel, D. (2014). Docker : Lightweight linux containers for consistent development and deployment. *Linux J.*, *2014*(239).

Migliorini, S., Gambini, M., Rosa, M. L. et ter Hofstede, A. H. (2011). Pattern-based evaluation of scientific workflow management systems

Monat, C., Tranchant-Dubreuil, C., Kougbeadjo, A., Farcy, C., Ortega-Abboud, E., Amanzougarene, S., Ravel, S., Agbessi, M., Orjuela-Bouniol, J., Summo, M.

et Sabot, F. (2015). Toggle : toolbox for generic ngs analyses. *BMC Bioinformatics*, *16*(1), 374. http://dx.doi.org/10.1186/s12859-015-0795-6

Namespaces (2017). http ://man7.org/linux/man-pages/man7/namespaces.7.html (Visited : 2017-01-28).

Nextflow (2017). http ://nextflow.io (Visited : 2017-01-28).

Okonechnikov, K., Golosova, O. et Fursov, M. (2012). Unipro ugene : a unified bioinformatics toolkit. *Bioinformatics*, *28*(8), 1166. http://dx.doi.org/10.1093/bioinformatics/bts091

Oracle VM VirtualBox (2017). https ://www.virtualbox.org/ (Visited : 2017-01-28).

P. Ewels, F. Krueger, M. K. et Andrews, S. (2017). Cluster flow : A user-friendly bioinformatics workflow tool. *F1000Research*, *5*. http://dx.doi.org/10.12688/f1000research.10335.2

Popek, G. J. et Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Commun. ACM*, *17*(7), 412–421. http://dx.doi.org/10.1145/361011.361073

Scott, R. J. et Irvine, C. E. (2000). Analysis of the intel pentium's ability to support a secure virtual machine monitor. Dans *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, SSYM'00, 10–10., Berkeley, CA, USA. USENIX Association.

Soltesz, S., Pötzl, H., Fiuczynski, M., Bavier, A. et Peterson, L. (2007). Container-based operating system virtualization : A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, *41*(3), 275–287. http://dx.doi.org/10.1145/1272998.1273025

van Dijk, E. L., Auger, H., Jaszczyszyn, Y. et Thermes, C. (2014). Ten years of next-generation sequencing technology. *Trends in Genetics*, *30*(9), 418 – 426. http://dx.doi.org/10.1016/j.tig.2014.07.001

VMware (2017). https ://www.vmware.com (Visited : 2017-02-31).

Woollard, D., Medvidovic, N., Gil, Y. et Mattmann, C. A. (2008). Scientific software as workflows : From discovery to distribution. *IEEE Software*, *25*(4), 37–43. http://dx.doi.org/10.1109/MS.2008.92

Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T. et Rose, C. A. F. D. (2013). Performance evaluation of container-based virtualization for high performance computing environments. Dans *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 233–240. http://dx.doi.org/10.1109/PDP.2013.41

Yildiz, A., Dilaveroglu, E., Visne, I., Günay, B., Sefer, E., Weinhausel, A., Rattay, F., Goble, C. A., Pandey, R. V. et Kriegner, A. (2014). Bifi : a taverna plugin for a simplified and user-friendly workflow platform. *BMC Research Notes*, *7*(1), 1–9. http://dx.doi.org/10.1186/1756-0500-7-740

Zhang, Q., Cheng, L. et Boutaba, R. (2010). Cloud computing : state-of-the-art and research challenges. *Journal of Internet Services and Applications*, *1*(1), 7–18. http://dx.doi.org/10.1007/s13174-010-0007-6