ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE AVEC MÉMOIRE EN GÉNIE
DES TECHNOLOGIES DE L'INFORMATION
M. Sc. A.

PAR
Ghassen ZAGDENE

MISE EN PLACE D'UN CLUSTER DE SYSTÈMES ARM POUR DES SOLUTIONS DE SÉCURITÉ

MONTRÉAL, LE 27 AVRIL 2016

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Chamseddine Talhi, directeur de mémoire
Département de génie logiciel et TI à l'École de technologie supérieure

M. Abdelouaheb Gherbi, président du jury
Département de génie logiciel et TI à l'École de technologie supérieure

M. Carlos Vázquez, membre du jury
Département de génie logiciel et TI à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 10-05-2016

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Tout d'abord, je remercie mon directeur de recherche, M. Chamseddine Talhi pour l'encadrement de qualité et le soutien qu'il ma offerts, ainsi que pour sa disponibilité et ses conseils qui m'ont été précieux tout le long de ma maîtrise.

Je tiens à remercier ma fiancée, Mariem Jelassi qui m'a apporté son soutien et m'a motivé surtout dans les moments difficiles et à qui je dois en partie ma réussite.

Je remercie aussi mon frère, Oussama Boudar qui a toujours été présent et qui m'a apporté son appui ainsi que mes collègues et amis du laboratoire LASI :Hanine Tout, Maroua Ben Attia, Manel Abdeltif, Mahdi Elarbi et à leur tête ILyes Lahmar.

Je tiens aussi à exprimer ma profonde gratitude pour ma famille qui m'a consacré tous les moyens et supports nécessaires au bon déroulement et à la réussite de mes études, particulièrement mes parents Hamadi et Najla, ainsi que mes grands-parents Hassan et Assia et sans oublier mon petit frère Khalil.

Par la même, je remercie également les membres du jury d'avoir accepté d'évaluer mon mémoire.

Enfin, je tiens également à remercier l'ensemble du corps enseignant de l'École de technologie supérieure pour toutes les connaissances et pratiques m'ayant été transmises de leur part lors de ma maîtrise.

MISE EN PLACE D'UN CLUSTER DE SYSTÈMES ARM POUR DES SOLUTIONS DE SÉCURITÉ

Ghassen ZAGDENE

RÉSUMÉ

Le monde de l'informatique de haute performance (HPC) a évolué ces dernières années surtout après l'apparition des clusters Beowulf qui consistent à utiliser les PC conventionnels pour les interconnecter et récolter leur puissance de calcul. La réduction du coût du matériel nécessaire a rendu l'informatique de haute performance accessible à un plus large public, et certaines entreprises mettent en place leurs propres cluster et centres de données en se basant sur les clustersBeowulf. Cependant, même avec de l'équipement conventionnel, ces clusters et centres de données restent relativement chers, occupent de l'espace et consomment beaucoup d'énergie.

L'approche proposée par la présente étude vise àréduire les coûts de la mise en place d'un cluster destiné au HPC en utilisant des ordinateurs mono-cartes à architecture ARM. La particularité de cette dernière est sa faible consommation d'énergie par rapport aux autres architectures. Ainsi, nous mettons en place un cluster ARM sur lequel nous exécutons des algorithmes liés à la sécurité. De plus, puisque combiner les clusters et la virtualisation offre des avantages non négligeables, nous déployons docker sur certaines cartes ARM, qui est une nouvelle solution de virtualisation afin de mesurer l'impact sur les performances du système.

Les résultats montrent que les appareils ARM délivrent de bons résultats à condition de bien choisir l'environnement de développement et de bien équilibrer la charge au sein du cluster. Docker n'a pas d'impact significatif sur les performances du système, mais il n'est pas encore adapté à l'architecture ARM.

Mots clés :architecture ARM; sécurité; cluster beowulf; HPC; virtualisation; docker

SETTING UP AN ARM-BASED CLUSTER FOR SECURITY SOLUTIONS

Ghassen ZAGDENE

ABSTRACT

The world of high performance computing (HPC) evolved these past years, especially after the appearance of the beowulf clusters which consist of commodity equipment interconnected to harvest their computing power. This cost reduction, made HPC accessible to a large public and some companies built their own cluster and datacenters based on the Beowulf concept. However, the costs are still relatively high and the clusters still consume a lot of energy.

In this study, we aim to provide a low-cost, low-energy cluster based on ARM devices. This architecture (ARM) is known to be more energy-efficient than any other architecture. Also, we implement security-related algorithms and execute them on our cluster to test its efficiency. Since combining clustering and virtualization has proved to provide notable advantages, we deploy Docker, a new lightweight virtualization solution, on some ARM boards to test its impact on system performance.

The results have shown that the ARM based cluster can deliver good performance, given the right message passing environment and load-balancing strategy. As for virtualization support, Docker had little impact on the board's performance but is not yet adapted for the ARM architecture.

Keywords: ARM architecture; beowulf cluster; security; HPC; virtualization; Docker

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ARM	Advanced Risc Machines
CPU	Central Processing Unit (processeur)
GPU	Graphic Processing Unit
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
IPC	Inter-Process Communication
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
MPI	Message Passing Interface
OS	Operating System (système d'exploitation)
PAES	Parallel Advanced Encryption Standard
PID	Process Identifier
PVM	Parallel Virtual Machine
RAM	Random Access Memory

LISTE DES SYMBOLES ET UNITÉS DE MESURE

O	octet
Mb	Megabyte, Mo (Mégaoctet)
Gb	Gigabyte, Go (Gigaoctet)
Kb	Kilobyte, Ko (Kilooctet)
MHZ	Mégahertz, unité de mesure de fréquence du Système international (SI)
Ms	Milliseconde
S	Seconde

INTRODUCTION

Mise en contexte

Ces dernières années, le monde de l'informatique connait un besoin grandissant en terme de vitesse de calcul, de traitement, ainsi que de stockage de données. L'exécution de tâches nécessitant une large puissance de calcul appartient au monde de l'informatique de haute performance et a été longtemps réservée aux entreprises et organisations possédant des moyens financiers considérables.

Le calcul de haute performance bénéficie de plus en plus à un nombre grandissant de domaines autres que l'informatique elle-même. Ces domaines incluent l'imagerie, la médecine, la mécanique et d'autres domaines de l'ingénierie qui ont un besoin de simuler de larges problèmes. À ce jour, les outils les plus performants pour le domaine de l'informatique de haute performance sont sans doute les supercalculateurs qui sont composés de milliers de processeurs travaillant en parallèle à la résolution de problèmes liés par exemple aux origines de l'univers et au développement de remèdes pour le cancer. Mais, ces supercalculateurs ont un coût élevé qui avoisine les 20M$ (insidehpc, 2016b)et ne sont pas donc accessibles à n'importe qui. De plus, ces supercalculateurs consomment énormément d'énergie.

L'un des secteurs qui peut bénéficier de la puissance offerte par le HPC, est celui de la sécurité. En effet, de nombreuses applications liées à la sécurité, telles que le chiffrement ou la détection de malwares[1] demandent une puissance de calcul qui n'est pas toujours disponible sur les systèmes où s'exécutent ces algorithmes. En effet, dans le cas des téléphones mobiles ou autres systèmes embarqués, les ressources sont limitées et effectuer des analyses de détection de malwares ou exécuter des algorithmes de chiffrement affectent grandement les performances du système d'où le besoin de les "décharger" et les exécuter dans un environnement moins contraignant.

---

[1] Logiciel malveillant

D'un autre côté, la technologie de la virtualisation apporte de nombreux avantages dont le déploiement rapide et à distance d'environnements de développements. Cette dernière est l'une des fondations sur lesquels reposent les centres de données grâce à la facilité de gestion des ressources et l'isolation des environnements des différents serveurs qu'elle offre.

Combiner la technologie des clusters avec la virtualisation serait grandement bénéfique du point de vue de l'exploitation de toutes les ressources offertes par un cluster. De plus, plusieurs types d'applications pourraient s'exécuter sur un cluster, chacune demandant un environnement différent. La virtualisation permettrait à ces applications de tourner sur un même nœud physique, sans avoir à dédier un nœud à chaque application ce qui augmente la flexibilité du cluster. Ce dernier peut aussi bénéficier de la haute disponibilité et de la tolérance aux fautes qu'offre la virtualisation, puisque l'échec d'une machine virtuelle ne signifie pas l'échec de tout le système.

Un autre avantage de la combinaison des clusters et de la virtualisation est la sécurité. En effet, l'isolation offerte par la virtualisation permettrait à plusieurs applications de s'exécuter sur le même cluster physique, en parallèle tout en étant complètement isolées les unes des autres.

Pour les deux technologies citées ci-dessus, il existe des objectifs en commun,dont le besoin de réduire le coût du matériel ainsi que la consommation d'énergie. En effet, selon(Hamilton, 2009), plus de 35% du coût total des centres de données est lié à l'énergie électrique consommée, aux opérations de refroidissement ainsi qu'aux installations physiques. Réduire ces coûts est alors une nécessité.

Dans cette optique, l'intérêt pour la technologie ARM est de plus en plus grandissant en vue des avantages qu'offrent les appareils présentant cette architecture. En effet, cette architecture équipe la plupart des téléphones portables sur le marché et d'autres systèmes embarqués. Dernièrement elle vient de dépasser l'architecture x86 dans la part de marché et les projections prévoient que cette tendance va s'accroitre comme le montre la figure suivante.

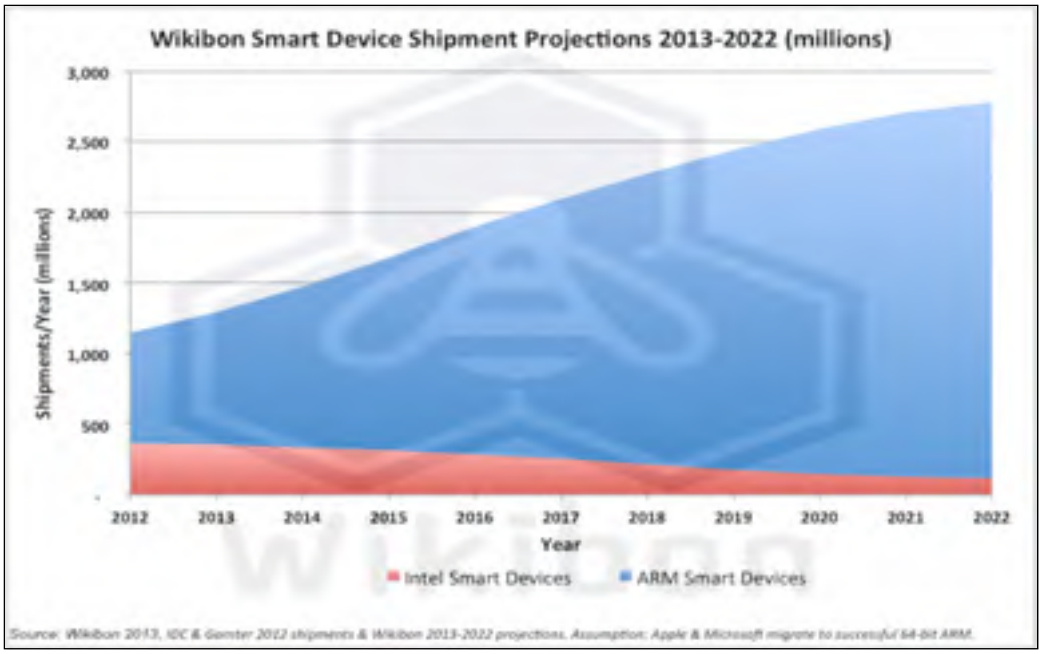

Figure 0.1 Projection de la part de marché de l'architecture ARM
en comparaison avec Intel Tirée de (Floyer, 2013).

Parmi les raisons de cette popularité est la faible consommation d'énergie des appareils ARM et le ratio performance/coût qui est très réduit. En plus des téléphones intelligents de nouvelle génération, l'architecture ARM offre des cartes de développement avec des caractéristiques intéressantes. En effet, ces derniers sont l'équivalent d'une carte mère comportant toutes les composantes nécessaires à l'exécution d'un système d'exploitation puisqu'elles possèdent des processeurs multicoeurs, une mémoire RAM décente ainsi que d'autres périphériques.

Est-il alors possible de tirer profit des avantages qu'offre l'architecture ARM et les appliquer dans le domaine de l'informatique de haute performance dans le but d'exécuter des algorithmes de sécurité? Est-ce qu'en combinant plusieurs de ces cartes, on arriverait à avoir des résultats satisfaisants en comparaison avec les systèmes conventionnels? D'autre part, est-il possible de virtualiser les appareils ARM et garder d'assez bonne performance afin de faciliter la gestion et la sécurisation des cluster et considérer l'usage de ces derniers dans le cadre d'un centre de données?

C'est dans ce cadre que s'inscrit la présente étude, où nous testons dans un premier temps l'efficacité de ces cartes dans l'exécution d'algorithmes de sécurité (chiffrement et détection de malwares) de manière individuelle puis en formant un cluster avec plusieurs cartes. Dans un second temps, nous déployons une technique de virtualisation légère sur les cartes et les testons pour mesurer l'impact sur les performances du système.

Problématique

Notre problématique de recherche se divise en deux grandes parties :

- Est-il possible d'utiliser les ordinateurs mono-cartes possédant l'architecture ARM pour exécuter des algorithmes de cryptographie et de détection de logiciels malveillants?
- Est-il possible d'adapter la technologie de virtualisation sur les appareils ARM sans que cela n'impacte leur fonctionnement?

La première partie de la problématique implique des sous-problématiques que nous tentons de résoudre dans ce travail:

- Quelles sont les difficultés techniques liées à l'utilisation des appareils ARM?
- Quels sont les outils les mieux adaptés aux appareils ARM pour implémenter une architecture HPC?
- Est-il possible d'améliorer la performance en équilibrant la charge entre les appareils ARM?

Afin de répondre à ces questions, nous mettons en place dans un premier temps un cluster constitué de plusieurs ordinateurs mono-cartes possédant l'architecture ARM. Nous implémentons pour ce dernier un système de monitorage adéquat pour les systèmes à ressources limitées. Afin de déterminer l'outil le mieux adapté pour le passage de message au sein de ce cluster, nous adaptons différents algorithmes aux deux principaux outils sur le marché à savoir MPI et PVM, et testons les performances de ces deux derniers. Nous implémentons deux algorithmes de chiffrement sur le cluster qui sont le AES et le AES

parallèle ainsi qu'un algorithme de correspondance des patrons malicieux. Nous proposons aussi une solution d'équilibrage de la charge et testons son efficacité.

Pour ce qui est de la virtualisation, la plupart des cartes ARM n'étant pas adaptées à toutes les technologies de virtualisation, nous choisissons la solution la plus prometteuse sur le marché actuel et la testons sur deux types de cartes qui permettent de l'exécuter. Par la suite, nous effectuons des tests afin de déterminer l'impact de la virtualisation sur le fonctionnement du système.

Objectifs

Les objectifs que nous nous sommes donc fixés sont les suivants:

- Installer les environnements adéquats pour le fonctionnement des cartes ARM.
- Implémenter les algorithmes de sécurité sur les cartes.
- Mettre en place un cluster composé des cartes ARM en utilisant des mécanismes de passage de message.
- Améliorer le cluster en ajoutant un module de monitorage et un mécanisme d'équilibrage de la charge.
- Déployer la virtualisation sur les cartes ARM.

Méthodologie suivie

La méthodologie suivie est décrite par la figure 1-2. Cette dernière consiste à définir la problématique de recherche, ensuite nous avons recherché les différents exemples dans la littérature liés à notre étude. Les documents et articles recherchés concernent l'usage des cluster Beowulf ainsi que des clusters constitués par les appareils ARM. Nous investiguons dans la littérature le meilleur mécanisme de passage de message au sein d'un cluster et nous concluons qu'il n’y a pas de consensus quant à ce sujet. Nous recherchons les travaux qui s'intéressent aux traitements de sécurité sur des systèmes limités et qui proposent de les décharger vers des environnements disposant de plus de ressources. Nous consultons aussi les techniques d'équilibrage de la charge présents dans la littérature.

Afin de choisir la méthode de virtualisation à déployer sur les cartes ARM, nous recherchons les documents et articles liés aux différentes technologies de virtualisation et les différences entre ces dernières.

Nous avons par la suite mis en place un cluster implémentant trois algorithmes liés à la sécurité en utilisant les deux approches de passages de messages MPI et PVM. Afin que le cluster soit complet, nous le dotons d'un module de monitorage qui ne consomme beaucoup de ressources afin d'être adapté aux appareils ARM. Nous implémentons aussi un mécanisme d'équilibrage pour la distribution de la charge au sein du cluster. Pour ce qui est de la virtualisation, nous effectuons la configuration nécessaire afin de déployer Docker sur les cartes.

Finalement, nous testons les résultats de l'implémentation pour le cluster en comparant le temps d'exécution de chaque algorithme selon différentes configurations et selon les deux implémentations MPI et PVM.

Pour les tests de la virtualisation, nous exécutons plusieurs benchmark sur les cartes ARM afin de déterminer l'impact de la virtualisation sur l'accès à la RAM, à la CPU, au disque ainsi qu'aux interfaces réseau.

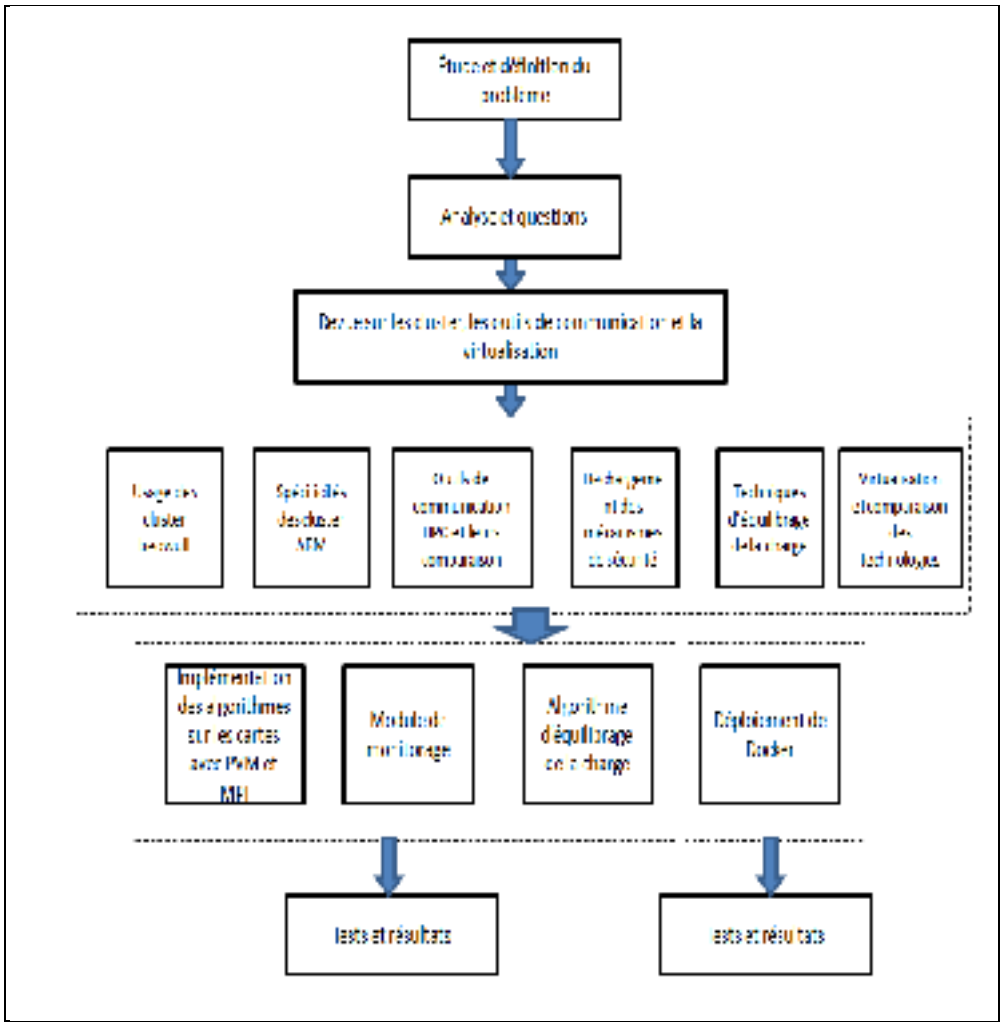

Figure 0.2 Méthodologie de notre étude

Organisation du rapport

Le rapport est organisé comme suit : dans la première partie, nous présentons les notions de base nécessaires à la compréhension des concepts principaux de notre étude. Ceci inclut la notion de cluster, de HPC, la technologie ARM et les technologies de virtualisation.

Le second chapitre comporte la revue de littérature des travaux liés à notre étude. Nous présentons les recherches ayant été effectuées sur les cluster Beowulf et notamment ceux réalisés avec des appareils ARM. Par la suite, nous présentons les comparaisons faites dans la

littérature entre les deux plateformes de développement pour les clusters qui sont PVM et MPI. De plus, nous détaillons le concept de distribution de la charge au sein d'un cluster hétérogène. Nous présentons par la suite les travaux qui ont proposé de décharger les traitements liés à la sécurité vers des environnements disposant d'assez de ressources. Et enfin, nous citons les travaux qui ont été faits sur la comparaison des technologies de virtualisation.

Le troisième chapitre détaille l'architecture du cluster que nous avons mis au point ainsi que les algorithmes de sécurité que nous avons implémentés dessus, à savoir la correspondance des patrons et l'AES séquentiel et parallèle. Aussi, nous présentons l'architecture du mécanisme de monitorage et l'algorithme de distribution de la charge.

Le quatrième chapitre est celui des résultats où nous présentons les résultats des différentes expérimentations pour déterminer l'efficacité du cluster ARM à travers les différents algorithmes implémentés, et ce avec PVM et MPI. Nous faisons des benchmarks pour déterminer l'impact de la virtualisation avec Docker sur les performances des cartes ARM.

Finalement, nous donnons une conclusion quant aux résultats obtenus et les contributions apportées par ce travail.

CHAPITRE 1

NOTIONS DE BASE

1.1 Introduction

Dans ce chapitre, nous allons présenter les notions de base essentielles à la compréhension de ce travail. Ces notions consistent en l'informatique hautement parallèle qui est en constante évolution avec les progrès technologiques et dont l'accessibilité au grand public en a fait un centre d'intérêt de plusieurs travaux de recherche. Les points les plus importants du HPC sont la notion de cluster et les mécanismes de communication au sein d'un environnement hautement parallèle.

La deuxième notion importante est l'architecture ARM, et plus précisément les appareils ARM et leur utilisation dans le domaine des clusters. Nous allons présenter entre autres les différents types de processeurs et leur évolution et aborder les environnements que peuvent offrir ces cartes.

La dernière partie de ce chapitre concerne la virtualisation. Nous présentons la définition de ce concept ainsi que les différentes techniques qui existent en matière de virtualisation, notamment Docker[2].

1.2 Informatique de haute performance

1.2.1 Généralités

Le calcul à haute performance consiste à rassembler la puissance de calcul de plusieurs processeurs de manière à surpasser la puissance de calcul que pourrait offrir un seul ordinateur conventionnel afin de résoudre des problèmes plus ou moins importants et exigeants en ingénierie, mathématiques...

---

[2] *https://www.docker.com/*

Durant les dernières années, l'importance des HPC dans le domaine de la mécanique (calcul de la mécanique des fluides), pharmaceutique (conception de médicaments), traitement d'image médical et d'autres domaines pouvant bénéficier d'une grande puissance de calcul a été cruciale (warwick, 2015).

Le rassemblement de la puissance de plusieurs processeurs se fait en général de manière à ce que ces derniers travaillent en collaboration ou en 'parallèle', on parle alors de programmation parallèle. Cette dernière prévoit deux paradigmes qui sont :

- Instruction unique plusieurs données : Comme son nom l'indique, une seule instruction est appliquée à différentes données afin de produire plusieurs résultats.
- Instructions multiples, données multiples :Différentes instructions sont appliquées à différentes données.

1.2.2 Gestion de la mémoire

Il existe trois architectures principales selon lesquelles la mémoire est gérée dans un environnement HPC (insidehpc, 2016a).

- Mémoire partagée : On parle de mémoire partagée lorsque les cpu travaillant en parallèle se partagent le même espace d'adressage physique. Ce type de mémoire dispose de deux manières d’accès Uniform Memory Access et On cache-coherent nonUniform Memory Access.
- Mémoire distribuée : L'espace d'adressage est séparé (physiquement et logiquement), chaque processeur possède son propre espace adressage et les processus peuvent communiquer entre eux en utilisant la communication réseau, plus précisément des mécanismes de passage de message qu'on abordera dans la prochaine partie.
- Système hybride : On parle de système hybride lorsque les deux types de mémoires (partagée et distribuée) sont utilisés. Cela consiste en plusieurs blocs (processeurs multicoeurs) partageant une mémoire et qui sont interconnectés en réseau. Cette topologie est la plus utilisée en pratique.

1.2.3 Cluster Beowulf

À ses débuts, le monde du HPC consistait en d'imposants supercalculateurs utilisés principalement dans les départements de défense et certains laboratoires de recherche. Ces appareils étaient non seulement très couteux, mais aussi difficiles à programmer et à entretenir ce qui nécessitait des experts propres à l'architecture de chaque supercalculateur. Tous ces facteurs faisaient en sorte que le calcul hautement parallèle était inaccessible aux scientifiques et autres personnes ne disposant pas d'importantes ressources financières(Senning, 2016).

Cette limitation a changé avec l'apparition d'une nouvelle approche et l'émergence de supercalculateurs formés par des clusters de machines conventionnelles. Cette architecture a été appelée Beowulf et consiste en un ensemble de PC connectés en réseaux et disposant de certaines librairies (MPI ou PVM) nécessaires à la distribution des tâches et la communication. Le premier prototype, qui a été réalisé à la NASA en 1994 et était composé de 16 processeurs DX4 connectés via Ethernet, a été un tel succès que les universitaires l'ont adopté(Merkey, 2015).

Figure 1.1 Exemple de cluster Beowulf.
Tirée de (Senning, 2016).

Certains facteurs ont contribué à la popularité que connaît le cluster Beowulf aujourd'hui, parmi lesquels le grand nombre de constructeurs de PC et d'autres composantes destinées au grand public qui a induit une forte concurrence faisant baisser les prix considérablement. D'autre part, le monde des logiciels libres a connu une grande évolution, notamment Linux et les librairies MPI et PVM dont dépendent les clusters Beowulf. Aussi, les technologies réseau n'ont cessé de se développer depuis le premier prototype de cluster en 1994 où à l'époque les processeurs étaient trop rapides pour une seule connexion Ethernet et les switchs Ethernet n'étaient pas à la portée de tout le monde.

Ce qui constitue l'un des points forts des clusters Beowulf est le fait que le modèle de programmation ne dépend pas des composantes matérielles telles que les processeurs ou la vitesse du réseau, ce qui lui procure une grande flexibilité. Cela est dû en grande partie aux protocoles de communication qu'on verra dans le paragraphe suivant.

1.2.4 Mécanismes de passage des messages

L'un des concepts les plus importants dans le HPC est le mécanisme de passage des messages entre les différents nœuds. En effet, tout le système est dépendant de la bonne communication des données et des instructions, une mauvaise communication nuirait gravement aux performances et pourrait rendre le tout obsolète. La clé d'une bonne communication réside dans l'outil de passage choisi ainsi que la configuration des nœuds. Au sein d'un programme de passage de messages, chaque processeur exécute une tâche séparément avec des variables qui lui sont privées et donc locales. La communication se fait à travers des fonctions propres au système de passage de messages qui peut varier d'une simple variable à des structures plus complexes que nous verrons par la suite (Barney, 2015).

En général, l'envoi d'un message nécessite la connaissance de plusieurs éléments qui sont les suivants :

- l'expéditeur
- type de données à envoyer
- taille des données

- le destinataire

Cette liste peut varier d'une implémentation à une autre et inclure plus d'éléments tels que les rangs des destinataires, le type d'envoi...

Afin d'implémenter un tel mécanisme, il existe deux approches qui permettent le passage de messages dans un environnement distribué : PVM et MPI.

1.2.4.1 PVM

PVM (Parallel Virtual Machine) consiste comme son nom l'indique en une machine virtuelle qui fait abstraction du réseau connectant plusieurs équipements (ordinateurs, serveurs…). Il a été développé en 1989 à Oak Ridge National Laboratory. Cette approche mise sur l'hétérogénéité du système, en effet elle permet à différents types d'ordinateurs d'être vus en tant qu'une seule machine virtuelle où PVM gère lui-même la communication des données, le routage ainsi que la planification des tâches à travers le réseau de manière transparente. Un deamon[3] est exécuté sur chaque ordinateur composant le réseau, ces processus fonctionnent de manière concurrente et permettent ainsi de simuler un seul système(Al Geist, 1994).

PVM se veut simple d'utilisation pour l'utilisateur qui se contente d'écrire son programme en forme de tâches qui coopèrent entre elles. Ces taches ont accès aux ressources PVM à travers une bibliothèque de fonctions standard. Ces fonctions permettent d'organiser les tâches, c'est-à-dire le lancement ou la terminaison d'une tâche, la gestion de la concurrence et de la synchronisation. Les primitives de passage de messages disponibles offrent la possibilité d'envoyer des structures données en différents modes : envoi simple ou multiple. Les tâches font ici office d'unités de calcul pour PVM et sont semblables aux processus dans les systèmes UNIX. Plusieurs utilisateurs peuvent définir chacun sa propre machine virtuelle et leurs applications peuvent s'exécuter simultanément.

---

[3] Processus qui s'exécute en arrière plan

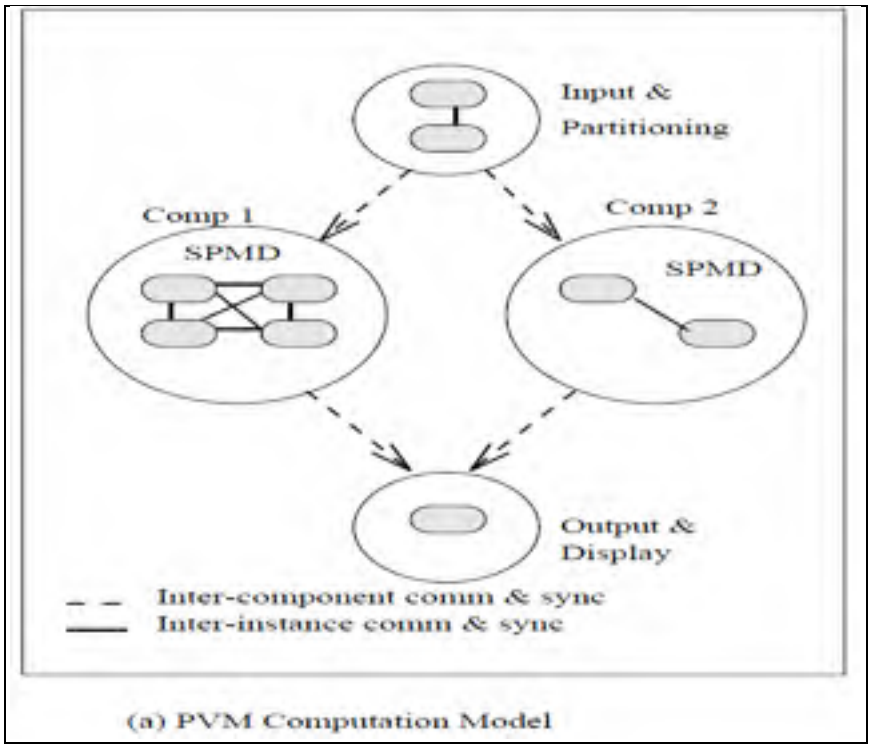

Figure 1.2 Communication dans PVM.
Tirée de (Al Geist, 1994).

Le modèle de calcul de PVM est basé sur la décomposition des applications en plusieurs tâches qui s'occuperait chacune d'une partie du traitement. Une des manières serait de décomposer selon les fonctions présentes dans l'application où chaque fonction serait une tâche (e.g une tâche pour la lecture des données, une tâche pour l'affichage…). Cette façon de faire est appelée parallélisme fonctionnel. Il existe un autre moyen plus utilisé pour la décomposition appelé parallélisme de données où toutes les tâches seraient semblables, mais chacune travaillerait sur une partie distincte des données. La figure 1.2 montre un exemple d'interaction des tâches dans PVM où les deux modèles parallélisme sont utilisés.

L'architecture de PVM permet aux tâches de s'échanger des messages, et suppose que n'importe quelle tâche a la possibilité de communiquer avec toutes les autres sans restrictions sur le nombre ou la taille des messages échangés. La taille des buffers alloués est donc dynamique et ne dépend que de la mémoire disponible dans le système, et dans le cas où l'utilisateur essaie d'envoyer des données de taille supérieures à la mémoire disponible, PVM

retourne une erreur sans pour autant interrompre l'exécution. En effet, le modèle de communication implémenté par PVM permet différents modes d'envoi :

- Envoi asynchrone bloquant : ce mode d'envoi permet le retour de la fonction d'envoi dès que le tampon d'envoi est libéré (toutes les données ont été envoyées) et indépendamment de l'état du destinataire.
- Réception non bloquante : dans ce mode, la fonction de réception retourne immédiatement soit avec les données ou avec un flag indiquant la non-réception de celles-ci.
- Réception asynchrone bloquante : La fonction de réception ne retourne que quand les données sont dans le tampon de réception.

Ces trois modes concernant la communication point à point, PVM permet aussi la communication de groupe tel que le broadcast et le multicast à un groupe de nœud prédéfini.

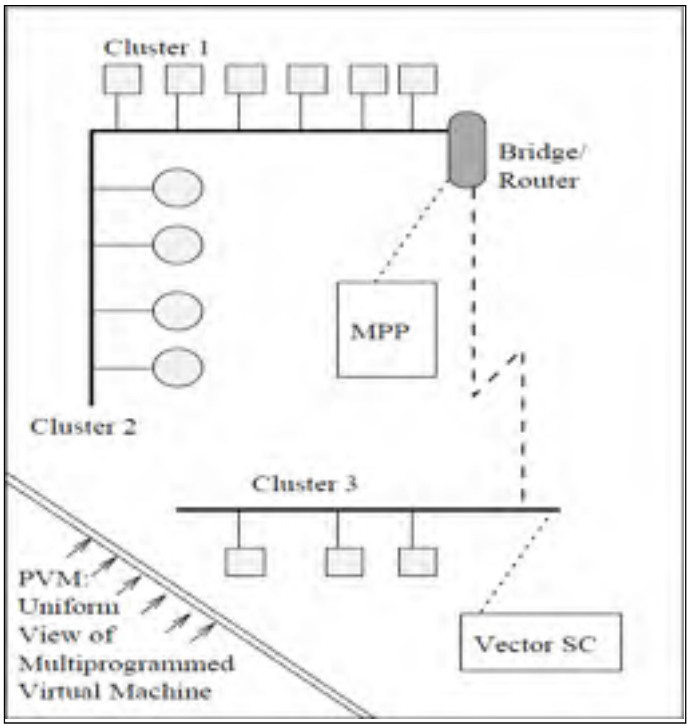

Figure 1.3 Architecture de PVM.
Tirée de (Al Geist, 1994).

La figure 1.3 représente l'architecture globale de PVM et met en valeur l'hétérogénéité du système où on peut voir des clusters de PC connectés avec une machine massivement parallèle.

1.2.4.2 MPI

MPI est une librairie standard de passage de messages développée lors du forum MPI regroupant plus de 40 organisations incluant des chercheurs, des développeurs de librairies logicielles ainsi que des utilisateurs. Les travaux ont débuté en 1992 et la première version de MPI a été achevée en mai 1994(Forum, 2012).

Le but derrière la réalisation de MPI est d'offrir un standard pour l'écriture de programmes de passage de message qui soit pratique, portable, efficace et flexible. Pour cela, des objectifs ont été fixés :

- MPI doit être une librairie permettant de développer des applications parallèles et non un système d'exploitation distribué.
- MPI doit donner la capacité de développer des applications hautement performantes quand déployé sur des systèmes de haute performance. C'est pour cela que les copies de mémoires ne sont pas obligatoires.
- Afin de favoriser la portabilité, MPI doit être modulaire. Cette modularité implique des choix de développements tels que toutes les références doivent être relatives à des modules et non au programme en entier.
- MPI doit être extensible afin de satisfaire aux éventuels besoins futurs, ce qui implique de ne pas viser un seul langage orienté objet, mais plutôt un ensemble de concepts orienté objet.
- MPI doit supporter l'hétérogénéité, c’est-à-dire fonctionner sur plusieurs types d'architectures en parallèle.
- MPI devrait donner les moyens de gérer la concurrence entre les processus.

Toutes ces conditions ont été prise en compte pour le développement de la première version de MPI qui était compatible avec C,C++ et Fortran.

1.2.4.3 Modèle de programmation

Au départ, MPI a été pensé pour les systèmes à mémoire distribuée, vu qu'ils étaient de plus en plus populaires ce qui donnait l'architecture de la figure 1.4 où chaque nœud possède sa propre mémoire locale.

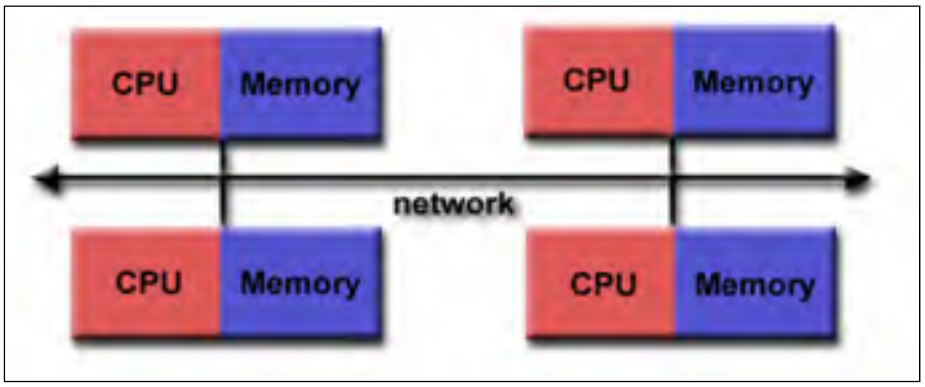

Figure 1.4 Mémoire distribuée.
Tirée de (Barney, 2015).

Avec l'apparition des multiprocesseurs symétriques, il a été possible de les combiner afin d'avoir un architecture hybride comme le montre la figure 1.5.

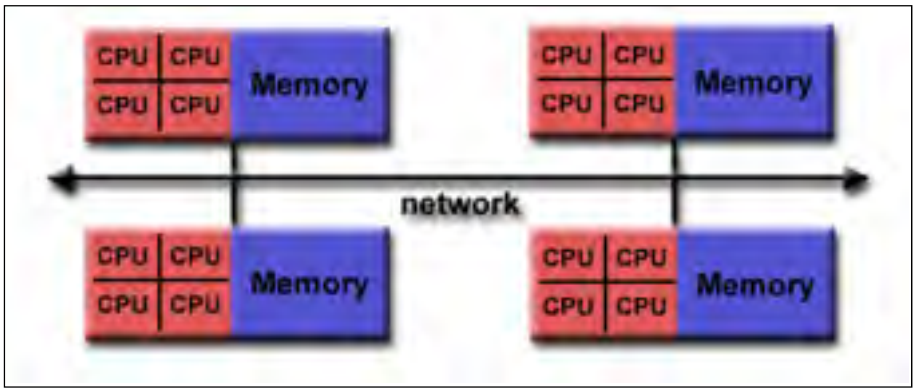

Figure 1.5 Architecture hybride.
Tirée de (Barney, 2015).

MPI est capable de s'adapter aux deux types d'architectures et ses développeurs ont pris en compte différents types de protocoles et d'interconnexions. Cependant , le modèle de programmation est essentiellement basé sur un système à mémoire distribuée, et ce indépendamment de l'architecture sous-jacente, de plus, tout parallélisme doit être implémenté par le programmeur.

1.2.4.4 Abstraction

Le standard MPI définit un API de haut niveau qui fait abstraction des couches de transport. En effet, le programmeur n'a pas besoin de connaitre le protocole réseau utilisé pour communiquer entre les différents processus, il a juste besoin d'utiliser une primitive MPI qui serait sous la forme 'Le processus de rang x veut envoyer les données de type k au processus de rang y'. Les données sont transférées sans aucune mention de connexion établie et sans que le programmeur ait à spécifier une adresse réseau. Cette abstraction a pour effet de cacher au niveau applicatif l'architecture sous-jacente qui pourrait s'avérer complexe et alléger ainsi le code. D'autre part, une telle approche vise à augmenter la portabilité vers différentes plateformes et protocoles de transport.

1.2.4.5 Communication

MPI définit plusieurs modes de communication entre les processus qui se divisent essentiellement en deux types: point à point et communication de groupe(Forum, 2012).

A. Point à point

- Envoi et réception bloquants : Cette fonction ne retourne que lorsque le buffer d'envoi est vide et prêt à l'écriture et que toutes les données ont été reçues. Ce mode permet de savoir si toutes les données ont été reçues et qu'ils peuvent être utilisés.

- Envoi et réception non bloquants : Cette fonction retourne sans attendre la fin de l'opération, ce qui permet d'accélérer le calcul parallèle tout en limitant les possibilités de blocage.

B. Communication de groupe

La communication de groupe ou collective s'effectue en invoquant la même routine dans les différents processus d'un même groupe. Les mêmes arguments doivent être mentionnés pour les émetteurs et les récepteurs, et l'un des plus importants est le communicateur qui sert à définir le groupe concerné. Certaines opérations ont comme source d'envoi ou point de destination un seul processus, il est alors considéré comme processus 'root' et certains arguments ne sont pris en considération que par ce dernier.

Les routines de communication collective, schématisées dans la figure 1.6 sont les suivantes :

- Broadcast : Les mêmes données sont envoyées d'un nœud vers tous les autres nœuds du groupe.
- Scatter : Partitionne les données d'un nœud et envoie chaque portion vers un autre nœud.
- Gather : L'opposé de scatter, rassemble les données depuis différentes sources vers un seul nœud.
- AllGather : Envoie toutes les données de tous les nœuds vers tous les nœuds du groupe. À la fin de cette opération, tous les nœuds auront le même ensemble de données.

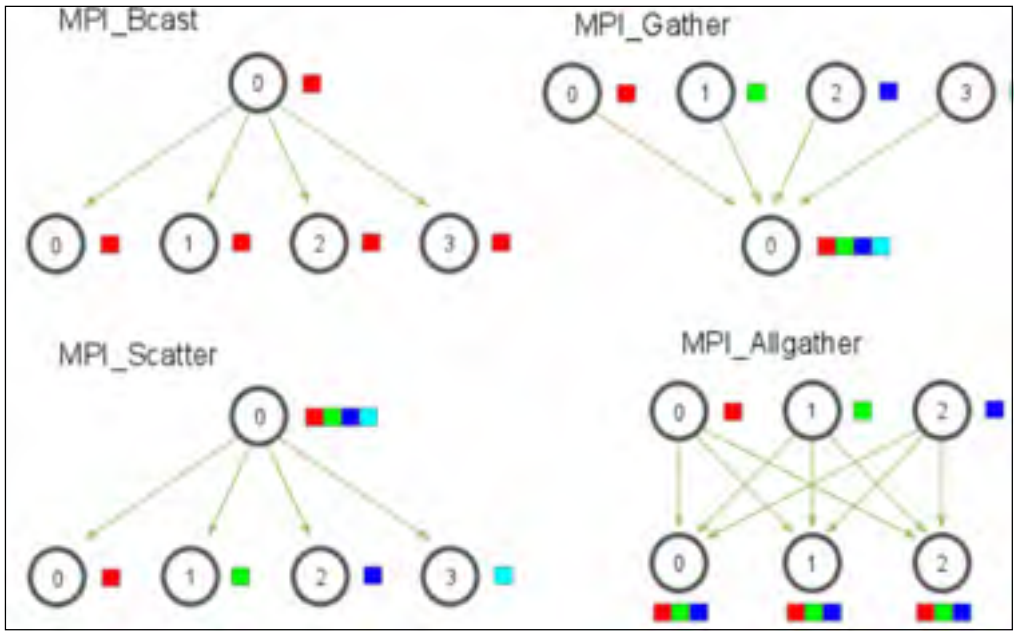

Figure 1.6 Communication collective dans MPI.
Tirée de (Kendall, 2016).

1.3 Appareils ARM

1.3.1 Historique

ARM est l'acronyme pour *Advanced RISC Machines* (Machines RISC Avancées) dont le projet a été lancé par Acorn Computers Ltd et le premier processeur ARM1 a vu le jour en 1985. Cette première version visait à offrir une faible latence et être économique afin de concurrencer le 6502 de MOS Technology(Levy et Promotions, 1990).

Le succès était tel, que Acorn a ouvert une nouvelle compagnie consacrée uniquement à ces nouveaux processeurs ARM Inc. Par la suite, en 1992, Apple a pu bénéficier de cette technologie en collaborant avec ARM pour le développement de ARM6 avec lequel ses agendas électroniques ont été équipés. C'est alors que la liste des fabricants ayant une licence ARM a commencé à s'élargir pour inclure Intel, LG, Marvell, Microsoft, Nvidia, Qualcomm, Samsung, Sharp, ST microelectronics, Symbios Logic, Texas Instruments, VLSI Technology, Yamaha, Zilabs(Levy et Promotions, 1990).

De nos jours, ARM est l'architecture la plus utilisée au monde avec 98% des Smartphones qui contiennent au moins un processeur ARM et 10 milliards de processeurs produits en 2013(Shilov, 2014).

1.3.2 Évolution de l'architecture ARM

Les objectifs derrière l'architecture ARM sont les suivants :

- Optimiser le rapport performance/prix au lieu d'essayer de concurrencer les microprocesseurs présents sur le marché en terme de performance pure. En effet, contrairement à l'architecture CISC, RISC a besoin de moins de silicone tout en offrant une performance similaire à d'autres processeurs de la même gamme.
- Réduire le temps de conception en offrant une bibliothèque de cellules standard qui permet, à l'aide d'un logiciel appelé QuickDesign de concevoir des processeurs personnalisés grâce à des macros. Les microprocesseurs ARM supportent l'architecture 32-bit (ARMv7) et 64-bit (ARMv8).
- Réduire la consommation d'énergie. Ceci est d'ailleurs l'avantage principal des processeurs ARM et son argument de vente le plus important. Ceci est rendu possible grâce au jeu d'instructions simplifié de l'architecture RISC (qui nécessite notamment moins de transistors).

ARM a commencé par fournir des processeurs exclusivement 32 bits avec le cortex A15 (voire figure 1.7) comme sa version la plus rapide. Par la suite, la compagnie s'est lancée dans l'architecture 64-bit avec le cortex A53 et le cortex A57 (figure 1.8). Cet intérêt pour le 64 bit s'est manifesté du fait du développement des appareils mobiles et de l'émergence de nouvelles technologies telles que la reconnaissance vocale, les jeux 3D avec un rendu de plus en plus réaliste et un affichage haute résolution. Ces avancées ont vite poussé l'architecture 32 bit à ses limites d'où la nécessité de passer au 64 bit(arm, 2015a).

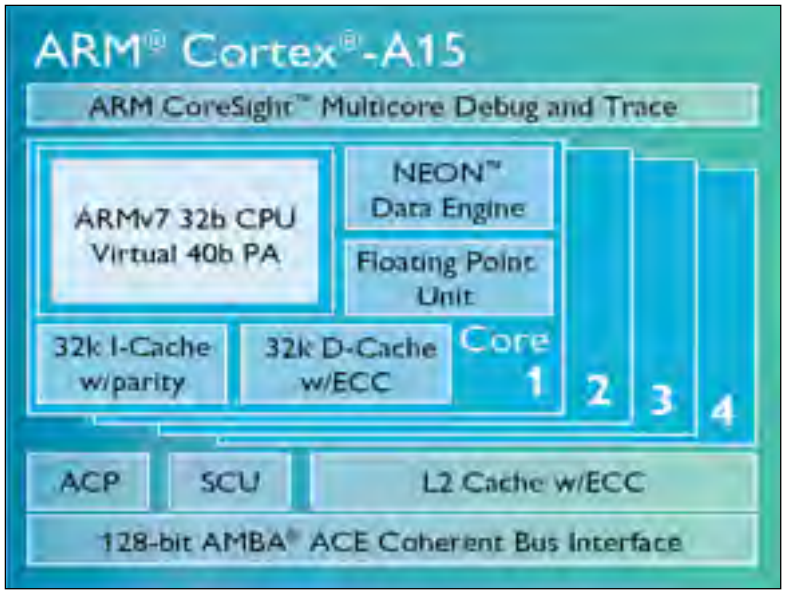

Figure 1.7Architecture Cortex A15.
Tirée de(arm, 2015b).

En passant au 64 bit, ARM a fait en sorte que ses nouveaux processeurs soient capables d'exécuter aussi bien des applications 32 bit que des applications 64 bit. Dans le cas des appareils tournant sous android par exemple, il suffit que le noyau soit adapté pour du 64 bit pour que le reste (les librairies, les applications...) soit du 32 bit ou du 64 bit. Toutefois, le passage du 32 bit au 64 bit ne veut pas dire une plus haute consommation d'énergie. Comme cité plus haut, l'un des objectifs principaux de l'architecture ARM est l'économie d'énergie. En effet, dans certains cas, un processeur 64 bits pourrait s'avérer plus économique qu'un 32 bit puisqu'il est plus rapide et met moins de temps à réaliser les calculs.

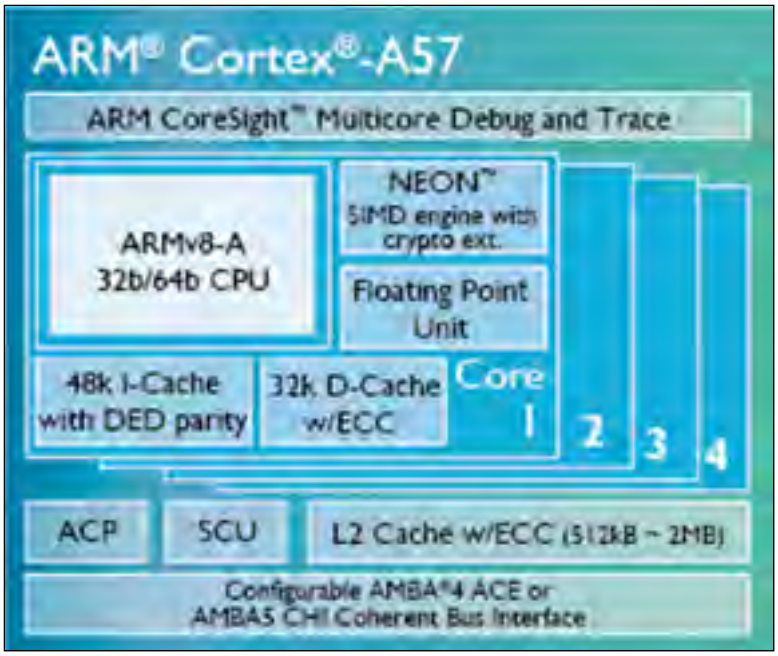

Figure 1.8 Architecture Cortex A57.
Tirée de (arm, 2015b).

Malgré le fait que le Cortex-A15 soit l'architecture ARM la plus rapide, elle est relativement récente et n'est pas encore largement utilisée comparée au Coretx-A9. Ce dernier a été introduit depuis 2008 et équipe la majorité des Smartphones, tablettes et systèmes embarqués. Sa caractéristique principale est son ratio performance/énergie qui est élevé par rapport aux autres architectures (arm, 2015b). La figure 1.9 montre l'architecture Cortex-A9.

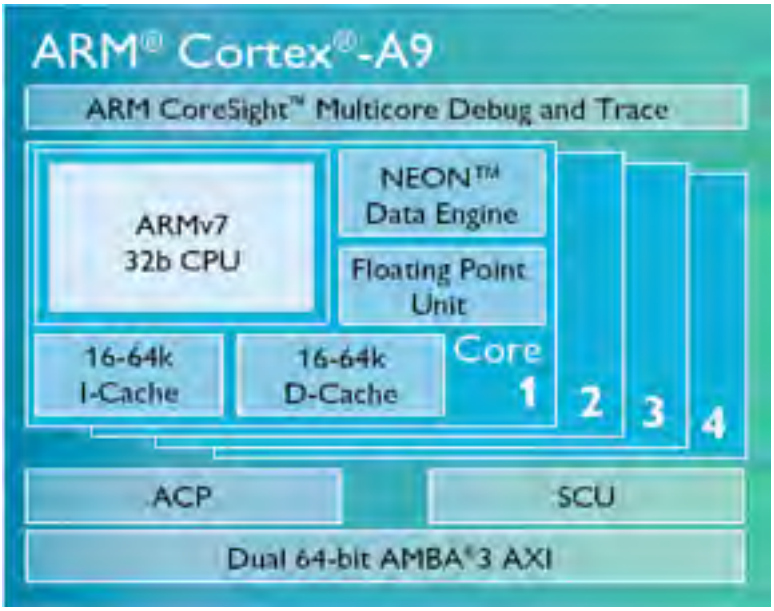

Figure 1.9 Architecture Cortex A9
Tirée de (arm, 2015b).

1.4 Virtualisation

La virtualisation est une technologie qui a attiré de plus en plus de professionnels et de chercheurs dans le monde de la technologie de l'information. Les avantages apportés par cette technologie sont si nombreux qu'elle suscite un intérêt de plus en plus grandissant cette dernière décennie. En effet, parmi les raisons qui ont suscité cet intérêt, on peut citer des facteurs comme l'économie des ressources, l'économie d'énergie et l'augmentation de la compétition(Marshall, 2016).

La virtualisation est donc utilisée par les entreprises afin de réduire le coût d'éventuels centres de données, diminuer la consommation d'énergie ainsi que l'espace nécessaire pour le déploiement du matériel physique. De plus, cette technologie permet d'assurer une haute disponibilité pour les applications jugées critiques. Ceci simplifie les opérations telles que la migration et le déploiement des applications(Burger, 2012).

1.4.1 Définition

La virtualisation consiste à exécuter sur un seul ordinateur physique, un ou plusieurs systèmes d'exploitation comme un (des) simple(s) logiciel(s), séparément les uns des autres. Chaque machine virtuelle (système d'exploitation) partageant les ressources de cet ordinateur physique dans plusieurs environnements. La virtualisation peut être effectuée sur plusieurs types d'entités tels que les ordinateurs de bureau, les unités de stockage et les serveurs.

Afin d'accomplir la virtualisation, plusieurs méthodes sont disponibles. Même s'ils ont pour but ultime la virtualisation, ces derniers se différencient par leur approche et leur architecture, ainsi que par le matériel qu'ils visent à virtualiser. Dans ce qui suit, nous présentons les différents types de virtualisation(Vmware, 2016).

1.4.2 Types de virtualisation

Nous distinguons trois importantes techniques de virtualisation : la virtualisation au niveau du noyau, la virtualisation par hyperviseur de type 1et la virtualisation par hyperviseur de type 2(Redhat, 2016).

1.4.2.1 Virtualisation par hyperviseur de type 1

Les hyperviseurs de type 1 s'exécutent directement au-dessus du matériel pour gérer les systèmes d'exploitation virtuels. En raison de cela, cet hyperviseur est aussi appelé natif. Il consiste en un noyau hôte allégé et optimisé sur lequel peuvent s'exécuter des systèmes d'exploitations "invités". Ces derniers peuvent être "conscients" du fait qu'ils sont virtuels et ils feront des appels directement à l'hyperviseur pour accéder au matériel, une telle technique s'appelle paravirtualisation. Des exemples d'hyperviseur de type 1 sont : Xen, Microsoft Hyper-V, VMware ESXi.

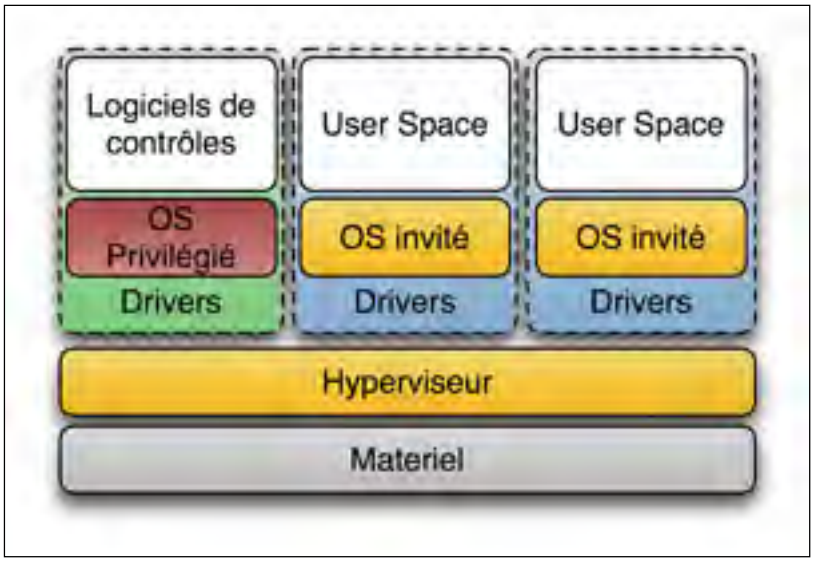

Figure 1.10 Hyperviseur de type 1.
Tirée de (Securityguy, 2014).

1.4.2.2 Virtualisation par hyperviseur de type 2

Les hyperviseurs de type 2 quant à eux s'exécutent comme un processus au sein d'un système d'exploitation hôte. Ils constituent une couche d'abstraction au-dessus de ce dernier. À la différence de la paravirtualisation, les systèmes invités n'ont pas conscience d'être virtuels et croient communiquer directement avec le matériel. Des exemples d'hyperviseurs de type 2 sont VMWare Player, Oracle VirtualBox et KVM.

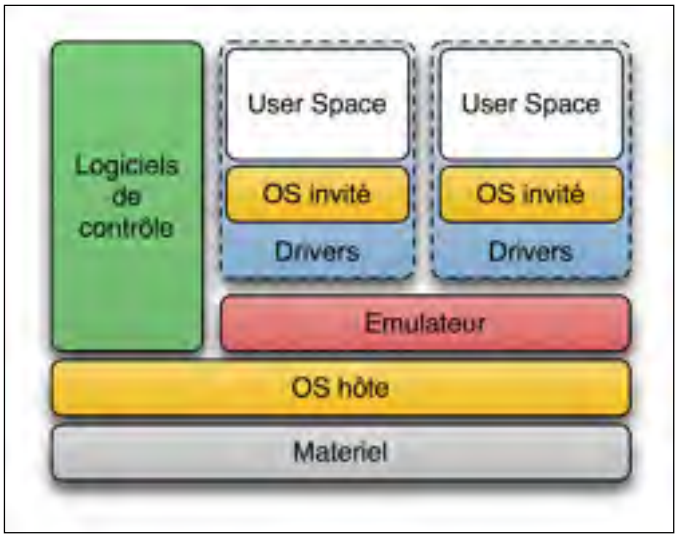

Figure 1.11 Hyperviseur de type 2.
Tirée de (Securityguy, 2014).

1.4.2.3 Virtualisation au niveau de noyau

La virtualisation au niveau noyau du système d'exploitation aussi appelée virtualisation par isolation est une technique permettant de partitionner le système d'exploitation en plusieurs domaines. Cette technique repose surtout sur l'usage des Cgroups qu'offre le noyau Linux.

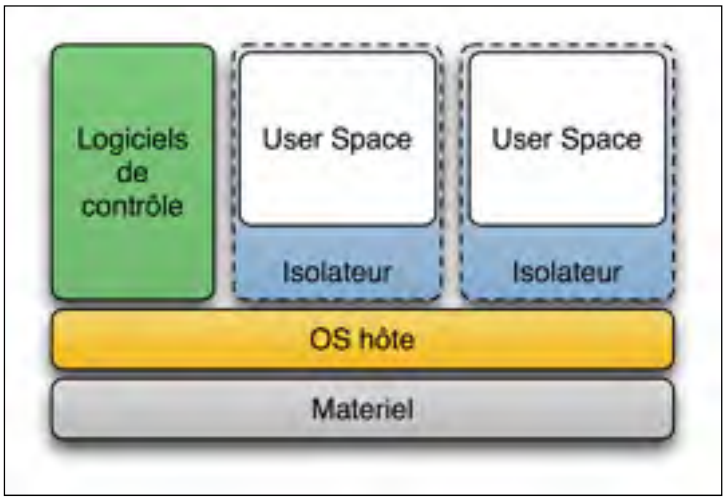

Figure 1.12 Virtualisation niveau noyau.
Tirée de (Securityguy, 2014).

La principale différence avec les autres techniques de virtualisation est qu'elle ne permet de virtualiser qu'un seul type de OS à la fois. Ceci est dû au fait que les différentes instances partagent le même noyau. Cependant, cet inconvénient est comblé par le fait que le coût supplémentaire (overhead) relatif au fait que chaque système virtualisé exécute tout un système d'exploitation est absent. En effet, un seul système d'exploitation s'occupe des appels destinés au matériel.

La figure 1.12 présente les différents types de virtualisation et leurs différentes implémentations. Nous nous intéresserons dans ce travail à la technologie de virtualisation dite la plus légère, à savoir la virtualisation au niveau noyau.

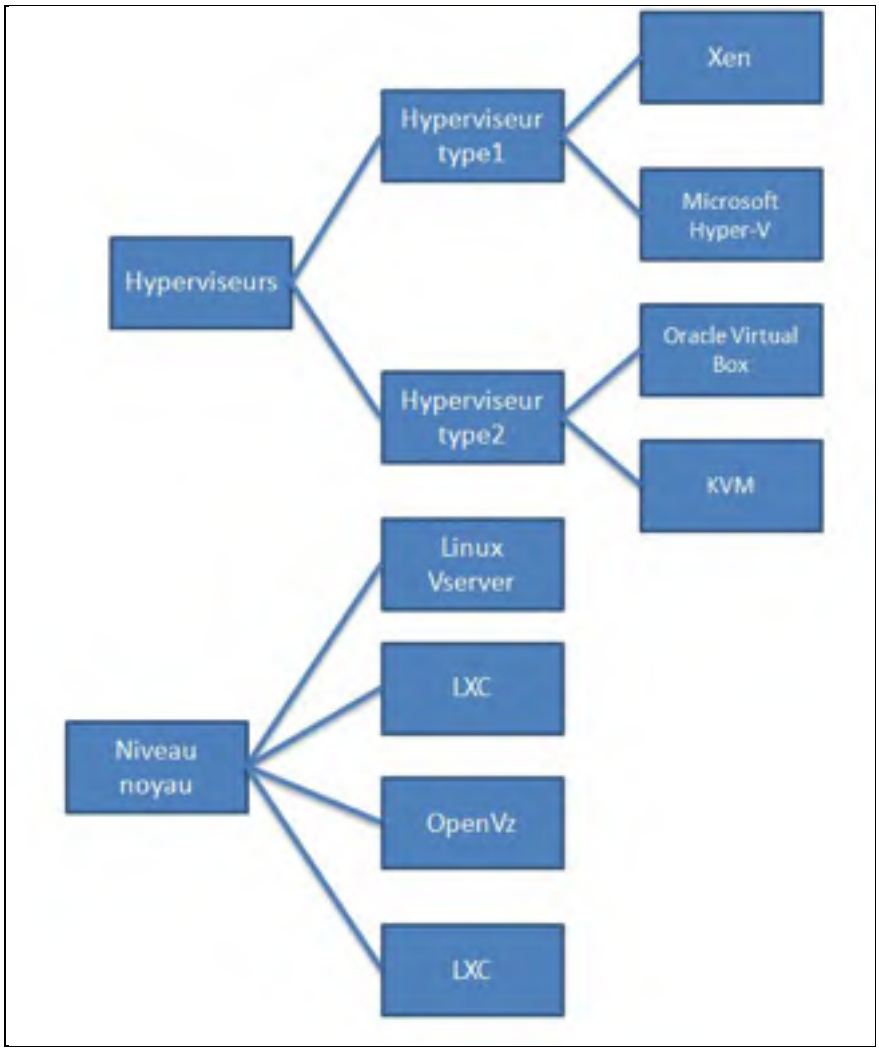

Figure 1.13 Techniques et logiciels de virtualisation.

1.4.3 Docker

Nous avons vu dans la partie précédente les différentes techniques de virtualisation, celles nécessitant un hyperviseur et celle au niveau du noyau qui s'appelle aussi isolation. Nous nous intéressons à cette dernière puisqu'elle offre le moins de coût supplémentaire comparée aux autres techniques (Morabito et al., 2015).

La technique de virtualisation au niveau du noyau a été introduite par les containers Linux qui elle-même se base sur le concept des espaces de nommage au niveau du noyau.

1.4.3.1 Espace de nommage

La fonctionnalité espace de nommage au niveau du noyau a été introduite dans les distributions Linux depuis la version 2.6.24 (RIDWAN, 2015). Cette dernière permet d'isoler un processus (ou un ensemble de processus) en donnant à chacun une différente vue du système. Il existe six implémentations de cet espace de nommage(Biederman, 2006):

- Espace de nommage pid : Permet de séparer les processus en différentes vues. Les processus appartenant à un même espace de nommage ne peuvent pas voir ou interagir avec des processus hors de cet espace.
- Espace de nommage réseau : L'utilité de cet espace est de pouvoir multiplexer l'interface réseau physique entre plusieurs processus. Les processus de différents espaces de nommage verront des interfaces réseau différentes.
- Espace de nommage de montage: Le système Linux possède une structure de données dans laquelle il stocke les différentes partitions du disque qui sont montées, l'endroit où ils ont été montés et d'autres informations comme les permissions... l'espace de nommage de montage permet de cloner cette structure pour chaque espace de nommage de façon à ce qu'un processus appartenant à un certain espace de nommage puisse modifier sa propre copie sans affecter la structure de montage des autres processus du système.
- Espace de nommage utilisateur: Cet espace de nommage permet de donner à un utilisateur les privilèges root au sein de son espace de nommage sans lui donner la possibilité d'interagir avec les processus hors de son espace de nommage.
- Espace de nommage IPC: Cet espace de nommage permet d'isoler un processus en lui donnant ses propres ressources d'intercommunication (messages POSIX ou system V IPC)
- Espace de nommage UTS: cet espace de nommage permet à un processus d'avoir sa propre copie du nom d'hôte et du nom de domaine NIS de façon à pouvoir les changer sans affecter le reste du système.

En prenant avantage du fonctionnement de ces différents espaces de nommage, il est possible de séparer les processus ainsi que les ressources du système en des groupes plus ou moins isolés appelés containers. C'est le principe de la virtualisation avec Linux containers.

1.4.3.2 Architecture de Docker

Docker est une implémentation de l'isolation par containers offerte par le noyau Linux. Les concepteurs de Docker présentent ce dernier comme une plateforme pour les développeurs et les administrateurs système afin de développer, envoyer et exécuter leurs applications (Docker, 2016).

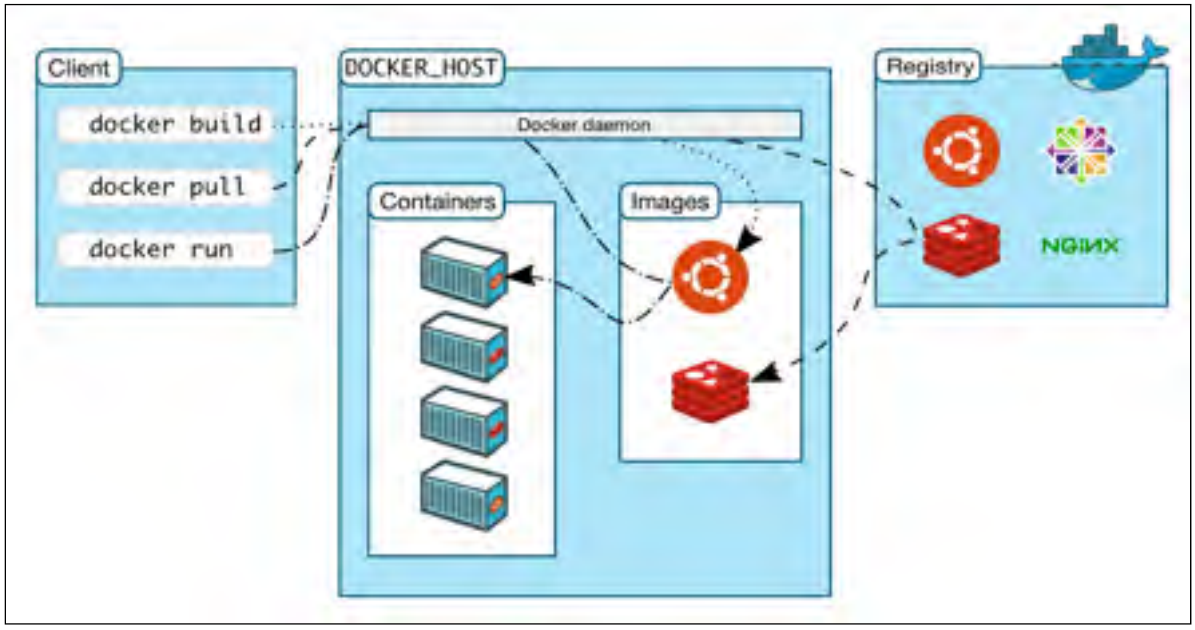

Figure 1.14 Architecture Docker.
Tirée de(Docker, 2016).

Le fonctionnement de docker repose sur la notion d'image et de containers. L'idée est que pour développer une application, le programmeur n'a pas besoin de toutes les fonctionnalités de son système d'exploitation, mais seulement de quelques librairies, API... Une image est donc une version très réduite d'un système d'exploitation ne contenant que l'application et l'environnement nécessaire à son exécution. Cette image peut ensuite être lancée dans un

container isolé. Plusieurs containers de la même image peuvent être lancés simultanément. La figure 1.13 montre l'architecture de docker.

Docker est composé de trois modules, à savoir le registre qui héberge les différentes images, un daemon qui tourne sur l'hôte et qui gère la création des containers à partir des images et enfin le client à travers duquel on communique avec le daemon pour lui indiquer de lancer, stopper, interrompre des containers.

L'une des raisons qui motive notre choix de déployer Docker et le tester sur les cartes ARM est sa facilité de déploiement comme le démontre son architecture.

1.5 Conclusion

Dans ce chapitre, nous avons abordé les différentes notions essentielles à la compréhension de ce travail. Nous avons abordé l'informatique de haute performance et la notion du cluster puisqu'une importante partie de ce projet tournera autour de ce concept. Nous avons aussi abordé l'architecture ARM ainsi que le principe de virtualisation, les différentes techniques de virtualisation et docker comme implémentation d'une de ces techniques.

Dans ce qui suit, nous présenterons la revue de littérature relative à la virtualisation des appareils ARM ainsi que les outils utilisés pour implémenter le passage de messages au sein d'un cluster.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Introduction

Nous présentons dans ce chapitre la revue de littérature relative à notre travail. Entre autres, nous visons dans ce projet à tester les performances des ordinateurs mono-cartes ARM en mettant en place un cluster Beowulf. Pour cela, nous abordons dans un premier temps les études qui ont été faites sur les clusters Beowulf et notamment ceux basés sur les appareils ARM afin de situer le cadre de notre contribution. De plus, nous décidons du meilleur outil en terme de passage de messages en testant les deux principales plateformes utilisées, ainsi nous citons les travaux qui ont visé à comparer PVM et MPI. Aussi, nous mentionnons les études ayant été faites sur le déchargement des traitements de sécurité dans le cas des systèmes à ressources limités. De plus, nous présentons des techniques d'équilibrage de la charge mentionnées dans la littérature.

Notre seconde contribution consistant à appliquer la virtualisation sur les appareils ARM, nous abordons ensuite les études qui ont comparé les différentes technologies de virtualisation.

2.2 Cluster Beowulf

Depuis le premier cluster Beowulf réalisé en 1994 par la NASA (Merkey, 2015), le domaine du HPC est devenu de plus en plus accessible au grand public. En effet, l'idée est que tout particulier peut avoir accès à une puissance de calcul plus ou moins importante en alliant plusieurs des équipements qui ne coûtent pas très cher.

2.2.1 Usage des clusters Beowulf

Plusieurs exemples existent dans la littérature qui décrivent l'utilité des clusters Beowulf, (Datti, Umar et Galadanci, 2015)ont construit un cluster à but éducatif. En effet, les

universités n'ont généralement pas les moyens de s'offrir un supercalculateur et le cluster Beowulf de (Datti, Umar et Galadanci, 2015) permet aux étudiants de s'initier aux HPC. Ce dernier délivre des performances satisfaisantes selon les auteurs qui testent dans un premier des multiplications de matrices et dans un second temps exécutent un algorithme qui trouve les nombres premiers dans un intervalle donné. Le cluster est composé de 5 nœuds et les résultats montre une augmentation maximale de 2,472 fois par rapport à l'exécution sur un seul nœud (séquentiel) pour la multiplication de matrice et de 2,459 pour la recherche des nombres premiers. La figure 2.1 montre l'accélération ainsi que l'efficacité dans le temps d'exécution de la multiplication de matrice selon le nombre de nœuds du cluster.

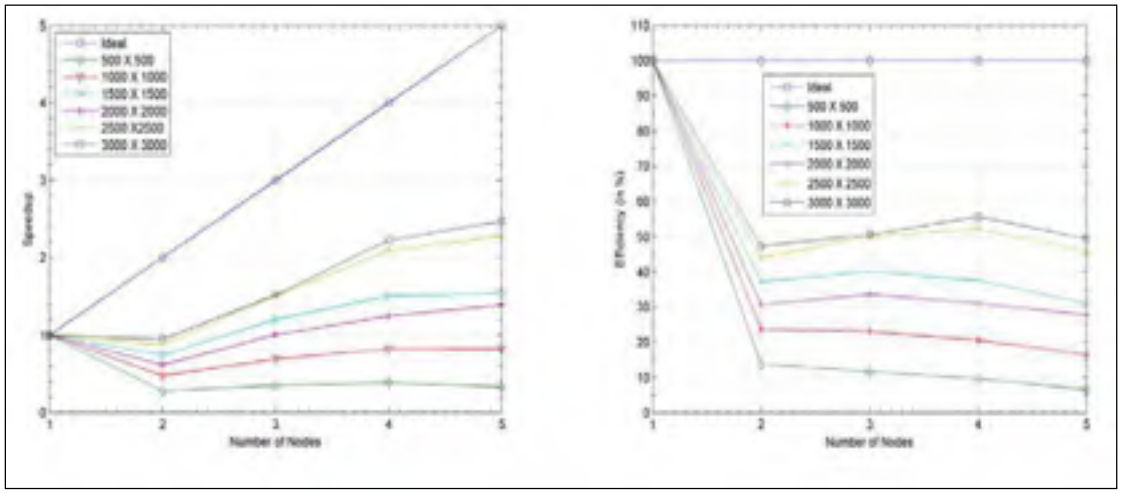

Figure 2.1 Accélération et efficacité de la multiplication des matrices selon le nombre de nœuds et la taille des matrices.
Tirée de (Datti, Umar et Galadanci, 2015)

Bien que le cluster cité ci-dessus soit limité par le nombre de nœuds, il donne un bon exemple de l'accessibilité de l'informatique haute performance, puisque le matériel utilisé est constitué d'ordinateurs déjà disponibles dans l'université et les logiciels utilisés sont libres.

L'exemple suivant, un cluster Beowulf plus performant a été mis en place. (Matthias, 2005)ont mis en place un cluster Beowulf avec 64 processeurs (interconnexion de 32 doubles processeurs) et utilisent un lien Myrinet pour l'interconnexion. Ce dernier est beaucoup plus rapide que les liens Ethernet traditionnels avec un débit de 2 Gbps. De plus, un disque dur

externe RAID (grande vitesse) est relié à l'un des nœuds à travers un lien de 1 Gbps. Un autre nœud sert d'interface d'utilisation et de gestion du cluster et est connecté à ce dernier à travers un lien de 1 Gbps. La figure 2.2 montre l'architecture du cluster de(Matthias, 2005).

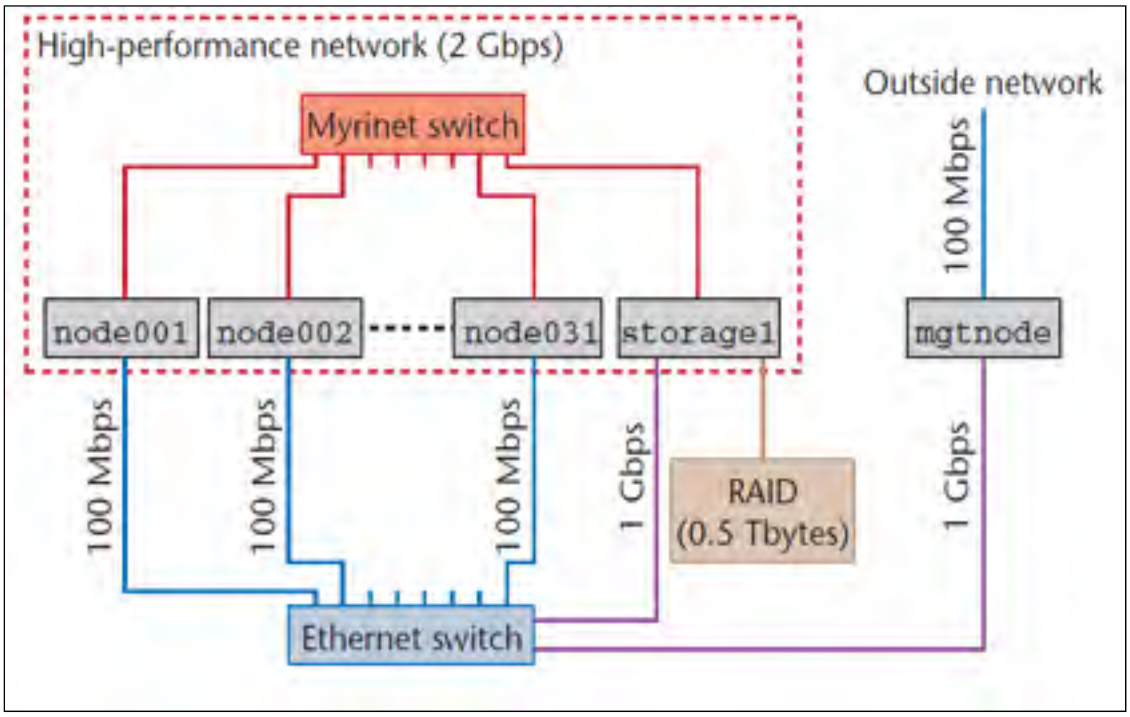

Figure 2.2 Architecture d'un cluster Beowulf.
Tirée de (Matthias, 2005).

Les auteurs ont proposé de résoudre des équations différentielles à l'aide de leur cluster Beowulf, bien que l'article se concentre sur les méthodes utilisées pour optimiser l'algorithme de calcul parallèle, il présente les résultats de l'accélération du temps d'exécution selon les différentes grandeurs du problème. Le temps d'exécution est réduit d'un facteur maximum de 30 par rapport à une exécution séquentielle.

Bien que les auteurs présentent leur cluster comme étant Beowulf, le matériel utilisé n'est pas à la portée de tous les budgets, en effet l'interconnexion se fait à travers un lien Gigabit qui est relativement cher. Le coût reste tout de même négligeable devant celui d'un supercalculateur.

Dans ce qui suit, nous présentons les cluster Beowulf composés de nœuds ayant des processeurs ARM.

2.2.2 Cluster ARM

Dans la littérature nous pouvons trouver deux exemples pertinents de l'utilisation des appareils ARM dans la construction d'un cluster : Le premier type est le cluster destiné au traitement de grands fichiers de données, ce qu'on appelle Big Data, le second type est un cluster sur lequel les applications suivantes ont été déployées : serveur web, transcodage vidéo et une base de données en mémoire. Bien que ces deux types de cluster puissent sembler similaires à première vue, il n'en est rien. Leur point en commun est de rassembler la puissance de plusieurs appareils ARM afin d'accomplir des tâches conséquentes. Cependant, la différence fondamentale est que le traitement de larges fichiers de données implique des ressources et des approches différentes que le traitement de trois applications du second cluster. Dans ce qui suit, nous présentons les études qui se sont intéressées à ces deux types de cluster.

2.2.2.1 Cluster Bigdata

L'étude de (Kaewkasi et Srisuruk, 2014) s'est intéressée à la possibilité de l'utilisation des appareils ARM dans le contexte du BigData. Leur motivation est que les serveurs traditionnels des centres de données sont gourmands en énergie et qu'ils nécessitent de plus des systèmes de refroidissement coûteux.

De ce fait, une entreprise de petite ou moyenne taille ne peut se permettre d'installer et de maintenir un tel matériel localement et se voit contrainte de recourir aux services du cloud où, d'après (Kaewkasi et Srisuruk, 2014) les menaces de sécurité sont importantes. Les auteurs démontrent qu'une entreprise de petite taille peut mettre en place son propre centre de donnée et avoir une meilleure sécurité puisqu'elle a accès à tout le matériel qu'elle utilise, qui n'est pas le cas pour les services cloud.

L'application choisie pour tourner sur le cluster est Hadoop qui est une plateforme servant à traiter les données structurées et semi-structurées. La spécificité de Hadoop est qu'il utilise le patron d'architecture MapReduce et système de fichiers HDFS (*Hadoop Distributed File System*) inspirés des deux publications de Google (Chang et al., 2008) et (Dean et Ghemawat, 2008). Cette plateforme présentant de bonnes performances dans le traitement de larges fichiers de données, elle est très répandue dans le domaine du cloud.

L'équipement utilisé par (Kaewkasi et Srisuruk, 2014)est constitué de 22 ordinateurs mono-cartes ARM équipés chacun d'un processeur cortex-A8 à 1 GHz de fréquence et 1GB de mémoire RAM. Chaque nœud est connecté à un disque dur de 60GB. De plus, 5 sources de courants sont utilisées pour alimenter les nœuds.

Étant donné que Hadoop est écrit en langage Java, son fonctionnement dépend des paramètres de la machine virtuelle Java JVM. Puisque Hadoop a été pensé pour tourner sur des serveurs possédant de grandes capacités, ses paramètres par défaut ne sont pas adaptés pour bien fonctionner sur les processeurs ARM. Dans leur article, (Kaewkasi et Srisuruk, 2014)expliquent la démarche suivie pour modifier les paramètres par défaut du JDK afin d'adapter le fonctionnement de hadoop dans un environnement ARM.

En se basant sur les modifications apportées aux paramètres par défaut du JDK , les auteurs présentent les résultats obtenus en utilisant les différentes configurations qui sont au nombre de six pour en choisir la meilleure en terme de performance et en terme de consommation d'énergie. Parmi les configurations, les auteurs proposent de changer une partie de Hadoop et de la remplacer par une bibliothèque C qui aurait pour effet d'utiliser les instructions des processeurs ARM lors de la compression des données.

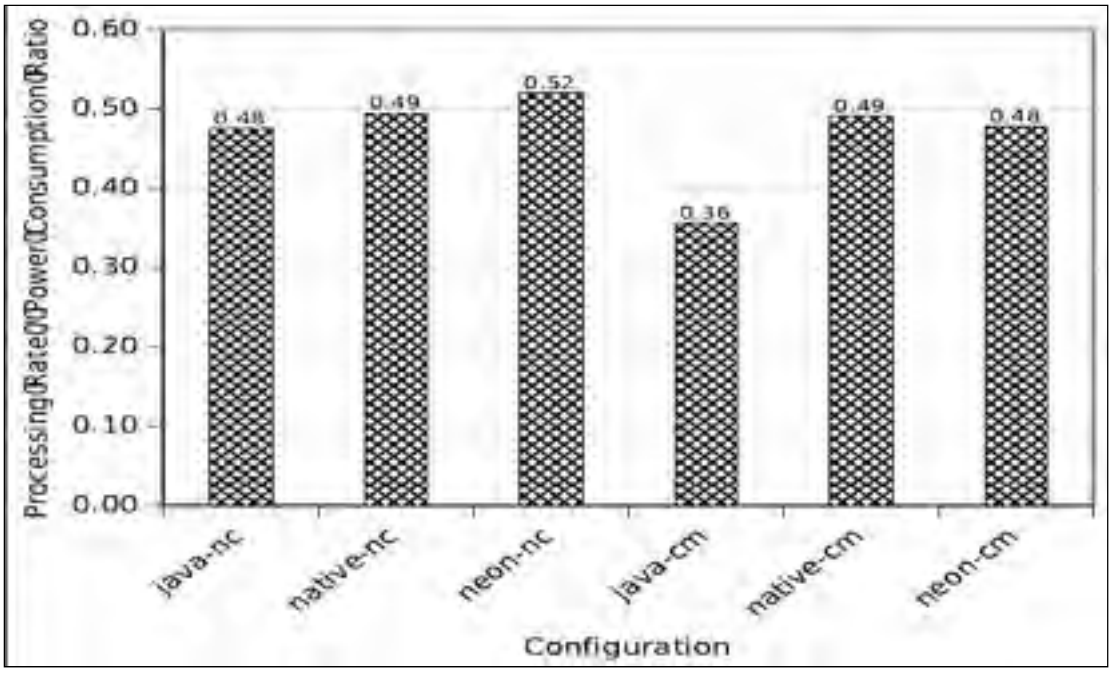

Figure 2.3 Ratio performance/énergie du cluster ARM selon différentes configurations. Tirée de(Kaewkasi et Srisuruk, 2014).

Afin de déterminer la meilleure configuration, les auteurs calculent le ratio performance/énergie comme le montre la figure 2.3 et concluent que l'utilisation des instructions Neon du processeur ARM donnait le meilleur ratio.

Les auteurs concluent que gérer des traitements de Big Data à l'aide des systèmes ARM peut être faisable, mais qu'il faut apporter de grandes modifications.

Il est à noter que notre travail dans ce mémoire vient compléter le travail présenté dans (Kaewkasi et Srisuruk, 2014) puisque nous abordons une autre particularité utilisée dans les centres de données qui est la virtualisation.

2.2.2.2 Mesure du rapport coût/efficacité d'un cluster ARM

Dans (Ou et al., 2012), les auteurs se proposent de comparer un cluster composé d'ordinateurs mono-cartes ARM avec une station de travail Intel x86. La motivation derrière

leur travail est que l'utilisation des machines traditionnelles dans les centres de données en tant que serveurs est de plus en plus répandue ces dernières années (Barroso, Clidaras et Hoelzle, 2013), cependant ces machines n'étant pas été créée avec l'idée d'économiser l'énergie, leur consommation de courant devient problématique.

C'est dans l'optique de trouver une solution au problème de consommation d'énergie que(Barroso, Clidaras et Hoelzle, 2013)s'intéressent aux systèmes embarqués ARM, en comparant leur ratio de coût/efficacité et énergie/efficacité avec un ordinateur traditionnel.

Le cluster ARM est composé de 4 cartes PandaBoard avec chacune un processeur ARM cortex-A9 à 1 Ghz, 1 GB de RAM et une carte SD de 16 GB. La station de travail Intel x86 possède un processeur intel core2 Q9400 avec 2.66 Ghz de fréquence, 8 GB de RAM et un disque dur de 250 GB.

Les auteurs définissent le ratio énergie/performance de la façon suivante :

$$EE = \frac{Work}{Energy} = \frac{Work}{Power * Time} = \frac{Performance}{Power}$$

Figure 2.4 Calcul du ratio énergie/performance.
Tirée de (Ou et al., 2012).

En effectuant les benchmarks sur les trois applications déployées, les auteurs concluent que l'efficacité énergétique du cluster ARM est supérieure à celle de la station de travail Intel dans les trois cas étudiés. Aussi, le modèle de consommation d'énergie des appareils ARM peut être considéré linéaire à l'utilisation de la CPU, alors que celui d'Intel ne l'est pas.

En ce qui concerne le ratio coût/performance, les auteurs ont utilisé le modèle de (Hamilton, 2009)afin d'évaluer le coût de construction d'un centre de données avec des appareils ARM et un autre avec des stations de travail Intel. Leurs estimations prévoient un ratio en faveur du centre de données avec des appareils ARM lorsque les tâches à traiter sont légères, mais cet avantage est perdu au fur et à mesure que les tâches sont lourdes et demandent beaucoup de capacités de calcul.

L'étude de (Ou et al., 2012) est intéressante du fait qu'elle traite de la faible consommation d'énergie des appareils ARM en comparaison avec les ordinateurs possédant l'architecture Intel. Cependant, les auteurs ont diversifié les applications testées sans pour autant diversifier les cartes ARM sur lesquels elles ont été déployées. En effet, les ordinateurs mono-cartes ARM ne sont pas tous conçus dans le même but, par exemple, les cartes PandaBoard ne sont pas optimisées pour l'encodage vidéo alors que des cartes telles que la Parallella, du fait que son coprocesseur est mieux adapté à ce genre de calcul.

2.3 MPI vs PVM

Nous avons introduit dans le chapitre précédent les deux principaux environnements servant à déployer des systèmes parallèles, à savoir PVM et MPI. Ces deux outils ont des approches plus ou moins différentes pour collecter la puissance de calcul de plusieurs processeurs et les utiliser de façon parallèle. En effet, PVM est une machine virtuelle alors que MPI est une librairie standard de passage de messages. Les différences ne s'arrêtent pas là et quelques études se sont intéressées à la comparaison entre PVM et MPI et JAVA.

2.3.1 Différences entre les deux approches

Avec le regain de popularité de la programmation parallèle, les utilisateurs de cette technologie se sont vu offrir le choix entre PVM et MPI, (Sunderam et al., 2001)se sont intéréssés aux points communs et aux différences entre les deux outils. (Sunderam et al., 2001)ont aussi effectué une comparaison entre PVM, MPI, mais aussi JAVA, cette étude est plutôt superficielle et n'est pas allée en profondeur pour discuter de l'architecture de ces approches. Les chercheurs ont mis d'un côté PVM et MPI et de l'autre JAVA en présentant les principes généraux de programmation avec ces outils ainsi que les différentes fonctions pour envoyer et recevoir des messages.

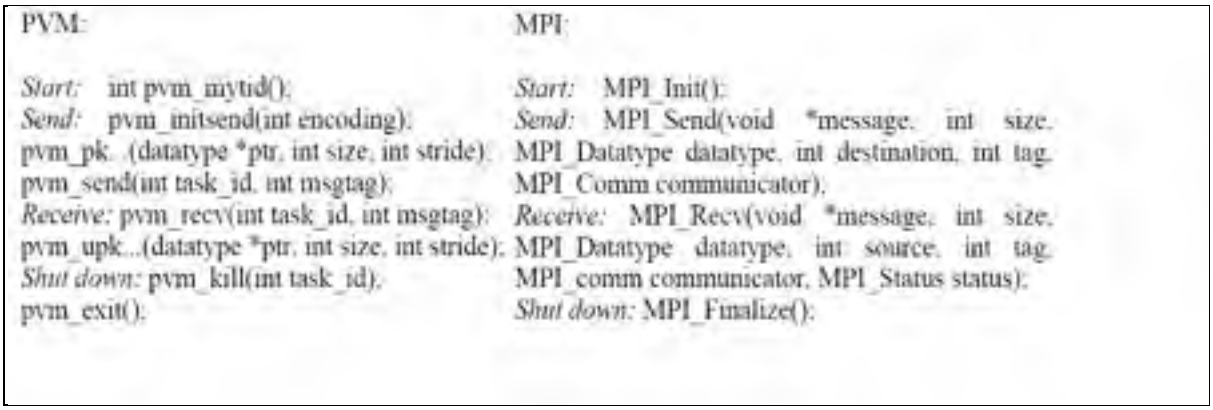

Figure 2.5 processus d'envoi et de réception de données standard de PVM et MPI.
Tirée de (Sunderam et al., 2001).

La figure 2.5 montre que la séquence de code pour initialiser l'environnement, envoyer et recevoir les données est à peu près la même pour PVM et MPI. Ils en concluent donc que ces deux approches sont similaires du point de vue de l'écriture des programmes. Par contre, (Sunderam et al., 2001)affirment que JAVA est différent puisque la programmation parallèle à l'aide de ce dernier est réalisée grâce aux RMI (l'appel distant aux méthodes) et au threading de ce langage. Cependant, une version de MPI a été adaptée pour JAVA et (Baker, Grove et Shafi, 2006) le comparent à MPI de point de vue performance et portabilité. Nous allons aborder ceci dans la partie suivante.

Il est possible d'effectuer de la programmation parallèle distribuée avec JAVA en faisant en sorte de créer un objet qui puisse être threadé. Cela peut se faire soit en créant une classe qui étend la classe Thread, soit en implémentant l'interface Runnable dans la classe. Une fois le Thread créé, il est possible de l'exporter vers une machine distante à l'aide de RMI (Remote Method Invocation) qui permet comme son nom l'indique d'appeler des méthodes distantes. Néanmoins l'objet créé ne se trouve pas physiquement sur la machine distante et il faut utiliser le registre RMI pour pouvoir le référencer.

Cette approche de programmation de JAVA a fait en sorte que les auteurs de (Baker, Grove et Shafi, 2006)considèrent ce dernier comme étant fondamentalement différent de PVM et MPI et plus difficile et complexe à manier.

2.3.2 Comparaison des caractéristiques

D'après (Sunderam et al., 2001), les deux approches ont en commun les points suivants :

- **Portabilité**
  En effet, PVM et MPI sont tous les deux portables puisqu'ils offrent une abstraction du matériel et de l'architecture utilisée. Le code source peut être copié d'une machine à une autre et être exécuté de la même manière.
- **MPMD-SPMD**
  Les deux approches permettent d'utiliser les paradigmes de programmation parallèle MPMD ainsi que SPMD.
- **Interopérabilité**
  L'interopérabilité est définie comme étant la capacité pour différentes implémentations de la même spécification de pouvoir interagir et s'échanger des messages. Pour PVM et MPI, les versions d'une même implémentation sont interopérables.
- **Hétérogénéité**
  Lorsqu'il est question d'interconnecter plusieurs types d'ordinateurs et de les faire travailler en concurrence, on est en face à différents problèmes d'hétérogénéité : Format des données : En effet, le format de certaines données peut être incompatible d'un ordinateur à un autre, rendant les données échangées lors d'un calcul parallèle illisible. Vitesse de calcul : Généralement, les machines interconnectées n'ont pas la même vitesse de calcul ce qui peut engendrer des problèmes de synchronisation. Charge de calcul : même pour des machines identiques, la charge de travail que ces dernières ont peut varier en fonction de leur utilisation. Selon (Sunderam et al., 2001), MPI et PVM supportent tous deux ces types d'hétérogénéité. En effet, pour remédier au problème du format de données, MPI encapsule différents types de messages dans un seul type dérivé permettant un échange hétérogène de données. De plus, une conversion des messages permet la communication entre différentes architectures.
  PVM quant à lui permet le passage de messages contenant plus d'un type de données entre des machines possédant différentes représentations pour ces données.

Le fonctionnement de PVM est centré autour du concept de machine virtuelle, il tire ainsi toutes ses caractéristiques de ces dernières que ce soit l'hétérogénéité, la portabilité ou l'encapsulation des messages. Par contre, le standard MPI ne s'occupe que du système de passage de message et il est même mentionné explicitement que la gestion des ressources est laissée au soin des programmeurs. De ce fait, PVM et MPI différent selon (Sunderam et al., 2001)sur les points suivants :

A. Gestion des processus : PVM en sa qualité de machine virtuelle, offre la possibilité de gérer les tâches en cours et d'en lancer de nouvelles ou d'en terminer dynamiquement. Ceci peut être accompli en utilisant des commandes à partir de la console de pvm, par exemple la commande jobs liste les tâches en exécution et la commande kill permet de terminer un processus déjà lancé. MPI n'offre pas cette possibilité et la gestion des tâches est assez limitée puisqu'il est seulement possible de lancer de nouvelles tâches ou d'envoyer un signal pour en terminer.
B. Gestion des ressources : En terme de ressources, pvm offre là encore une gestion dynamique de par sa qualité de machine virtuelle. Comme pour les processus, il permet d'ajouter ou de retirer des nœuds dynamiquement, un tel contrôle sur l'environnement donne la possibilité de gérer la charge de calcul des différents nœuds de migrer aisément les tâches et d'avoir une assez bonne tolérance aux fautes. Entre autres, la possibilité d'ajouter des noeuds à partir du programme (et non de la console pvm) est très utile dans le cas où une partie à l'intérieur du programme nécessite un calcul parallèle assez lourd. Par contre, la nature statique de MPI fait en sorte que le nombre et la distribution des tâches sontconnus au départ et il n'est pas possible de les changer dynamiquement en cours d'exécution. L'absence de dynamisme est un choix qui a été fait pour privilégier la performance en terme de vitesse d'exécution.
C. Topologie virtuelle : La topologie virtuelle peut être définie comme étant une manière abstraite d’organiser les nœuds afin de mieux satisfaire aux exigences de communication entre les processus ce qui peut permettre de simplifier et parfois d'optimiser le code. MPI offre un plus haut degré d'abstraction que PVM en ce qui

concerne cette topologie, en effet, il permet d'organiser un ensemble de processus en une architecture logique telle que la grille.

La figure 10 montre une grille de calcul où les coordonnées de chaque nœud (processus) sont en fonction des rangs, les lignes représentent les différents chemins de communication entre les nœuds et il existe aussi une limite cyclique c'est-à-dire que le processus de rang 10 peut communiquer avec le processus de rang 1.

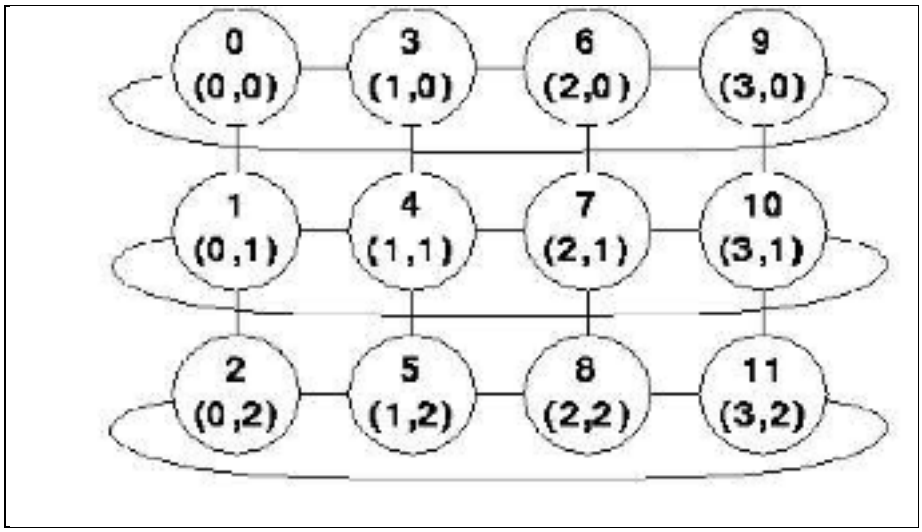

Figure 2.6 Topologie virtuelle de 12 processus en grille avec les coordonnées et le rang de chaque processus.
Tirée de (Geist, Kohl et Papadopoulos, 1996).

D. Topologie de message: MPI offre un plus grand nombre de fonctions d'envoi, d'où une plus grande flexibilité que PVM qui se contente de simples opérations d'envoi (envoie simple, diffusion). Le standard MPI 2 offre 248 fonctions liées au passage de messages(Geist, Kohl et Papadopoulos, 1996).

E. Tolérance aux fautes : La version 1 du standard MPI n'offre aucune tolérance aux fautes alors que la version 2 ajoute la capacité de pouvoir créer dynamiquement de nouveaux processus. Cependant, il n'est toujours pas possible avec MPI de rétablir l'état du système après une erreur causant la perte d'un processus par exemple. Par

contre, PVM donne la possibilité de connaitre l'état du système au cours de l'exécution d'un programme, si l'une des machines virtuelles plante ou qu'une tâche échoue, une notification est envoyée ce qui permet de gérer l'erreur.

D'un autre côté, (Gropp, Lusk et Division)ont aussi effectué une comparaison entre PVM et MPI et font remarquer qu'une erreur assez courante dans la communauté scientifique fait que les comparaisons sont faites entre la spécification de MPI et l'implémentation de PVM. Selon cette étude, la comparaison devrait se faire entre les implémentations de MPI et les implémentations de PVM. Ils justifient cela par le fait que lors de la réalisation du standard MPI (1 et 2), le choix a été laissé aux développeurs d'éventuelles implémentations d'intégrer ou non certaines fonctionnalités. Les auteurs prennent comme exemple la tolérance aux fautes, en effet le standard MPI ne donne pas de directives quant à cette fonctionnalité, elle peut néanmoins être implémentée comme c'est le cas pour MPICH. Donc, affirmer que PVM offre une tolérance aux fautes que ne possède pas MPI n'est pas tout à fait juste. Le tableau 2.1 récapitule les différences et les similarités entre PVM et MPI.

Tableau 2.1Comparaison PVM-MPI

		PVM	MPI
Différences	**Architecture**	Système d'exploitation distribué	Librairie pour écrire des applications
	Gestion des processus	Gestion dynamique	Gestion limitée
	Gestion des ressources	Gestion dynamique	Gestion limitée (statique)
	Topologie virtuelle	Basique	Flexible
	Passage de messages	Ensemble de fonctions limité	Ensemble de fonctions riche
	Tolérance aux fautes	Bonne tolérance aux fautes	Varie suivant les implémentations
Similarités	Portabilité		
	Hétérogénéité		
	Interopérabilité		

2.4 Déchargement des traitements de sécurité

Le besoin de décharger (offloading) les traitements liés à la sécurité s'est accentué avec le développement rapide des téléphones intelligents et autres appareils mobiles (Portokalidis et al., 2010). En effet, ces derniers commencent à assumer le même rôle que les ordinateurs personnels :ils contiennent des informations personnelles, sont utilisés pour accéder à des applications bancaires, regarder des vidéos… La diversité des applications exécutées sur les Smartphones devient alors source de bugs et de vulnérabilités qui peuvent être exploités par des tiers malveillants.

À la différence des PC, les téléphones intelligents ne disposent pas d'une source d'énergie illimitée mais possèdent des batteries dont la durée de vie est relativement courte. C'est pourquoi exécuter des applications de sécurité telle que la détection de malwares ou le chiffrement de données pose un problème d'autonomie. De plus, ces tâches peuvent impacter l'exécution d'autres applications sur le téléphones et dégrader l'expérience de l'utilisateur.

(Oberheide et al., 2008)proposent une solution de déchargement des analyses antivirus des Smartphones vers un serveur distant afin d'économiser les ressources des téléphones et permettre une meilleure détection en utilisant un environnement offrant les ressources suffisantes pour effectuer des analyses antivirus en parallèle.

La motivation de l'étude de (Oberheide et al., 2008)vient du fait que transférer le traitement de détection vers un service réseau, allouerait beaucoup plus de ressources à cette tâche la rendant plus efficace. D'autre part, cela a aussi pour effet de réduire la consommation de ressources sur l'appareil.

La solution proposée est une extension de CloudAV(Oberheide, Cooke et Jahanian, 2008) qui consiste en deux modules. Le premier module s'exécute sur l'hôte et parcours les fichiers locaux, si un fichier est modifié, il est envoyé au second module qui est le service réseau. Ce dernier est supposé contenir des containers virtuels qui exécutent en parallèle plusieurs mécanismes de détection de malwares sur le fichier.

Dans le même registre, (Portokalidis et al., 2010)proposent de décharger l'analyse de sécurité des Smartphones aux serveurs du Cloud afin de bénéficier de plus de ressources et d'économiser la batterie des appareils.

L'approche de (Portokalidis et al., 2010)consiste à enregistrer les traces d'exécution des applications sur le téléphone et les envoyer vers une réplique de ce dernier sur le Cloud. En utilisant les traces, l'émulateur rejoue l'exécution de l'application et des mécanismes de sécurité sont exécutés afin de détecter un éventuel comportement malicieux. Les analyses de sécurité consistent en les anti-virus offerts par le Cloud, les scanners de mémoires et les détecteurs d'anomalies se basant sur les appels systèmes.

Nous constatons à travers les deux études que nous venons de citer le besoin de décharger les traitements liés à la sécurité des Smartphones vers un environnement possédant plus de ressources et de capacités de calculs. Les deux approches ont choisi de transférer ces traitements vers les services du Cloud.

Cependant, selon (Kaewkasi et Srisuruk, 2014) les serveurs du Cloud sont eux-mêmes proies à différentes attaques et y traiter des données sensibles serait un risque, d'où leur proposition de centre de donnée à coût réduit composé d'ordinateurs mono-cartes à architecture ARM. Nous proposons dans notre étude de mettre en place un cluster ARM et évaluer ses capacités à exécuter des mécanismes de sécurité qui sont le chiffrement et la détection de comportements malicieux.

2.5 Technologies de virtualisation

La virtualisation a connu un essor important depuis une dizaine d'années, et diverses technologies de virtualisation ont vu le jour. Les deux techniques qui sont les plus populaires sont celles reposent sur les hyperviseurs, cependant la virtualisation au niveau noyau (légère) gagne en popularité(Hess, 2016).

(Morabito et al., 2015) effectuent une comparaison de ces deux techniques. Les auteurs motivent leur travail par le fait que le besoin en virtualisation s'accroit de jour en jour surtout pour les centres de données qui bénéficient de cette technologie pour tirer le maximum de leurs serveurs en terme de ressources ainsi que pour avoir des environnements plus sécurisés en tirant avantage de l'isolation qu'offre la virtualisation.

Les auteurs procèdent à une comparaison du point de vue de l'approche et de l'architecture des deux technologies : La virtualisation basée sur les hyperviseurs simule tout le matériel ce qui apporte un coût supplémentaire lors de l'accès à ce dit matériel, puisque les appels devront passer par l'hyperviseur d'abord. Par contre, pour la virtualisation basée sur l'isolation, ce coût n'apparait pas, car le matériel n'est pas virtualisé. Une autre différence est le fait que l'isolation permet d'avoir des images (des systèmes virtualisés) plus légères puisqu'elles partagent toutes le même noyau. Par contre, pour les hyperviseurs les images

sont relativement plus lourdes puisqu'il s'agit de systèmes d'exploitation complets. Cette caractéristique implique aussi un désavantage pour l'isolation qui ne permet pas de virtualiser des systèmes autres que Linux, ce qui n'est pas le cas pour la virtualisation basée sur l'hyperviseur.

Les auteurs de (Morabito et al., 2015)choisissent de considérer KVM pour représenter la virtualisation par hyperviseur, LXC et Docker pour la virtualisation par isolation. De plus, ils testent un nouveau système d'exploitation servant d'"invité" et qui tourne exclusivement sur KVM, il s'agit de OSv (Kivity et al., 2014).

Différents benchmarks sont utilisés pour tester la CPU, l'accès à la mémoire, l'accès au disque et les performances des interfaces réseau sur les différents environnements. Comme repère, ces expérimentations sont aussi effectuées sur un système vierge (sans virtualisation).

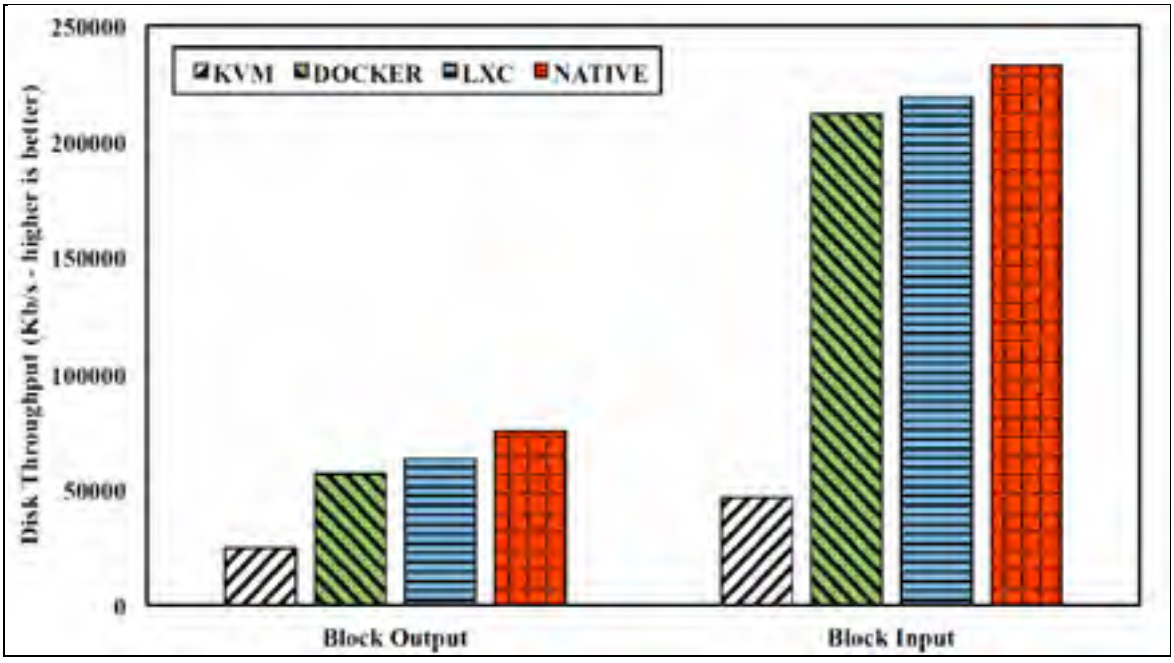

Figure 2.7 Entrée/sortie disque.
Tirée de (Morabito et al., 2015).

Les résultats montrent que les technologies de virtualisation légères présentent des coûts supplémentaires négligeables dans tous les cas considérés qui sont inférieurs aux coûts engendrés par KVM. Entre autres, ce dernier montre des coûts supplémentaires relativement élevés en ce qui concerne les entrées/sorties du disque comme le montre la figure 2.7.

Dans le même registre, l'étude fournie par IBM (Felter et al., 2014) offre une comparaison entre la virtualisation avec KVM et celle utilisant Docker et LXC. Cette étude est aussi motivée par l'utilisation intensive de la virtualisation dans le cloud et que les coûts supplémentaires introduits par les machines virtuelles impactent négativement le marché.

L'étude d'IBM, à l'instar de (Morabito et al., 2015), mesure les performances reliées à la CPU, la mémoire, le réseau et les entrées/sorties du disque. L'étude conclut que Docker et LXC offrent des performances au moins égales à celles de KVM et parfois supérieures. Les auteurs notent que les coûts supplémentaires ajoutés par KVM sont en diminution par rapport aux versions précédentes. Ils notent aussi le coût de ce dernier dans les opérations d'entrée/sortie et recommandent d'éviter son usage pour les tâches qui sont sensibles aux latences entrée/sortie.

En ce qui concerne Docker, les auteurs notent que les réglages par défaut du système de fichiers utilisé induisent une dégradation de la performance pour les accès au disque. Aussi, le NAT utilisé par Docker comme paramètre réseau par défaut est jugé une source de coût supplémentaire.

Les auteurs proposent l'idée d'utiliser les deux technologies KVM et Docker de façon à bénéficier des avantages apportés par les deux solutions et recommandent dans ce cas d'exécuter KVM au sein d'un container et non l'inverse.

2.6 Équilibrage de la charge

L'équilibrage de la charge à travers un cluster n'étant pas l'axe principal de recherche dans notre travail, nous ne présentons ici que l'exemple le plus pertinent dans la littérature qui correspond le plus à notre cas.

(Sharma et Kanungo, 2014) présente un mécanisme d'équilibrage de la charge dynamique pour un cluster hétérogène. Les auteurs justifient leur proposition par le fait qu'au sein d'un cluster hétérogène, les capacités et les ressources disponibles sont différentes d'un nœud à l'autre, d'où l'intérêt de distribuer la charge de travail de manière qui conviendrait aux ressources disponibles sur chaque noeud.

Les auteurs définissent une variable AF_i qui est le facteur d'assignation pour un nœud i. La contrainte est que la somme de tous les AF_i doit être égale à 1. AF_i est calculé de la façon suivante :

$$AF_j = \frac{I_j(P)}{\sum_{j=1}^{N} I_j(P)}$$

Figure 2.8 Calcul du facteur d'assignation pour un noeud j.
Tirée de (Felter et al., 2014).

Où I_j est le nombre d'itérations d'une application effectuées par le nœud j au cours d'un intervalle de temps. Par la suite, des priorités sont assignées à chaque nœud en fonction de son facteur d'assignation. Les priorités et les facteurs d'assignations sont inversement proportionnels : Plus un nœud à un facteur élevé, moins sa priorité est importante.

La charge est ensuite distribuée selon la priorité, les nœuds ayant la plus haute priorité, recevront plus de charges de travail.

Pour leurs expérimentations, les auteurs ont essayé plusieurs scénarios, ils ont varié la stratégie d'exécution avec les paramètres suivants : distribution équitable/facteur d'assignation, MPI ou Hybride (utilisant MPI et OpenMP). Les différentes configurations sont montrées dans la figure 2.9.

Tasks Distribution	MPI/ Hybrid	Load Balancing	Policy Name
Even	MPI	Yes	EDMLB
Based on AF	MPI	Yes	AFDMLB
Even	MPI	No	EDMWLB
Based on AF	MPI	No	AFDMWLB
Even	Hybrid	Yes	EDHLB
Based on AF	Hybrid	Yes	AFDHLB
Even	Hybrid	No	EDHWLB
Based on AF	Hybrid	No	AFDHWLB

Figure 2.9 Stratégies d'exécution au sein du cluster.
Tirée de (Felter et al., 2014).

Les auteurs concluent finalement que leur algorithme permet de réduire le temps d'exécution du cluster dans toutes les stratégies d'exécution. Cependant le gain en temps d'exécution varie d'une stratégie à l'autre, le meilleur ratio étant obtenu par la stratégie AFDHLB.

Le mécanisme d'équilibrage de la charge que nous proposons dans notre travail est semblable à celui de (Felter et al., 2014) du point de vue de la prise en considération du temps d'exécution de l'application sur chaque nœud. Cependant, nous ne nous contentons pas de classer les nœuds par priorité, mais nous quantifions la charge à envoyer à chaque nœud.

2.7 Conclusion

Nous avons abordé dans ce chapitre les articles de recherche concernant les clusters dans un premier temps. Nous avons vu l'utilité des clusters Beowulf et leurs différentes utilisations surtout ceux composés d'ordinateurs mono-carte ARM. En effet, les études ont montré que ces appareils consomment peu d'énergie comparée à d'autres architectures(Ou et al.,

2012),cependant, la configuration utilisée était homogène (un seul type de carte) et le potentiel de ces cartes dans le calcul hautement parallèle n'a pas été étudié.

Afin d'exploiter le mieux le potentiel des cartes ARM, nous nous sommes fixé comme objectif d'utiliser le meilleur moyen de passage de messages possible. Nous avons donc abordé dans la deuxième partie de ce chapitre les études ayant effectué une comparaison entre les deux mécanismes utilisés pour passer les messages au sein d'un cluster, à savoir MPI et PVM. Nous avons noté qu'il existe des différences quant à ces comparaisons, certaines en contredisent d'autres et il ne semble pas y avoir de consensus quant au meilleur mécanisme de passage de messages. D'où notre décision d'implémenter le cluster avec les deux approches. Nous avons aussi présenté un mécanisme d'équilibrage de la charge au sein d'un cluster, cependant ce dernier permet de classer les nœuds suivant un ordre de priorité, mais ne quantifie pas la charge à envoyer à chacun. C'est ce que nous proposons (quantifier la charge pour chaque nœud) afin de tirer le meilleur de notre cluster.

De plus, nous avons vu que de nombreuses études s'intéressent au déchargement des traitements de sécurité dans certains cas où les ressources locales sont restreintes. D'où notre motivation d'évaluer les performances des appareils ARM dans le contexte des traitements de sécurité.

La dernière partie concerne la virtualisation où nous avons vu les études de (Morabito et al., 2015)et (Felter et al., 2014)qui ont démontré que la virtualisation avec les containers est plus performante que celle utilisant les hyperviseurs. D'où notre choix de tester Docker sur les cartes ARM.

CHAPITRE 3

ARCHITECTURE

3.1 Introduction

Nous présentons dans ce chapitre l'architecture du cluster dans un premier temps, le rôle et la disposition des nœuds le composant ainsi que la procédure suivie pour le mettre en place

La deuxième partie concernera le module de monitorage implémenté avec une description des métriques et autres paramètres. Le monitorage sera logiquement distribué selon l'architecture du cluster c'est-à-dire maître-esclave. Nous détaillons aussi le mécanisme d'équilibrage de la charge entre les différents nœuds du cluster.

Par la suite, nous présenterons les algorithmes implémentés. Il s'agit de l'algorithme de correspondance des patrons, AES séquentiel et le AES parallèle.

3.2 Architecture du cluster

Le cluster qui a été réalisé dans ce travail est de type Beowulf (voir chapitre 1). L'architecture sera la même pour les différents scénarios des expérimentations que nous avons prévus, à savoir un nœud master qui est responsable du partage des tâches entre les autres nœuds du cluster et qui récoltera les données du monitorage en les stockant dans une base de données locale.

La communication entre les composants du cluster se fait à travers ssh, qui doit être installé sur tous les nœuds et une connexion sans authentification a été configurée afin de permettre à PVM et MPI de communiquer avec les différents nœuds.

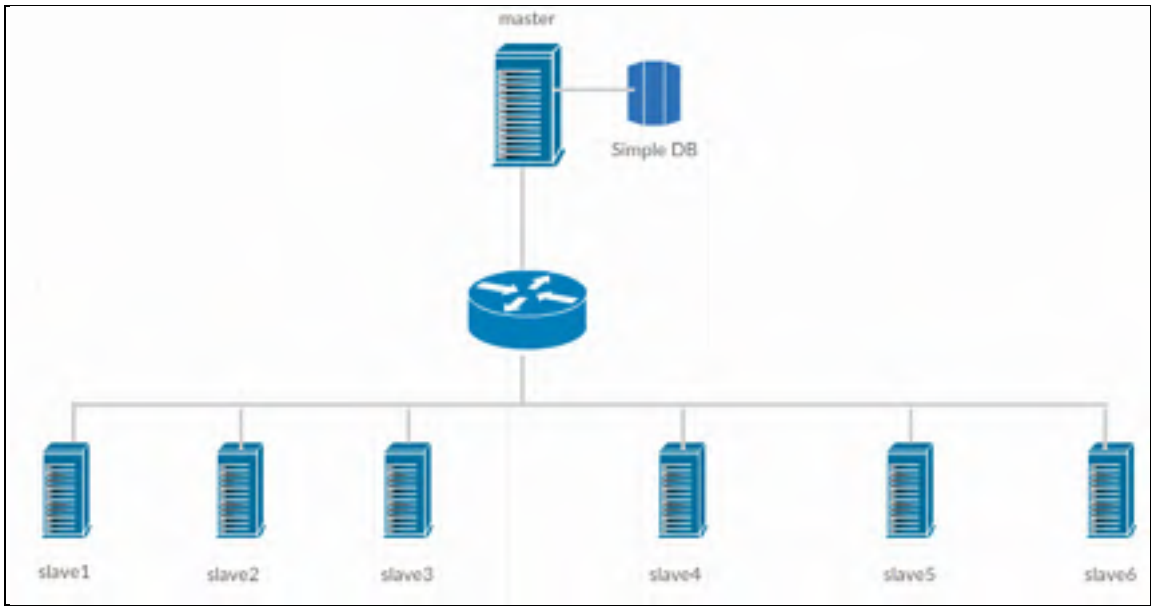

Figure 3.1Architecture du cluster

Nous détaillons ici les étapes nécessaires afin de configurer les connexions entre le nœud master et les slaves :

a) Sur le nœud master:
   - éditer le fichier /etc/hosts et ajouter les adresses IP des nœuds qui vont composer le cluster (e.g 10.180.121.50 slave1)
   - Afin que MPI et PVM puissent fonctionner, il faut lui permettre de se connecter aux différents nœuds à travers ssh et sans mot de passe. Pour cela, nous devons configurer ssh pour qu'il ne demande pas de mot de passe lorsque le noeud maître voudra se connecter aux noeuds esclaves.
     - générer une clé : $ssh-keygen -t dsa
     - Copier la clé: $cp /home/mpiuser/.ssh/id_dsa.pub /home/mpiuser/.ssh/authorized_keys
     - Copier la clé sur les noeuds slaves (Répéter la commande pour chaque noeud):$ scp /home/mpiuser/.ssh/id_dsa.pub mpiuser@slave1:.ssh/authorized_keys

- Ajuster les permissions: $ chmod 600 /home/mpiuser/.ssh/authorized_keys
  $ chmod 700 /home/mpiuser/.ssh/

b) Sur les noeuds esclaves:
   Nous avons besoin que le master puisse se connecter en tant que root sur les différents noeuds, pour cela nous modifions le fichier /etc/ssh/sshd_config et remplacer la ligne : PermitRootLogin without-password par PermitRootLogin yes.
   Finalement, redémarrer le service ssh avec la commande :$sudo service ssh restart .

3.3 Module de monitoring

Pour implémenter ce module, nous avons choisi de nous baser sur l'outil *collectl* qui est open source. Cet outil permet de monitorer un grand nombre de ressources que ce soit sur un seul nœud ou à travers tout le cluster. Notre choix ici est motivé principalement par la "légèreté" de ce logiciel relativement au nombre de métriques qu'il arrive à surveiller (highscalability, 2008) et dont l'overhead est inférieur à 0.1%.

Entre autres, collectl offre les avantages suivants :

- La possibilité de s'exécuter en tant que service en arrière-plan ou en temps réel.
- La possibilité d'afficher les sorties sous plusieurs formats.
- Intégration des métriques surveillées par top, ps, vmstat et iotop.
- Possibilité de sauvegarder les métriques surveillées.
- La possibilité d'exporter les données récoltées sous plusieurs formats, ce qui est très intéressant afin de l'intégrer dans notre module de monitoring.

3.3.1 Métriques

Comme mentionné ci-dessus, collectl offre une large variété de ressources à surveiller. En effet, il offre la surveillance de 13 ressources : CPU, fragmentation mémoire, Disque, NFS V3 Data, inode, interruptions, lustre, mémoire, réseau, sockets, tcp, interconnexion, caches

des objets systèmes. En plus, collectl permet de détailler encore plus chacune de ces métriques, une version détaillée de la surveillance de la CPU donnerait par exemple ce qui suit (figure 3.2) :

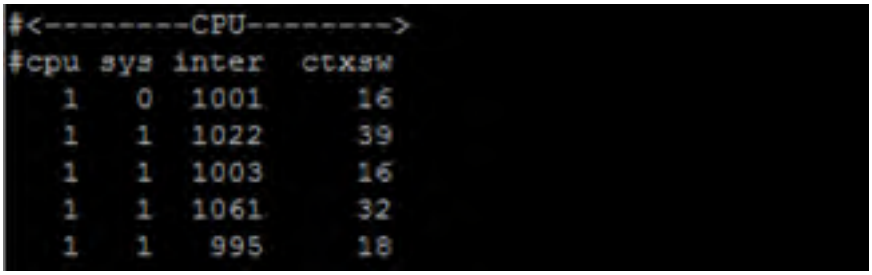

Figure 3.2 Mode détaillé de la surveillance de la CPU.

La première colonne "cpu" représente le pourcentage d'occupation de la cpu, la seconde colonne "sys" représente le pourcentage de temps où le processeur a exécuté des instructions en mode système. La colonne "inter" représente le nombre d'interruptions par seconde et "ctxsw" représente le nombre de changement de contexte par seconde.

Surveiller toutes ces variables ne serait pas efficace et serait même contre-productif, puisque notre but est d'avoir une solution légère avec le moins d'overhead possible afin de la déployer sur les cartes qui ont des ressources relativement limitées.

Il nous est alors important de bien choisir les métriques que nous allons stocker. Ainsi, nous avons choisi de traiter les données directement liées à la performance de notre cluster comme le suggère l'article de Jeff Layton paru dans le magazine consacré au HPC (Layton, 2016)à savoir:

- CPU : Nous nous intéressons à la charge CPU de chaque nœud du cluster. Nous implémentons la possibilité d'enregistrer à des intervalles réguliers que l'utilisateur pourra modifier à sa guise. Outre le fait d'enregistrer ces valeurs chaque x secondes, le système de monitorage calcule la moyenne de toutes les valeurs et nous donne ainsi l'utilisation moyenne de CPU pour chaque nœud.

- Mémoire : Une autre ressource importante pour la surveillance est la quantité de mémoire disponible par nœud. Comme pour la CPU, notre système stock les données à des intervalles réguliers et affiche les données en temps réel ou la moyenne de la mémoire disponible pour chaque nœud.
- Réseau : La raison pour laquelle nous avons choisi cette métrique est qu'un afflux trop important sur un nœud pourrait compromettre la performance globale du cluster et il serait utile d'avoir une idée sur la charge du trafic réseau sur chaque nœud.

3.3.2 Architecture et implémentation

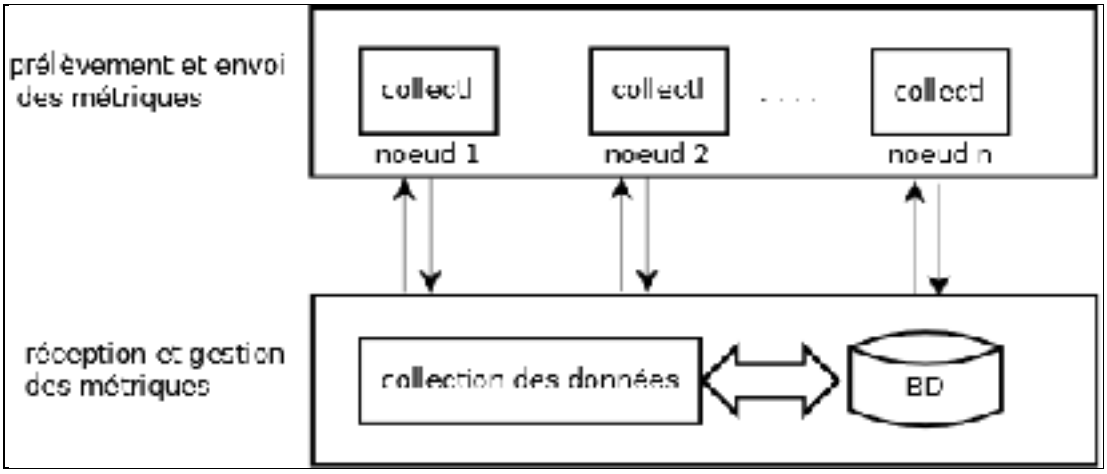

Figure 3.3 Architecture du système de surveillance.

Notre système est basé sur une architecture deux-tiers comme le montre la figure 3.3 et est composé de deux modules principaux qui sont le module de prélèvement et envoi des métriques et le module de réception et gestion des métriques.

3.3.2.1 Prélèvement et envoi des métriques

Ce module est présent sur chaque nœud composant le cluster et consiste en un service collectl qui tourne en arrière-plan et qui enregistre les données de l'utilisation CPU, mémoire et réseau dans un fichier. Ensuite, à la demande du nœud maitre, un processus lit le fichier et

envoie les données demandées au second module. Ce schéma est représenté dans la figure 3.4.

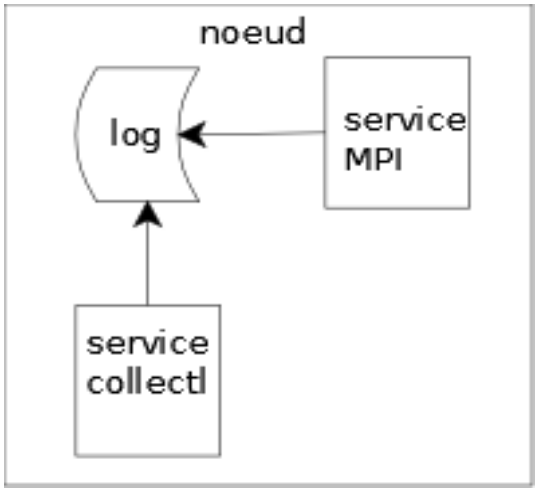

Figure 3.4 Collecte des données au niveau d'un nœud.

3.3.2.2 Réception et gestion des métriques

Ce module est présent dans le nœud maitre et est composé d'un programme MPI qui permet de contacter les nœuds du cluster et demander l'envoi des ressources surveillées, ainsi que d'une base de données permettant de stocker ces données pour consultation ultérieure. Par souci d'optimisation, la base de données ne contiendra que les moyennes d'utilisation. Le processus de transfert des données est optimisé en utilisant une seule fonction MPI_GATHER (voir figure 1.6) de l'API MPI qui permet d'assembler les données de plusieurs nœuds en une seule structure. En effet, cette fonction prend comme paramètres les valeurs des métriques de chaque nœud et les assemble dans une seule structure au nœud maître.

3.3.3 Visualisation temps-réel

Outre le fait d'archiver les données de surveillance, il est possible de les visualiser en temps réel, et ceci grâce à un outil de la suite collectl : colmux. Ce dernier communique directement avec le service collectl pour afficher les variables surveillées en temps réel de

façon centralisée sur le nœud maitre. L'utilisation de ce dernier se fait par une simple commande où on donne comme paramètres les métriques à afficher ainsi que les nœuds.

3.4 Équilibrage de la charge

Dans l'architecture du cluster que nous avons proposé, un nœud (maître) est responsable de la distribution des tâches aux autres nœuds (esclaves). Souvent, les éléments composant le cluster n'ont pas la même capacité de calcul, quelques-uns peuvent être plus chargés que d'autres... imposer alors la même charge de calcul ou les mêmes tâches à tous les nœuds ne serait pas optimal pour les performances globales du cluster, puisque souvent, ce dernier est limité par ses ressources disponibles.

Dans le cas d'un gros fichier en entrée que le nœud maître doit distribuer aux nœuds esclaves, la répartition des portions pour chaque nœud est importante. Comment équilibrer alors la charge entre les composants du cluster?

Quantifier les ressources utilisées par tel ou tel programme peut être un processus fastidieux et pas toujours précis. La quantité de paramètres à prendre en compte est relativement importante : CPU, mémoire, disque, réseau... et diffère surtout d'un programme à un autre. En effet, un programme pourrait avoir besoin principalement de CPU, un autre est gourmand en mémoire, un troisième pourrait nécessiter des coprocesseurs (l'algorithme de correspondance des patrons).

Pour cela, la métrique que nous avons choisie pour cette étude est le temps d'exécution qui est commun à tous les programmes. En effet, nous pouvons mesurer la capacité d'un nœud à exécuter un programme en mesurant son temps d'exécution sur ce dit nœud. L'idée est que plus le temps d'exécution est bas, plus ce nœud est apte à recevoir de la charge et inversement.

La solution que nous proposons dans cette étude distribue la charge de manière proportionnelle au temps d'exécution. Prenons un programme P et deux nœuds A et B. P s'exécute en 3 secondes sur A et en 6 secondes sur B. P s'exécute 50% plus vite sur le nœud A que sur B. A aura alors 50% plus de charges que B lorsqu'on distribuera les entrées entre

les deux. Cette approche est semblable à celle utilisée par (Felter et al., 2014)sauf que nous quantifions la charge à distribuer, alors que (Felter et al., 2014)se contentent de classer les nœuds par priorité.

Soit T_i le temps d'exécution d'un programme P sur le nœud i et n le nombre total de nœuds, nous introduisons une variable nommée coefficient d'exécution décrite par l'équation suivante (3.1) :

$$coef(i) = \frac{1}{n-1}\left(1 - \frac{\mathrm{Ti}}{\sum_1^n \mathrm{Tj}}\right) \qquad (3.1)$$

Nous distribuons ainsi la charge en fonction du coefficient d'exécution de chaque nœud. Soit D la taille totale des données, la portion à envoyer à chaque nœud est alors (équation 3.2):

$$portion(i) = D\,.coef(i) \qquad (3.2)$$

Ainsi, nous faisons en sorte que chaque nœud reçoit la charge adéquate à sa capacité d'exécution.

3.5 Algorithmes

Afin de tester les performances de notre cluster en matière de sécurité informatique, nous avons choisi d'implémenter différents algorithmes nécessitant une puissance de calcul conséquente. Le premier est un algorithme de chiffrement symétrique bien connu, le AES (Advanced Encryption Standard), le second est un algorithme de correspondance des patrons exclusif aux cartes parallella puisqu'il tire avantage de l'architecture epiphany pour effectuer des traitements parallèles.

3.5.1 AES

Cet algorithme de chiffrement a été proposé lors du concours AES, qu'il a remporté en 2000. Il est depuis utilisé par le gouvernement des états unis et est le système de chiffrement le plus utilisé au monde(Rouse, 2016).

Nous avons testé l'implémentation de deux variantes de cet algorithme, à savoir une version séquentielle, et une version parallèle.

3.5.1.1 AES séquentiel

AES est un algorithme de chiffrement symétrique, c'est-à-dire qu'il va prendre comme entrée un message clair P, le chiffrer avec une clé secrète K et donner en sortie un message crypté C. L'inverse est tout aussi possible : déchiffrer le message crypté avec la clé K.

Pour AES, le chiffrement se fait par bloc de taille 128 bits et une clé multiple de 32 et comprise entre 128 et 256. Il repose sur des algorithmes itératifs ou fonction de tours qui sont répétés t fois (t dépendant de la taille de la clé). À chaque itération une sous-clé ou clé de tour est utilisée, cette dernière est générée à partir de la clé secrète K. Une itération consiste en une opération de substitution, une opération de décalage et finalement une opération de mélange. Cette opération est schématisée dans la figure 3.5.

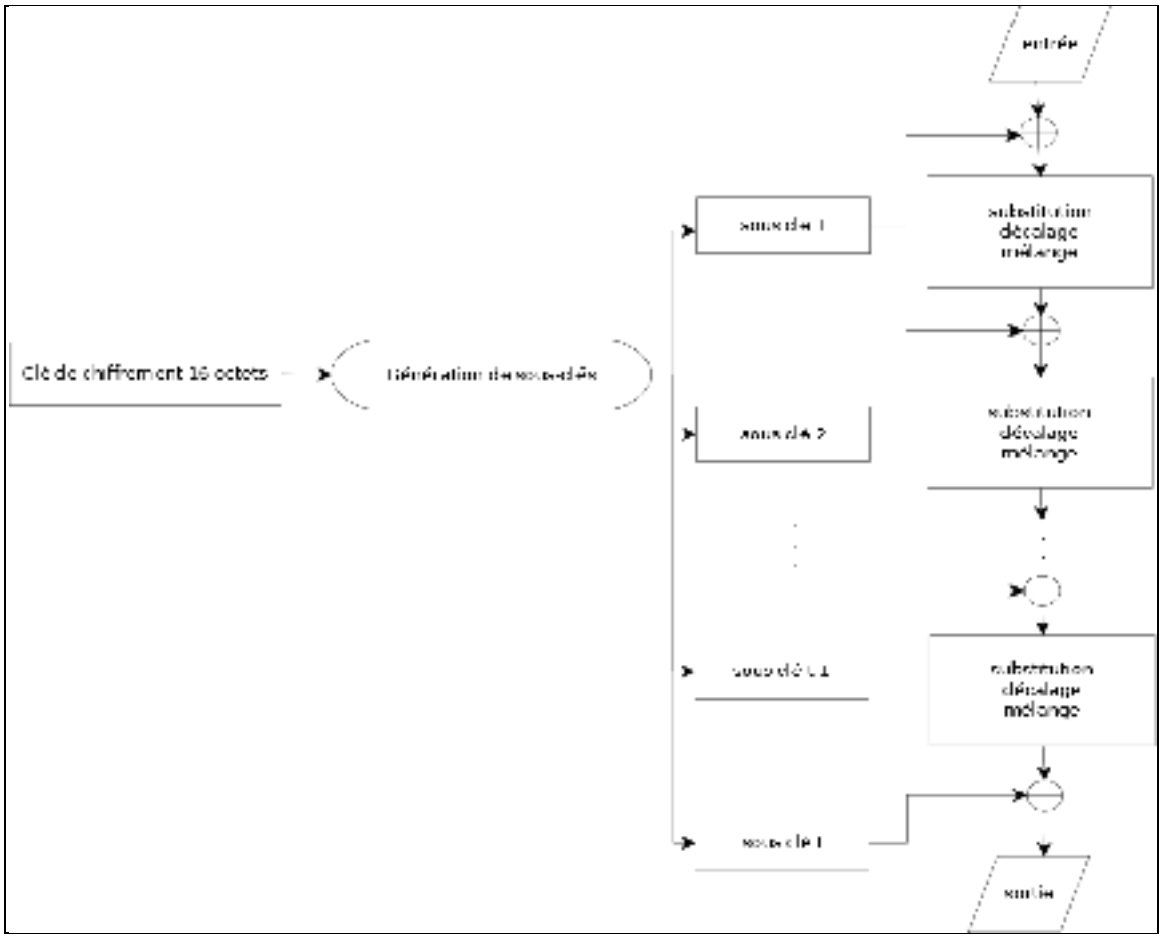

Figure 3.5 Chiffrement d'un bloc avec AES

L'opération décrite dans la figure 20 est effectuée pour chaque bloc composant le message original, ce dernier étant décomposé en des blocs de 128 bits. Ce bloc a la forme d'une matrice 4x4 où chaque case contient un octet de données. Ainsi les opérations de substitution, de décalage et de mélange s'effectueront sur cette matrice. De plus, à chaque itération, une nouvelle sous clé, générée à partir de la clé originale est utilisée.

- Substitution : La substitution s'effectue en appliquant la fonction S à chaque élément de la matrice d'entrée, ainsi pour tout $1 \leq i \leq 16$, $Y_i = S(X_i)$ où X est la matrice d'entrée, Y la matrice de sortie après substitution.
- Décalage : cette opération consiste en un décalage circulaire de i cases pour la ligne numéro i.

- Mélange : le mélange consiste en une opération de multiplication dans l'espace GF(2^8) qui est un corps fini. Chaque colonne est multipliée par la matrice 4x4 prédéfinie dans GF(2^8).

$$\begin{matrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{matrix} \text{ x } \begin{matrix} \text{X1} \\ X5 \\ X9 \\ X13 \end{matrix} = \begin{matrix} Y1 \\ Y5 \\ Y9 \\ Y13 \end{matrix} \qquad (3.3)$$

- Addition de la clé : La dernière étape consiste à additionner chaque élément X_i en entrée, avec l'élément K_i de la sous clé K.

3.5.1.2 AES parallèle

AES étant devenu de plus en plus populaire, il a remplacé DES pour le chiffrement de nombreuses applications(Le et al., 2010). Cependant, de plus en plus d'applications demandent un chiffrement rapide et la version séquentielle d'AES s'exécutant sur la CPU montre des performances en deca des attentes de ces applications.

C'est dans cet esprit que la version parallèle a été pensée, afin d'accélérer le chiffrement/déchiffrement et répondre aux exigences du marché. Comme présenté plus haut, l'algorithme AES classique (séquentiel), itère des opérations un certain nombre de fois. L'idée est alors de paralléliser ces itérations sur chaque bloc pour qu'elles soient effectuées en parallèle comme le montre la figure 3.6. En effet, chiffrer les blocs en parallèle constitue un gain important en matière de vitesse d'exécution.

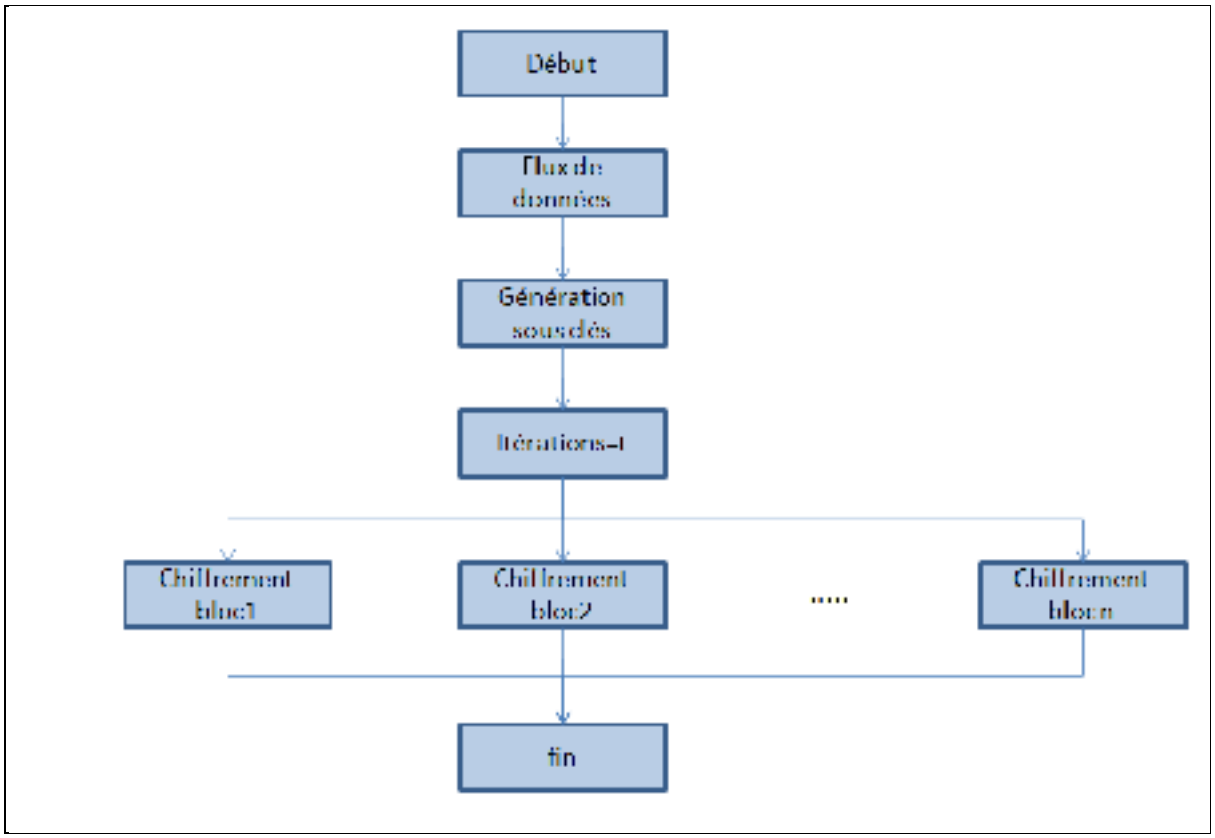

Figure 3.6 Diagramme PAES.

Le chiffrement d'un bloc est l'opération décrite dans la figure 3.5, à chaque thread est assigné le calcul relatif à un bloc. Initialement, cette architecture a été pensée pour des systèmes offrant une grande capacité de calcul parallèle telle que la GPU. En effet, cette dernière a été conçue pour traiter des calculs de hautes performances avec des contraintes temps-réel strict. Une mémoire globale partagée permet de stocker la clé de chiffrement ainsi que le texte à chiffrer. Ces derniers seront accessibles pour tous les threads. La figure 22 est un schéma simplifiée de la parallélisation de AES à l'aide de la GPU(Le et al., 2010).

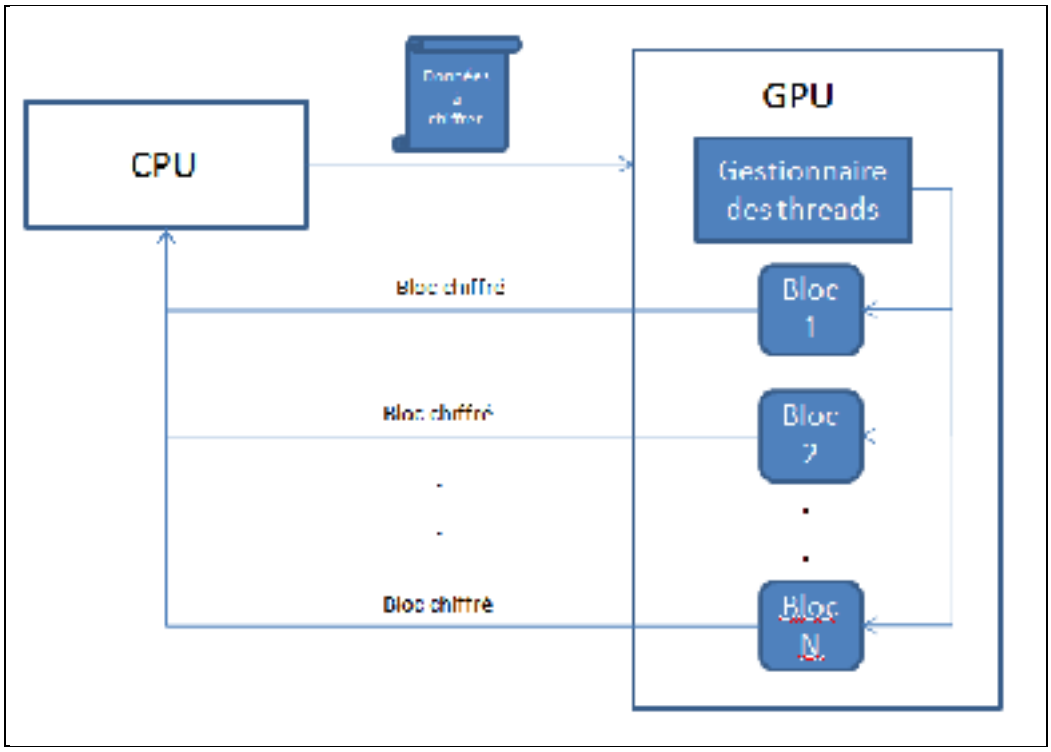

Figure 3.7 AES parallèle sur GPU

Dans l'architecture de la figure 3.7, la CPU s'occupe de transmettre le texte à chiffrer avec les données tels que la clé de chiffrement à la GPU. Cette dernière, s'occupe de distribuer les tâches à un nombre de threads, chacun s'occupant d'un bloc.Il est vrai que l'architecture a été initialement proposée pour la GPU, mais elle pourrait être implémentée sur d'autres plateformes proposant une approche parallèle et fonctionnant par thread. Dans le cadre de cette étude, l'implémentation a été effectuée sur les cartes Parallella en tirant profit de l'architecture Epiphany des coprocesseurs qu'elles offrent.

3.5.2 Correspondance de patrons

Dans le cadre d'un travail visant à accélérer la détection de malware sur les téléphones mobiles de nouvelle génération, un algorithme de correspondance de patrons utilisant la GPU a été proposé par(Abdellatif et al., 2015). Les auteurs ont par la suite adapté cet algorithme aux cartes Parallella en substituant l'utilisation de la GPU par celle du coprocesseur Epiphany.

Nous visons dans ce travail à évaluer les capacités des cartes Parallella à accélérer encore plus l'algorithme de correspondance des patrons en l'adaptant au cluster.

Nous donnons dans ce qui suit une brève description de l'architecture dont l'algorithme de correspondance des patrons est une partie.

L'architecture de la solution de détection de malware dont fait partie la correspondance de patron est composée de trois parties : une partie de prétraitement, une partie de traitement, et un module de cryptographie(Abdellatif et al., 2015).

Le prétraitement (voir figure 3.8) consiste en la construction d'une base de données à laquelle va se référer par la suite l'algorithme de détection de malwares. Cette base de données va contenir des patrons d'appels systèmes issus du profil d'exécution de plusieurs logiciels malveillants. Par la suite, ces patrons sont utilisés afin de construire un automate déterministe fini. Ce dernier servira dans la partie de traitement parallèle afin de détecter éventuellement la présence d'un malware en analysant les traces d'exécution. L'avantage de l'utilisation de la structure d'automate est le fait qu'il est possible d'analyser les traces reçues en entrée en une seule passe d'exécution.

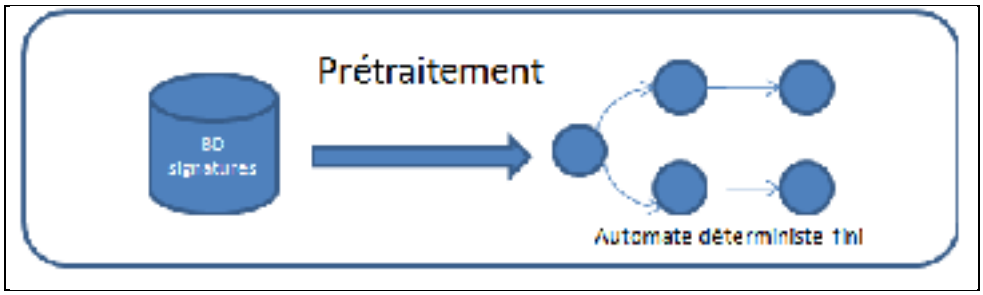

Figure 3.8 Processus de prétraitement du module de détection des malware

La partie du traitement parallèle est la partie qui nous intéresse dans ce travail et laquelle nous allons implémenter sur un cluster. Il s'agit comme décrit précédemment d'analyser les traces d'exécutions sous la forme d'appels systèmes afin de vérifier la présence ou non d'une séquence d'exécution d'un programme malveillant.

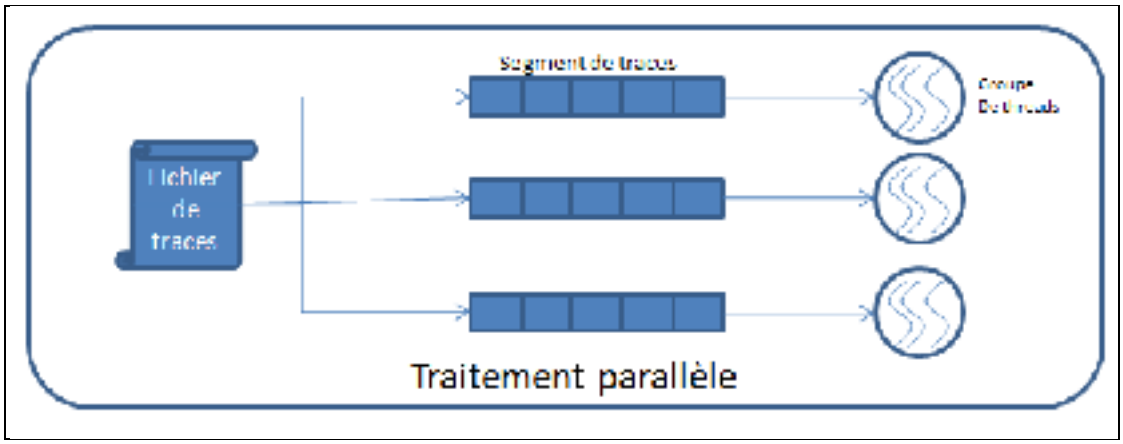

Figure 3.9 Bloc de traitement parallèle dans la GPU.

Le traitement (voir figure 3.9) de cette partie est essentiellement effectué par la GPU. En effet, la CPU a seulement la tâche de transmettre les données en entrée et c'est la GPU qui s'occupe du scan en divisant les données en plusieurs segments qui seront analysés en parallèle par des threads. Chaque groupe de threads analyse l'un des segments. L'analyse se fait par un algorithme de correspondance de patrons nommé *Parallel Failurless Ahoc-Corasick.* Ce dernier prend en entrée les patrons des appels systèmes malicieux sous la forme d'un automate et les compare aux traces d'exécutions à analyser. Les résultats sont par la suite envoyés à la CPU.

CHAPITRE 4

EXPÉRIMENTATIONS ET RÉSULTATS

4.1 Introduction

Dans ce chapitre, nous présentons les résultats des expérimentations effectuées sur l'implémentation des différents modules que nous avons détaillés dans le chapitre précédent.

Afin d'arriver à mettre en place le cluster et de déployer la virtualisation sur les cartes, nous avons dû relever nombreux défis techniques liés à l'environnement logiciel mais aussi matériel. Nous présentons alors à la deuxième partie de ce chapitre, les difficultés liées à l'utilisation des cartes ARM.

Nous évaluons les performances du cluster que nous avons mis en place avec différentes dispositions. Plusieurs scénarios de test ont été réalisés afin d'une part choisir la meilleure alternative en termes de mécanisme de passage de message et d'autre part mesurer l'impact de l'équilibrage de la charge sur les algorithmes implémentés.

Nous commençons par tester la performance d'un cluster hétérogène composé de plusieurs types de cartes. Nous testons différentes manières de partager les tâches entre les nœuds du cluster et nous varions de plus la taille de ce dernier. Par la suite, nous expérimentons avec les algorithmes spécifiques à la carte parallella, puisque ceux-ci nécessitant la GPU et ont été adaptées à l'architecture epiphany de ces cartes et procédons de façon similaire (variation du nombre des nœuds et des répartitions des tâches).Finalement, nous expérimentons le déploiement de Docker sur certaines cartes ARM pour mesurer l'impact de la virtualisation sur les performances de ces derniers.

4.2 Environnement

4.2.1 Matériel utilisé

Plusieurs cartes ont été utilisé pour ce travail, elles offrent des ressources différentes et ont été conçus pour des objectifs différents, mais elles ont en commun leur micro-processeur d'architecture ARM Cortex-A9 (sauf la BeagleBone qui intègre le Cortex-A8).

Notre choix est motivé par le fait que le Cortex-A9 possède le meilleur ratio performance/énergie et qu'il est assez mature puisqu'il est utilisé depuis 2008 dans divers systèmes embarqués (arm, 2015b). Bien que le cortex A15 soit plus rapide, il est relativement nouveau et n'est pas encore autant utilisé que le A9.

4.2.1.1 Parallella

Les cartes Parallella sont construites par la société Adapteva et sont des nano-ordinateurs de petite taille (90 x 55 x 18 mm). Cette carte intègre un processeur de type ARM Cortex-A9 à deux cœurs d'une cadence 1 GHz. La particularité de cet appareil est son coprocesseur Epiphany et ses 16 cœurs qui sont supposés permettre d'augmenter les performances de calculs et sont spécialement destinés au calcul parallèle.

Figure 4.1 Carte Parallella.
Tirée de(adapteva, 2016).

Tableau 4.1Spécifications de la parallella

Processeur	ARM Cortex-A9 1GHz x 2 coeurs Zynq Z70X0
Coprocesseur	Epiphany III 16 coeurs
Mémoire RAM	SDRAM DDR3 1Go
Mémoire flash	Quad-SPI 128 Mbit
Ports	Ethernet, microSD, micro-HDMI, USB
Puissance	5 W
Dimensions	90 x 55 x 18 mm

4.2.1.2 PandaBoard

La PandaBoard est une carte mère possédant un processeur ARM cortex-A9 à double cœur et qui intègre le système sur puce (SoC) OMAP4430 de Texas Instrument. Cette carte est adressée surtout aux développeurs et leur offre une puissance non négligeable avec 1 Go de

mémoire et un processeur double cœur à 1.2 GHz ce qui en fait un bon environnement de test pour l'industrie.

Le grand nombre de ports présents sur la carte (voir tableau 4.2) constitue aussi un atout majeur pour le développement d'applications mobiles, la PandaBoard supporte entre autres l'OpenGL et peut encoder/décoder de la vidéo jusqu'à 1080p.

Tableau 4.2Spécifications de la PandaBoard

Processeur	ARM Cortex-A9 double cœur 1.2 Ghz
Mémoire	1GB DDR2 RAM
Sans Fil	802.11 b/g/n, Bluetooth v2.1
Vidéo	Cœur graphique SGX540 supportant OpenGL
Affichage	Support LCD, HDMI.
Connectivité	Ethernet, 3 ports USB2.0, expansion caméra, MMC
Dimensions et poids	114.3 x 101.6 mm , 81.5 grammes

Figure 4.2 PandaBoard.
Tirée de (isctruments, 2016).

4.2.1.3 Odroid-U3

L'Odroid-U3 (Voir figure 4.3) est un ordinateur monocarte construit par la société coréenne HardKernel. L'avantage de cette carte est qu'elle coute 59$ et possède le même processeur que Samsung s3, avec 2GB de RAM.

Figure 4.3 Odroid-U3.
Tirée de(HardKernel, 2016).

Ainsi, cette carte repose sur SoC de samsung Exynos 5 possédant 4 processeurs ARM Cortex-A7 (LITTLE) et 4 Cortex-A15 (big). À l'instar de la PandaBoard, la odroid offre plusieurs sorties, ce qui en fait une carte assez intéressante pour le développement. Le tableau 4.3 liste les spécifications de la carte Odroid-U3.

Tableau 4.3 Spécifications de la Odroid-U3

Processeur	Exynos quadcore cortex-A9 1.7GHz
Mémoire	2 Go RAM LPDDR3
Vidéo	3 GPU PowerVR SGX544MP3
Connectivité	3 ports USB 2.0 , ethernet, lecteur SDHC, connecteur HDMI, lecteur eMMC
Dimensions	98x74x29 mm

4.2.1.4 BeagleBone

La beaglebone (voir figure 4.4) est un ordinateur mono-carte fabriqué par TexasInstruments et est destinée aux développeurs. Cette carte possède un processeur Cortex-A8 à 1GHzm, un cœur graphique PowerVR SGX 530 et 512 MB de RAM. La beaglebone se vend à 80$. Ses spécificités sont indiquées dans le tableau 4.4.

Figure 4.4 Beaglebone.
Tirée de (instruments, 2014).

Tableau 4.4Spécifications de la beaglebone

Processeur	Sitara AM335x Cortex-A8 1GHz
Mémoire	512 MB RAM DDR3L
Vidéo	PowerVR SGX 530
Connectivité	1 port USB 2.0 , ethernet, lecteur MicroSD, connecteur HDMI, lecteur eMMC
Dimensions	86.36mm x 53.34mm x 4.76mm

4.2.2 Environnement logiciel

L'implémentation des algorithmes décrits dans le chapitre précédent a été effectuée avec le langage C. En effet, PVM et MPI ont en commun le fait de supporter ce langage.

L'implémentation MPI que nous avons choisie est OpenMpi version 1.8.4 qui est développée et maintenue par une large communauté scientifique dans le domaine de l'informatique de haute performance. De plus, cette implémentation est jugée légère et portable, ce que nous recherchons pour le cluster que nous visons à déployer.

Notre second mécanisme de passage de messages est PVM 3.4.6 qui est la dernière version sortie à ce jour. PVM nécessite une installation sur tous les nœuds du cluster.

Pour ce qui est des tests avec la technologie docker, nous avons utilisé un environnement virtuel dont nous détaillerons les éléments lors de la présentation des résultats des expérimentations sur cette partie.

4.3 Défis techniques

Parmi les objectifs que nous nous sommes donnés dans ce travail, c'est l'évaluation des cartes ARM pour leur utilisation dans les traitements de sécurité. Une partie de cette évaluation consiste à déterminer les difficultés techniques liés à l'utilisation de ces systèmes.

En effet, bien qu'ils suscitent une attention grandissantes ces dernières années, les ordinateurs mono-cartes ARM en sont encore à leurs premiers pas comme nous le verrons dans cette partie. Contrairement aux ordinateurs traditionnels, ces cartes ne sont pas destinées pour l'instant au grand public, mais plutôt pour les développeurs et la plupart des produits sont encore en phase de test.

Nous abordons dans ce qui suit les défis notables que nous avons rencontrés afin de mettre en place un cluster ARM et expérimenter la virtualisation sur ces cartes.

4.3.1 Installation des environnements

À la différence des ordinateurs traditionnels ou des téléphones mobiles, les cartes ARM ne viennent pas avec un système d'exploitation installé et prêts à l'emploi.

En effet, la plupart de ces cartes nécessitent l'installation d'un système d'exploitation sur une carte micro-sd. Puisque les cartessont composées de matériels différents, chaque type de carte nécessite une version adaptée à son matériel. Une version d'Ubuntu adaptée pour la Odroid par exemple, ne fonctionnerait pas sur la BeagleBone et vis-versa. Les versions standards de Linux (destinés aux PC) ne fonctionnent évidemment pas sur les cartes ARM.

Certains constructeurs offrent des images précompilées (prebuilt) de systèmes fonctionnant sur leur matériel. C'est le cas pour la compagnie Adapteva qui offre une image prête d'Ubuntu 14.10.

Pour les autres cartes (PandaBoard, BeagleBone et Odroid-U3), nous avons dû compilé nos propres images. Les fabricants fournissent néanmoins la configuration nécessaire du noyau Linux pour faire fonctionner le système d'exploitation sur les cartes sous la forme de code source non compilé.

Afin de construire une image, certains logiciels sont nécessaires. Parmi ces logiciels, nous pouvons citer le U-boot (denx, 2016) qui est un logiciel libre utilisé dans le domaine des systèmes embarqués servant à contenir les instructions nécessaires pour démarrer le noyau du

système d'exploitation. U-boot doit être configuré et compilé pour avoir la version spécifique à chaque carte.

Un autre outil nécessaire est le ToolChain qui consiste en un inter-compilateur (cross-compiler) servant à compiler le noyau Linux destiné à fonctionner sur les systèmes ARM. On parle d'inter-compilation car cette procédure se passe sur un PC (architecture x86) et produit un exécutable destiné à une architecture ARM.

Les étapes servant à construire les images sont donc l'acquisition et la configuration des logiciels que nous venons de citer, le téléchargement et la compilation du noyau linux (cette étape nécessite plusieurs heures) et l'acquisition du système de fichier adéquat (version de Linux). Par la suite, il faut formater manuellement (à l'aide de l'utilitaire fdisk de Linux) la carte micro-sd qui accueillera le système d'exploitation et préparer les partitions.

Cependant, comme nous l'avons mentionné plus haut, les cartes ARM sont relativement récentes et il arrive que des versions sortent sur le marché et soient rapidement abandonnés par le fabricant qui sort une nouvelle version quelques mois après. Ceci engendre des problèmes de compatibilité qui rend la construction des images de systèmes d’exploitation extrêmement délicat. C'est ce qui s'est produit dans notre cas avec la PandaBoard. En effet, la version que nous possédons de cette carte est la Rev-B3 et après compilation du noyau et construction de l'image, celle-ci ne démarrait pas. Nous avions donc un problème de boot. Il s'est avéré que le fabricant n'offrait plus de support pour cette carte (la dernière mise à jour sur son site date de 2013 (isctruments, 2016)et qu'il fallait utiliser une version de U-boot différente que celle suggérée sur le site du fabricant (Katta, 2014).

Ce genre de déconvenue est malheureusement fréquent avec les cartes ARM car c'est un milieu évoluant rapidement et qui n'est pas encore stable comme nous l'expliquerons avec davantage d'exemples au cours de cette partie.

4.3.2 Surchauffe des Parallella

Un autre défi rencontré, mais cette fois de type matériel, c'est le problème de surchauffe des cartes Parallella.

Les premières versions de cette carte souffraient d'un important problème de surchauffe qui faisait en sorte que les cartes s'éteignaient parfois lors d'une utilisation normale. Pour remédier à cela, les fabricants ont inclus un dissipateur de chaleur dans leur nouvelle version qu'il faut installer manuellement (adapteva, 2016).

Cependant, lors de l'utilisation des Parallella nous avons constaté que le problème de surchauffe subsistait lorsque nous exécutons les algorithmes de sécurité sur ces dernières. Les cartes s'éteignaient parfois après une trentaine de minutes d'utilisation, de plus, elles représentaient un risque de brulure puisque nous devions les manipuler manuellement.

Pour remédier à cela, nous avons bricolé un système permettant de dissiper la chaleur des cartes qui consiste à les assembler à l'aide de vis de support et à installer un ventilateur comme montré dans la figure 4.5.

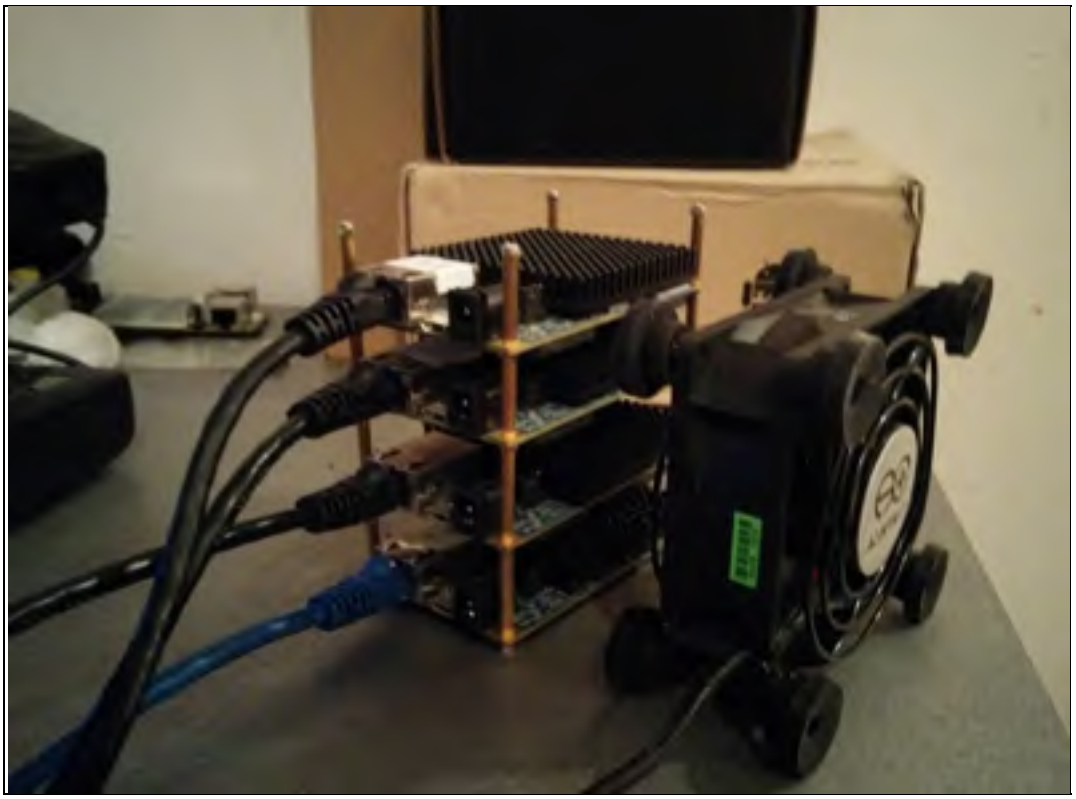

Figure 4.5 Assemblage des cartes Parallella

4.3.3 OMAP5432 et KVM

Afin d'expérimenter KVM, nous avons commandé la OMAP5432 qui est une carte fabriquée par Texas Instrument(instruments, 2016a) et qui possède un processeur ARM Cortex-A15. La particularité de cette architecture est qu'elle est plus rapide que le Cortex-A9 mais surtout qu'elle procure un support pour l'hyperviseur KVM. D'où notre intérêt pour cette carte.

Cependant, dans la version du noyau proposée par le fabricant, la virtualisation n'est pas prise en compte. Pour cela, nous avons dû reconfiguré le noyau et construire notre propre image. L'image résultante contenait un bug qui arrêtait le démarrage de la carte lors du chargement du noyau.

Nous avons contacté l'équipe technique du fabricant (TexasInstruments) mais n'avons pas trouvé de solution au problème. D'ailleurs d'autres utilisateurs de cette carte ont rencontré le même problème (Instruments, 2016b).

Nous n'avons pas vu l'intérêt d'investiguer encore plus ce problème, puisque comme nous le mentionnons dans la revue de littérature, Docker montre des performances supérieures à celles de KVM.

4.3.4 Déploiement de Docker

Souvent, les développeurs de logiciels visent les plateformes grand public en priorité. Cette politique est un inconvénient pour les plateformes ARM qui ne sont pas encore grand public (les ordinateurs mono-cartes et non les Smartphones). Ce fut le cas pour Docker qui est sorti en premier lieu sur les plateformes traditionnels (PC, serveurs, MAC…). Afin de pouvoir utiliser Docker sur les appareils ARM, des modifications doivent être apportées au noyau Linux pour que les modules supportés par ce logiciel soient présents.

Cependant, les fabricants des cartes ARM ne voient pas encore l'intérêt d'intégrer le support de Docker dans les images de leurs noyaux. C'est le cas pour la société Adapteva (fabricant de Parallella) que nous avons contacté pour les interroger sur l'éventualité d'intégrer Docker dans les prochaines versions de leurs images. Ils ont répondu que ce n'est pas dans leur projet

et que compte tenu du coprocesseur Epiphany présent sur la carte, l'isolation serait impossible. En effet, le coprocesseur a un accès direct à la mémoire physique et ne peut exécuter qu'un processus à la fois.

Nous avons alors investigué des moyens de faire fonctionner Docker sur les autres cartes. Nous donnons plus de détails sur la procédure dans la partie sur les résultats de la virtualisation de ce chapitre.

4.3.5 OpenMPI et Parallella

Afin de pouvoir accéder au coprocesseur Epiphany, un programme a besoin de s'exécuter en tant que super utilisateur. C'est le cas pour l'algorithme de correspondance des patrons et le PAES. Cependant, OpenMPI ne permet pas l'exécution en tant que root, du moins l'installation par défaut ne le permet pas.

Afin d'exécuter un programme avec les droits root, il faut utiliser un flag " --allow-run-as-root", cependant cela ne permet l'exécution que localement car les processus OpenMPI distants ne possèdent pas les privilèges root.

Pour remédier à cela, nous avons cherché dans le fichier de configuration et essayé plusieurs options pour trouver celle qui permettrait d'exécuter des programmes OpenMPI en tant que root sur tous les nœuds. Il a fallu alors recompilé les fichiers sources de OpenMPI sur toutes les cartes. Lors de la configuration, il est nécessaire d'ajouter l'option "-enable-orterun-prefix-by-default".

4.4 AES

Nous commençons par présenter les résultats de l'implémentation de AES sur les cartes ARM. Nous aurons comme environnement alors un cluster hétérogène composé de cartes Parallella, Beaglebone, PandaBoard.

Nous décomposons les tests en deux scénarios, le premier consiste en une distribution naïve de la charge, le second en distribuant la charge avec notre solution d'équilibrage.

4.4.1 Distribution naïve

Le fichier d'entrée est distribué de manière égale entre les nœuds. Nous faisons varier le nombre des nœuds composant le cluster, mais aussi le nombre de processus exécutant l'algorithme. En effet, les cartes possédant pour la plupart un double processeur, il est possible d'exécuter le chiffrement en un seul ou deux processus sur la même carte. Les tests sont effectués à l'aide de MPI et de PVM dans le but de comparer les deux outils. Le tableau 4.5 montre les résultats des tests.

Tableau 4.5 Temps d'exécution en secondes selon le nombre de processeurs.

Nombre de nœuds	Nombre de processus	MPI	PVM
1	1	24	24
	2	16.32	18.7
2	2	12.55	13.5
	3	8.43	11.2
	4	7.8	9.1
3	3	8.95	9.6
	4	6.5	8.8
	5	5.43	8.2
	6	5.3	6.7
4	4	6.4	7
	5	5.2	6.3
	6	4.6	5.4
	7	4.3	5
	8	3.7	4.6
5	5	5.3	6
	6	4.6	5.4
	7	4	5.1
	8	3.6	4.9
	9	3.3	4.5
	10	2.9	4.3
6	6	4.4	5
	7	4	5.3
	8	3.4	5.1
	9	3.3	4.4
	10	2.9	4.2
	11	2.9	4.1
	12	2.9	4.2

À l'analyse du tableau 4.5, le plus évident est que MPI est relativement plus performant que PVM dans toutes les configurations d'exécution. D'autre part, nous remarquons qu'exécuter l'algorithme sur un nombre n de processus varie suivant le nombre d'hôtes composant le cluster. Prenons comme exemple le cas où n' est égale à 2: l'exécution sur un seul hôte s'est faite en 16.32s en utilisant MPI et 18.7s en utilisant PVM, alors que sur deux hôtes, on passe à 12.55s et 13.5s respectivement. Ceci est prévisible et s'explique par le fait que les deux processus s'exécutant sur des hôtes différents, il n'y aura pas d'overhead lié à la concurrence pour les entrées/sorties et à la taille du cache au niveau du processeur. Ce qui n'est pas le cas lorsque les deux processus sont exécutés sur le même hôte.

Aussi, d'après le tableau, nous constatons que la configuration donnant les temps d'exécution les plus optimaux est celle où deux processus sont exécutés par nœud. Le graphe de la figure 4.6 décrit le temps d'exécution en fonction du nombre de nœuds dans le cas où le nombre de processus est deux fois celui des nœuds. Il n'est pas possible d'accorder plus de processeurs par carte. La différence en terme de performance entre PVM et MPI est aussi démontrée.

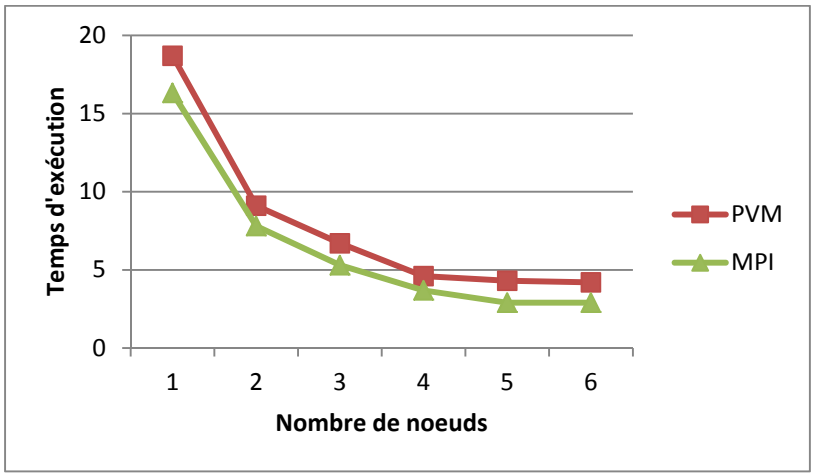

Figure 4.6 Temps d'exécution en fonction du nombre de nœuds.

Nous constatons aussi d'après ce graphe, qu'à partir de 5 nœuds composant le cluster, il n'y a plus de gain en terme de temps d'exécution que ce soit pour MPI ou PVM.

4.4.2 Débit d'exécution

Les tests réalisés dans la partie précédente ont été réalisés avec un fichier d'entrée de taille fixe égale à 3MB. Nous nous intéressons dans ce scénario à varier la taille des données en entrée afin de mesurer l'impact sur le débit d'exécution. Pour cela, nous nous appuyons sur les résultats des tests du tableau 4.7 pour effectuer de nouvelles expérimentations en faisant varier la taille des entrées comme montrées dans le graphique de la figure 4.6. La distribution des données se fait de manière naïve ici aussi (même taille pour chaque nœud).

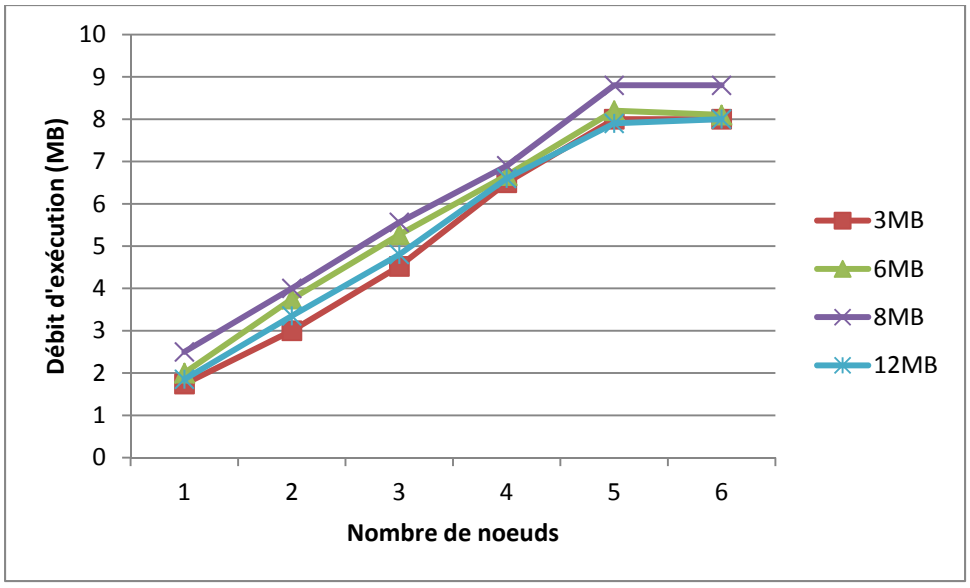

Figure 4.7 Débit d'exécution en fonction du nombre de nœuds et de la taille de l'entrée.

Nous constatons que lorsque la taille des données en entrée est égale à 8MB, le cluster donne le meilleur débit d'exécution quel que soit le nombre des nœuds.

4.4.3 Équilibrage de la charge

Dans ce scénario, nous testons l'efficacité de notre algorithme d'équilibrage de la charge. Nous prenons la configuration du cluster qui a donné le meilleur résultat, c'est à dire avec 6 nœuds et 12 processus et refaisons les tests, mais en chargeant les nœuds différemment. C'est

à dire que nous allons exécuter un autre algorithme en parallèle qui va consommer des ressources et ainsi nous augmentons intentionnellement le temps requis par un nœud pour exécuter l'algorithme de correspondance des patrons. L'algorithme exécuté en parallèle pour charger la carte est une simple multiplication de matrice qui consommera assez de ressources pour impacter les performances de chaque nœud. Les résultats de cette expérimentation sont présentés dans la figure 4.8.

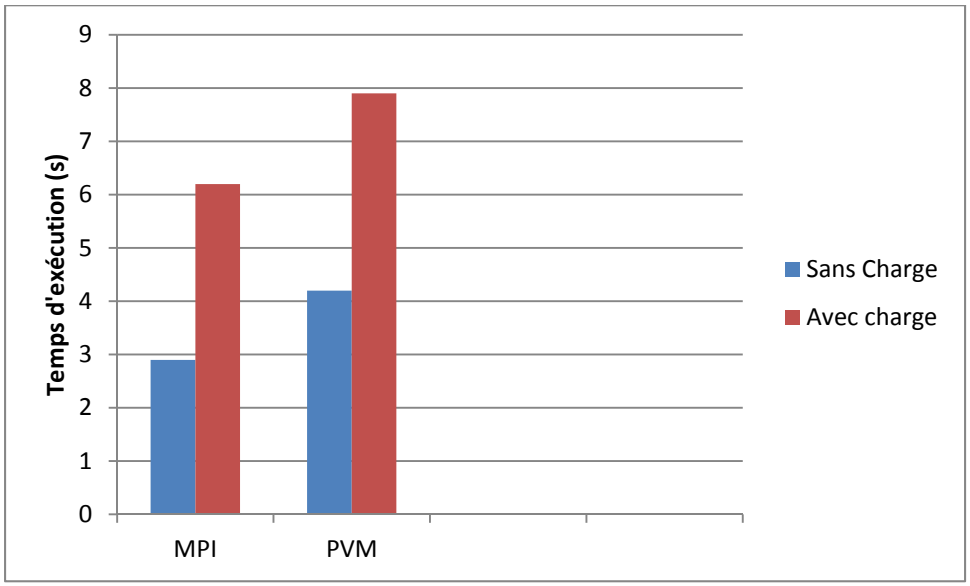

Figure 4.8 Effet de la charge supplémentaire sur le temps d'exécution.

Charger les nœuds a conduit à augmenter le temps d'exécution total comme prévu. Nous testons l'impact de la charge sur le temps d'exécution individuel de chaque nœud afin de récolter les données nécessaires à l'algorithme d'équilibrage de la charge.

Nous avons effectué les tests d'exécution en prenant les mêmes paramètres pour chaque carte : taille de l'entrée de 3 MB et une clé de 128 bits. Le tableau 4.6 montre les résultats obtenus.

Tableau 4.6 Coefficient d'exécution selon le type de la carte.

Type de carte	**Temps d'exécution(s)**	**Coefficient d'exécution**
Parallella	25	0.387
BeagleBone	45	0.295
PandaBoard	40	0.318

Suivant les temps T_j d'exécution individuels, nous décidons d'accorder plus de données en entrée aux cartes de types parallella, suivie de la PandaBoard et finalement la BeagleBone. La troisième colonne représente le coefficient d'exécution de chaque type de carte qui est égal à $coef(i) = \frac{1}{n-1}(1 - \frac{\mathrm{Ti}}{\sum_1^n \mathrm{Tj}})$, n étant le nombre de cartes différentes.

La redistribution de la charge de l'entrée entre les nœuds suivant les coefficients d'exécution donne les résultats de la figure 4.9.

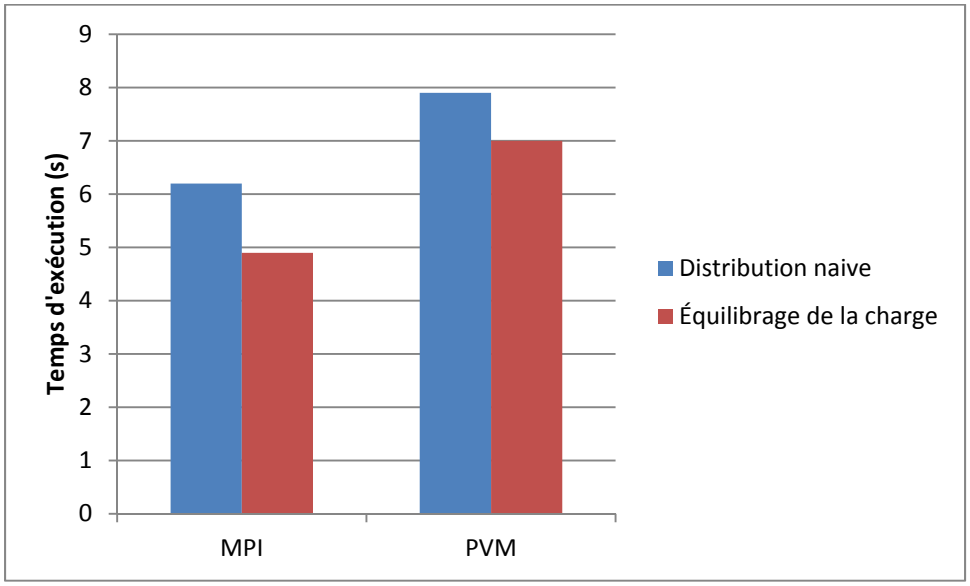

Figure 4.9Effet de l'équilibrage de la charge.

Nous constatons d'après la figure 4.8 une diminution du temps d'exécution pour les deux outils de passage de messages. L'augmentation de la performance pour MPI est de 21% et elle est de 11% pour PVM. En effet, sans équilibrage le temps d'exécution est de 6,2s avec MPI et 7.9s avec PVM, avec l'équilibrage ces valeurs passent à 4.9s et 7s respectivement comme montré dans la figure 4.8. Cette différence dans le taux de gain en termes de temps d'exécution est due aux mécanismes de transfert de messages de PVM et MPI. En effet, MPI propose une fonction interne scatterv(), qui permet de diviser et d'envoyer d'une manière optimale un message de longueur l entre les différents processus. En plus, cette fonction prend comme paramètre un tableau contenant la taille de la portion à envoyer à chaque processus. La division des entrées est alors faite de manière optimale. PVM quant à lui ne propose pas ce genre de fonctions. La division du fichier d'entrée se fait de manière "manuelle", ce qui conduit à un coût supplémentaire en termes de temps d'exécution. Manuelle veut dire que nous stockons le fichier d'entrée dans un buffer et par la suite nous calculons chaque portion à envoyer et la stockons séparément dans un tableau pour finalement faire appel à la fonction d'envoi. Chaque nœud devant recevoir une portion, doit dans un premier temps recevoir la taille de la portion afin d'allouer l'espace nécessaire. Toute cette procédure est accomplie en un seul appel avec scatterv().

4.5 Correspondance de patrons

Ce module est spécifique aux cartes parallella. L'algorithme étant parallèle par nature, nous allons mesurer le gain en débit d'exécution en ajoutant une autre couche de parallélisation. Cette double parallélisation est schématisée dans la figure 4.10.

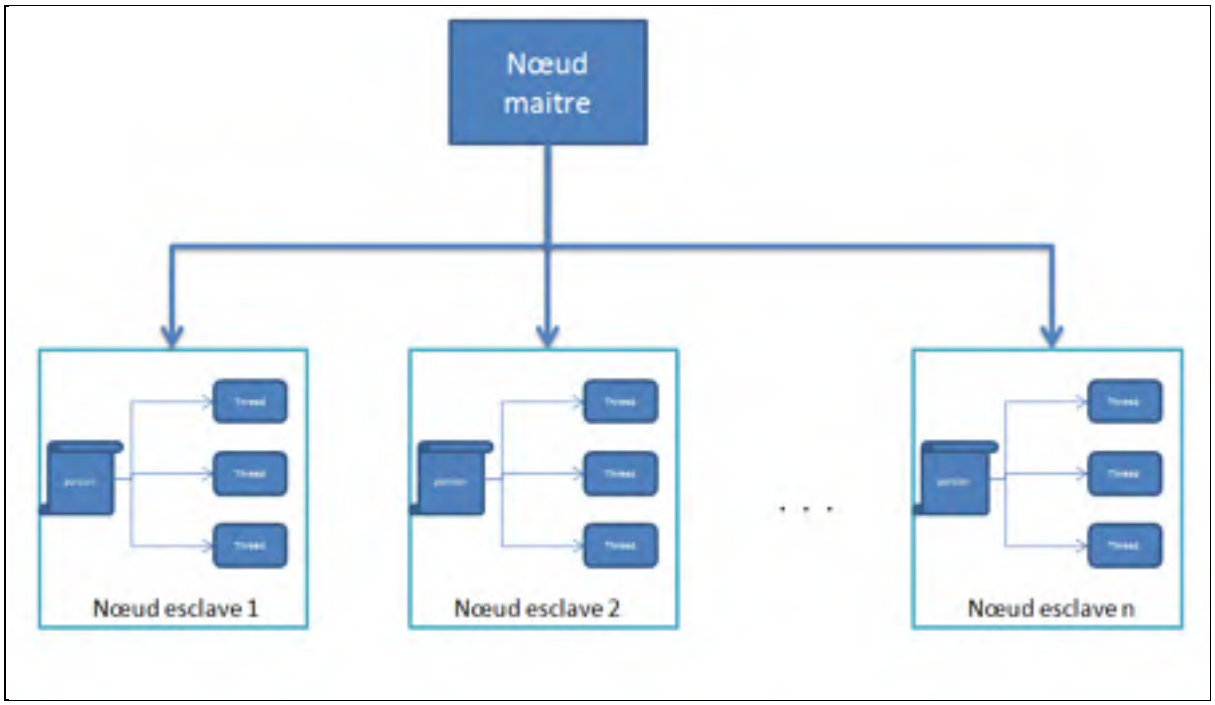

Figure 4.10Parallélisation de l'algorithme de correspondance des patrons.

Le nœud maître s'occupe de distribuer le fichier contenant les traces à analyser entre les différents nœuds composant le cluster. Le découpage et la distribution des différentes portions du fichier se font suivant deux scénarios : le premier consiste à partager le fichier de manière naïve, c'est-à-dire que toutes les portions seront égales. Le second scénario consiste à appliquer l'équilibrage des charges selon la stratégie que nous avons détaillée dans le chapitre précédent.

4.5.1 Distribution naïve

Dans ce scénario, tous les nœuds reçoivent des portions de même taille. Le nœud maître a la tâche de découper le fichier original suivant le nombre de nœuds esclaves présents dans le cluster. Pour un fichier de taille T, les portions seront égales à T/n où n est le nombre de nœuds.

Cependant, une particularité de l'algorithme de correspondance de patrons nous oblige à modifier un peu la distribution naïve pour des raisons de fiabilité. En effet, le but de cet

algorithme est de détecter une séquence malicieuse dans une longue liste d'appels systèmes. Procéder à un découpage aléatoire risquerait de séparer une séquence malicieuse en deux, envoyant chaque partie vers un nœud différent. L'algorithme serait alors dans l'incapacité de reconnaitre cette séquence puisqu'elle ne serait pas complète chez aucun des deux nœuds.

Pour remédier à cela, nous faisons en sorte lors du découpage de n'interrompre aucune séquence potentielle. Ceci a pour résultat que les portions se chevaucheront et qu'il y aura une redondance dans les données, cependant cette dernière serait infime. L'opération est décrite par la figure 4.11.

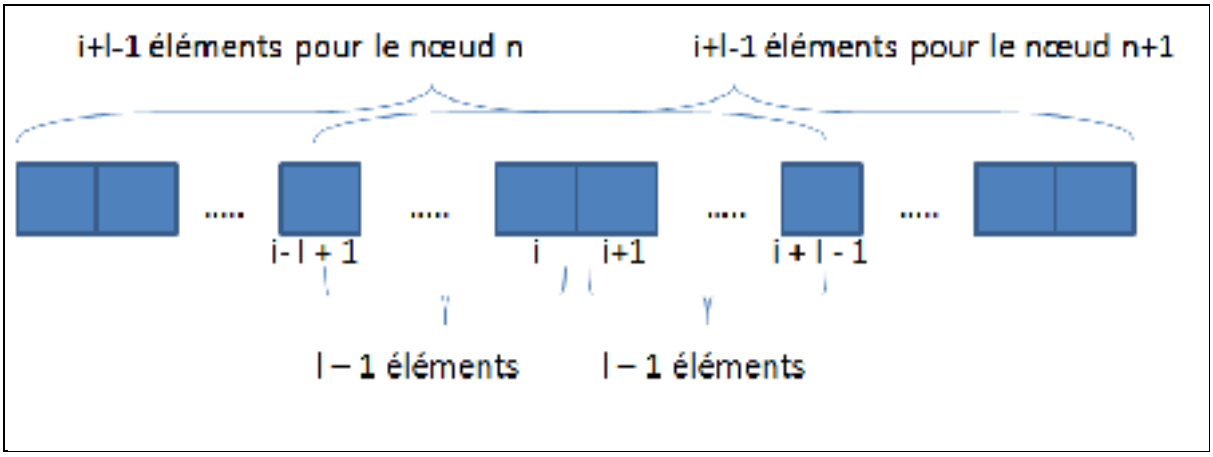

Figure 4.11Découpage des portions pour la correspondance des patrons.

Soit *l* la longueur maximale des séquences malicieuses dans la base de données servant à construire l'automate (voir chapitre 3). Supposons que la portion pour un nœud n s'arrête à la position i, on rajoute alors les *l*-1 éléments suivants (appartenant à la portion du nœud n+1) afin d'être sûr qu'aucune séquence n'a été découpée. De même, à la portion n+1, on rajoute *l*-1 éléments appartenant à la portion n.

Nous avons effectué les tests en variant le nombre des nœuds ainsi que la taille des données en entrée. Nous avons exécuté l'algorithme en utilisant MPI et PVM comme mécanisme de passage des messages.

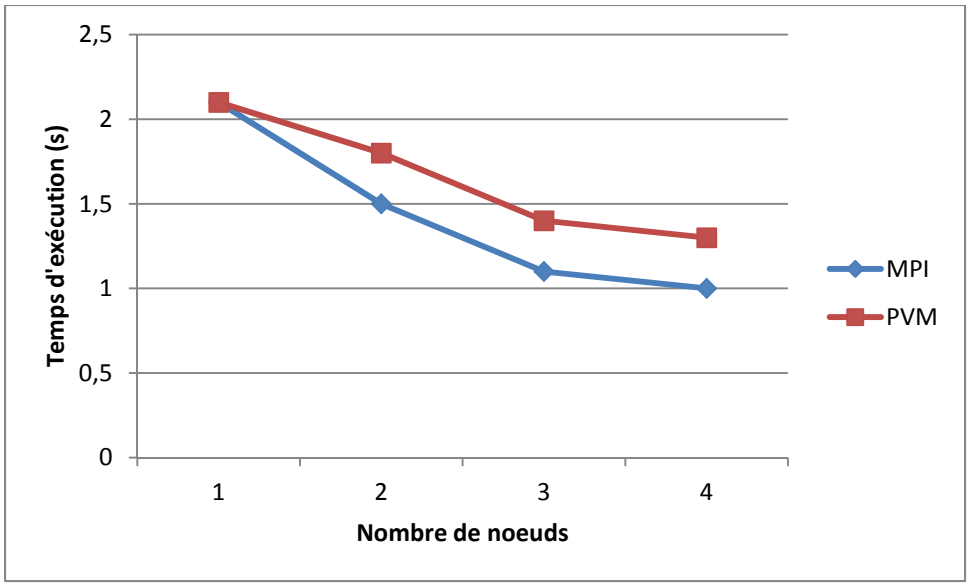

Figure 4.12Temps d'exécution en fonction du nombre de nœuds.

La figure 4.12 montre que les temps d'exécution obtenue avec MPI sont encore une fois inférieurs à celles obtenues en utilisant PVM et ce quel que soit le nombre de nœuds composant le cluster.

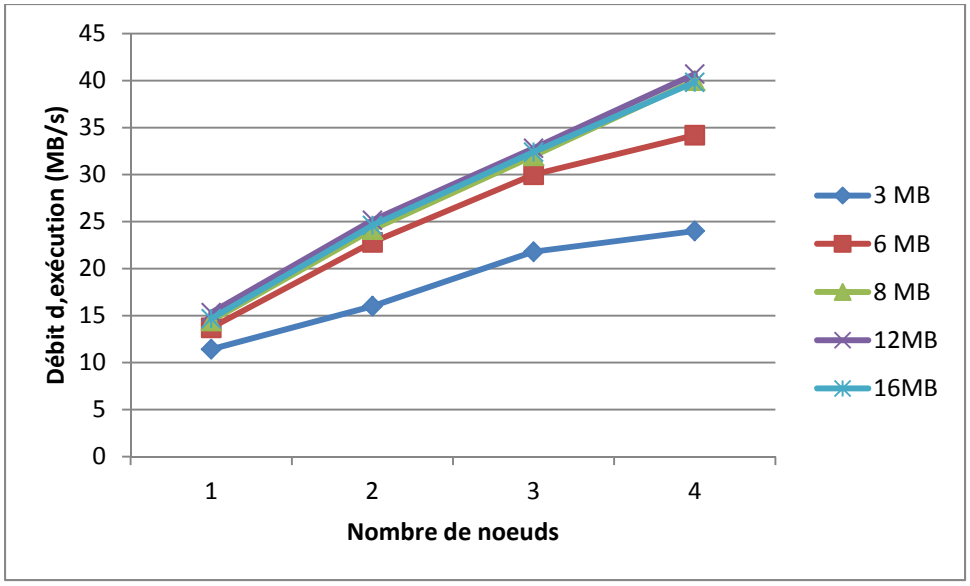

Figure 4.13 Débit d'exécution en fonction du nombre de nœuds et de la taille des entrées.

La figure 4.13 montre la variation du débit d'exécution en fonction de la taille des entrées et du nombre de nœuds, nous constatons que pour cet algorithme, la taille optimale est de 12MB. La différence avec les résultats obtenues en testant l'AES (8MB) est dû à la nature parallèle de l'algorithme de correspondance de patrons qui arrive à traiter plus de données simultanément en entrée.

4.5.2 Équilibrage de la charge

Le deuxième scénario consiste à découper le fichier différemment, en prenant en compte la capacité de chaque nœud à exécuter l'algorithme de correspondance des patrons. Nous suivons la stratégie décrite dans le chapitre précédent qui consiste à considérer la vitesse d'exécution de chaque nœud.

Vu que nous n'utilisons que des cartes parallella pour ce cluster, il est alors essentiel de différencier les nœuds afin que le processus d'équilibrage de la charge soit pertinent. La différenciation se fait en "chargeant" chaque carte différemment lors de l'exécution de l'algorithme de correspondance des patrons. C’est-à-dire que nous allons exécuter un autre algorithme en parallèle qui va consommer des ressources et ainsi nous augmentons intentionnellement le temps requis par un nœud pour exécuter l'algorithme de correspondance des patrons. L'algorithme exécuté en parallèle afin de charger la carte est une simple multiplication de matrice qui consommera assez de ressources pour impacter les performances de la carte.

Nous réalisons les tests suivants en prenant comme taille des entrées un fichier de 12MB et avec 4 nœuds composant le cluster. Les résultats sont montrés dans la figure 4.14.

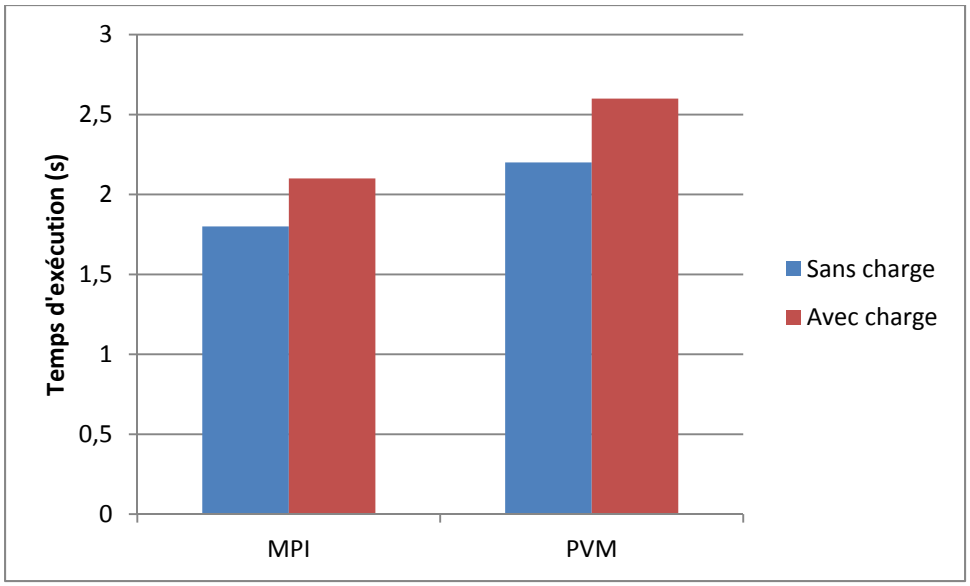

Figure 4.14Effet de l'addition de la charge sur le temps d'exécution.

Nous constatons que les résultats obtenus en chargeant les nœuds ne diffèrent pas de manière significative des résultats obtenus lorsque les nœuds sont libres. Le tableau 4.7 présente le temps d'exécution et le coefficient d'exécution pour chaque nœud.

Tableau 4.7 Coefficient d'exécution des nœuds.

Nœud	Temps d'exécution	Coefficient d'exécution
1	6.9	0.251
2	7.5	0.244
3	6.5	0.256
4	7.2	0.248

Pour cela, l'algorithme d'équilibrage de charge n'a pas un grand impact dans ce cas de figure comme le montre la figure 4.15.

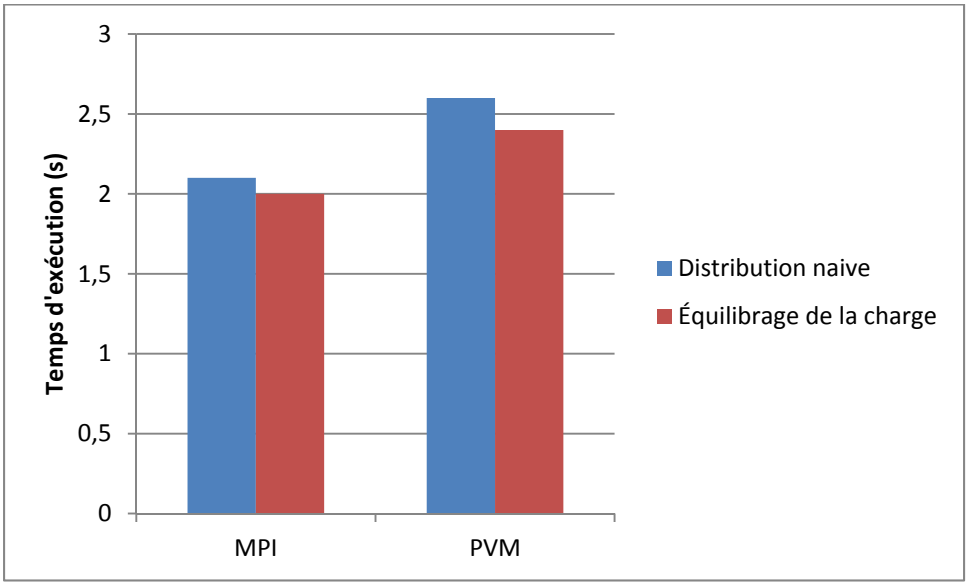

Figure 4.15Effet de l'équilibrage de la charge.

La raison pour laquelle il n'y a pas de grande différence est que l'algorithme de correspondance de patrons utilise principalement les coprocesseurs epiphany et que l'algorithme de multiplication de matrice consomme principalement de la CPU. Le programme destiné à charger la carte n'a pas alors d'impact significatif sur le temps d'exécution de l'algorithme de correspondance de patrons. La partie où la multiplication de matrice affecte notre programme principal, est lorsque ce dernier fait appel à la CPU afin de lire, distribuer et recevoir les données en entrée, cependant la concurrence à ce niveau n'est pas assez significative pour décroitre les performances du système. Augmenter la charge plus n'affecterait pas les performances puisqu'il n'est pas possible de surcharger le coprocesseur.

Le programme de correspondance des patrons utilisant tous les cœurs du coprocesseur, il nous est impossible de charger celui-ci encore plus. Lancer d'autres tâches sur le coprocesseur en parallèle conduit à des bugs et au plantage de toutes les tâches.

4.6 AES parallèle

Pour les expérimentations, nous procédons avec l'algorithme AES parallèle de la même manière que nous l'avons fait pour l'algorithme de correspondance des patrons, sauf que nous jugeons inutile de procéder à l'équilibrage de la charge, puisque les tests précédents ont démontré la limite de ce mécanisme dans le cas des applications s'exécutant sur le coprocesseur.

La distribution du fichier d'entrée se fera donc de façon égale entre tous les nœuds. La figure 4.16 montre le gain en terme de temps d'exécution suivant le nombre de nœuds. MPI montre de meilleurs résultats que PVM encore une fois.

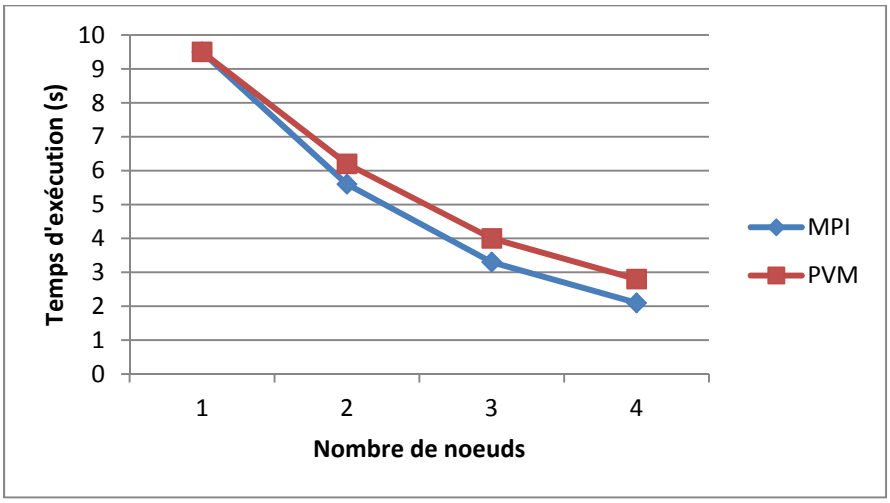

Figure 4.16L'exécution de PAES en fonction du nombre de nœuds.

Nous pouvons aussi remarquer que PAES délivre une meilleure performance que le AES séquentiel malgré un nombre de nœuds inférieur.

Nous avons aussi fait varier la taille du fichier en entrée afin de déterminer le meilleur débit d'exécution, la figure 4.17 montre les résultats

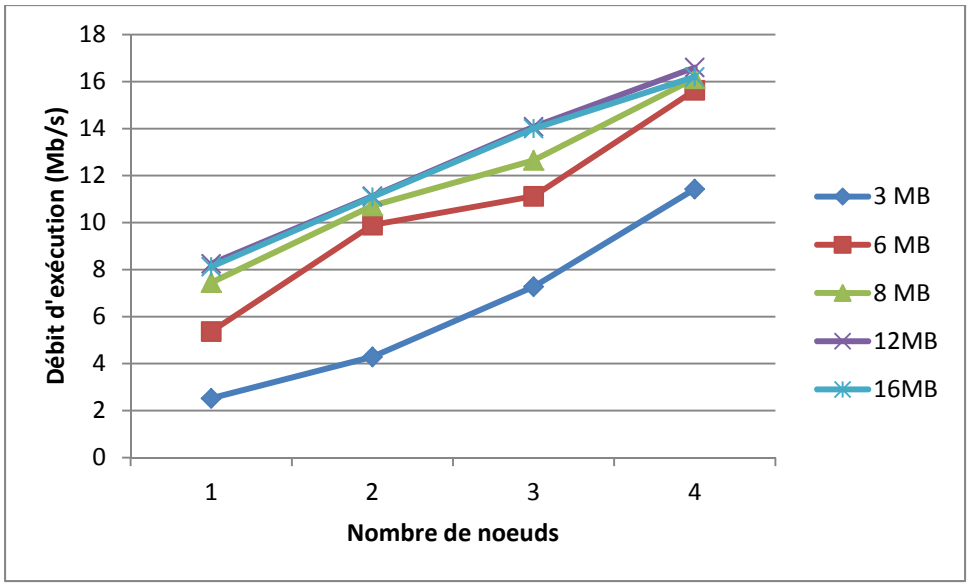

Figure 4.17Variation du débit d'exécution en fonction de la taille de l'entrée et du nombre de nœuds.

Les tests sont effectués en utilisant MPI, puisqu'il donne de meilleurs résultats que PVM. Nous constatons que le meilleur débit d'exécution est obtenu avec un fichier en entrée de 12MB.

4.7 Résultats de la virtualisation

4.7.1 Choix de la technologie et l'installation de docker

Il existe différentes technologies de virtualisation et chaque technologie a ses propres implémentations. Cependant, ces implémentations ne se valent pas et délivrent des performances différentes selon les études que nous avons mentionnées dans le chapitre de la revue de littérature.

Puisque les ordinateurs mono-cartes ARM sont dotés de ressources relativement limitées par rapport aux serveurs traditionnels, nous avons jugé pertinent de choisir la technologie de virtualisation par isolation, puisque c'est la plus légère et demande moins de ressources que les autres technologies (Morabito et al., 2015).

Parmi les implémentations de la virtualisation par isolation, Docker est la plus populaire du moment. Cette popularité vient du fait de sa simplicité de déploiement et d'usage, en plus d'offrir les performances de la virtualisation au niveau noyau et différents moyens de gestion.

Cependant, afin de pouvoir s'exécuter, Docker a besoin que plusieurs modules du noyau Linux soient activés comme les namespaces et les cgroups ainsi que des modules liés aux fonctionnalités réseau. Les distributions Linux destinées aux architectures traditionnelles telles que PC ou MAC sont mises à jour avec un noyau où ces modules sont activés par défaut. L'utilisateur n'a alors qu'à installer docker depuis les dépôts officiels de la distribution Linux qu'il utilise. En ce qui concerne les appareils ARM, ceci est une toute autre histoire, puisque ni les développeurs de Docker ni les fabricants (des cartes) ne considèrent le déploiement sur ces plateformes comme une nécessité. Ceci fait que les images fournies par les constructeurs pour leurs ordinateurs mono-cartes n'ont pas la configuration noyau nécessaire au fonctionnement de docker.

Le déploiement de Docker sur les ordinateurs mono-cartes devient alors délicat puisque nous devons créer notre propre image en configurant le noyau Linux et en le recompilant. La figure 4.18 montre les différents modules nécessaires au fonctionnement de Docker.

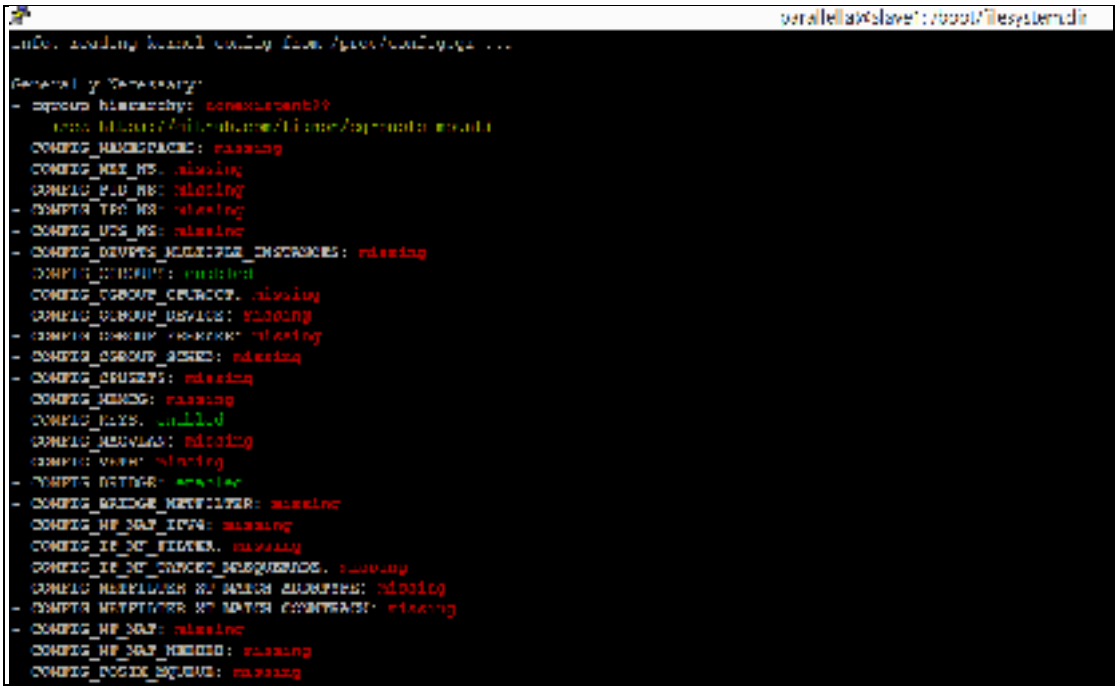

Figure 4.18 Modules nécessaires pour exécuter Docker

Nous avons donc recompilé les noyaux pour chaque type de carte en incluant les modules nécessaires, cependant cela n'a pas marché pour toutes les cartes puisqu'elles ont chacune un matériel différent et qu'en activant ou désactivant certains modules, nous risquons d'avoir une image qui ne fonctionne pas.

Une autre limite est que, afin de pouvoir déployer un cluster composé des containers Docker, il est indispensable d'utiliser un outil de gestion nommé Swarm[4], qui permet la gestion de l'adressage des différents nœuds virtuels et la gestion dynamique d'un cluster. Cependant, Swarm vient sous la forme d'une image Docker qui n'est pas adaptée à l'architecture ARM.

Finalement, la recompilation du noyau a fonctionné pour la carte beaglebone ainsi que la odroid-u3. Les tests de performance se feront donc sur ces deux cartes.

[4]https://docs.docker.com/swarm/

4.7.2 Tests de performance

Afin de tester l'impact éventuel de la virtualisation en utilisant Docker sur les appareils ARM, nous procédons à des tests de benchmark sur le système natif (sans container) ainsi que sur un container Docker.

Nous visons à mesurer l'éventuel coût supplémentaire que pourrait introduire docker en comparaison avec le système hôte. Pour ce faire, nous mesurons les performances en terme de vitesse de calcul (CPU), de vitesse d'accès à la mémoire RAM et de vitesse d'accès au périphérique de stockage.

Nous utilisons l'outil de benchmark open-source phoronix qui fournit une suite de tests fonctionnant sur les systèmes d'exploitation Linux. Nous présentons dans ce qui suit les tests effectués et leurs résultats.

Lors de ces tests, nous remarquons que la différence de performance entre les cartes est notable. Ceci est prévisible compte tenu de la différence des ressources disponibles dans chacune des cartes. En effet, la Odroid-u3 possède des caractéristiques supérieurs à la BeagleBone. Il est à noter que nous nous intéressons aux effets de la virtualisation sur chaque carte séparément.

4.7.2.1 CPU

Le test utilisé pour mesurer la performance de la CPU est l'encodage d'un échantillon WAV en format FLAC. Cette opération va tester la capacité d'exécution des instructions arithmétiques par le processeur.

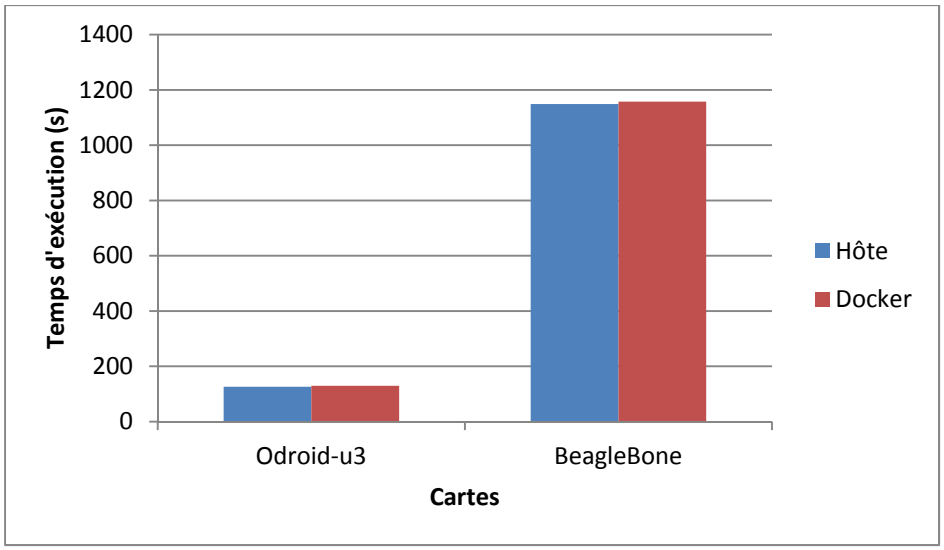

Figure 4.19Test d'encodage FLAC mesurant la performance CPU.

La figure 4.19 montre le résultat du test où nous pouvons voir que la performance de la CPU est presque pareille que ce soit sous le système hôte ou depuis un container docker. Le coût supplémentaire mesuré pour la beaglebone est de 0.8% et il est de 0.6% pour la odroid-u3.

4.7.2.2 RAM

La mémoire RAM est essentielle pour stocker des données temporaires ou volatiles. La vitesse d'accès à cette mémoire est importante pour la performance d'un système. Afin de tester l'effet de la virtualisation docker sur l'accès à la RAM, nous effectuons le benchmark RAMspeed offert par phoronix qui consiste à mesurer la débit d'accès en copiant une suite d'entier vers et depuis la mémoire.

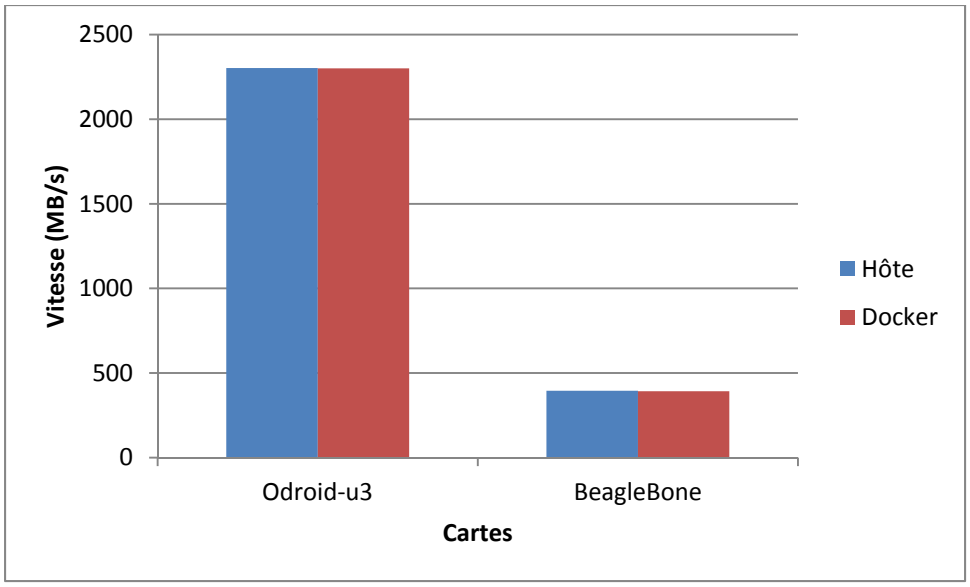

Figure 4.20Test RAMspeed mesurant la vitesse d'accès à la RAM.

Nous constatons que le débit n'est pas affecté quand on accède à la RAM depuis le container docker pour les deux cartes comme le montre la figure 4.20.

4.7.2.3 Disque

Le disque dur servant à stocker les données permanentes, il est aussi important de mesurer l'impact de la l'usage de docker sur l'accès à ce périphérique. Les ordinateurs mono-cartes ne possédant pas pour la plupart de disque dur interne, le stockage se fait sur des cartes sd. Le benchmark IOzone nous permet de tester la vitesse d'écriture sur ce type de périphérique en y écrivant un fichier de grandeur 4GB avec des blocs de 1MB.

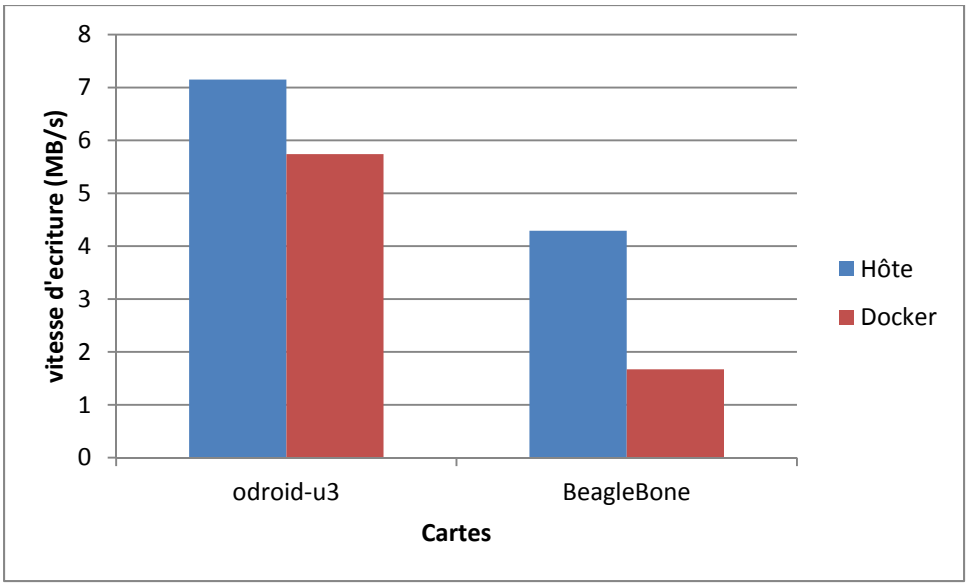

Figure 4.21Test IOzone d'écriture sur le disque.

Les résultats du test de la figure 4.21 montrent une diminution importante de la vitesse d'écriture sur le disque avec l'utilisation de docker. Cette diminution est de 18% pour la odroid-u3 et de 61% pour la beaglebone. Cette diminution dans l'écriture sur le disque provient du système de fichier utilisé par docker pour les containers. En effet, ce dernier utilise les systèmes de fichier UFS (*Union File System*) qui fonctionnent en créant des couches pour chaque container. Plusieurs variantes de ce système sont utilisées par docker et elles ne sont pas toutes pareilles en terme de vitesse. Dans notre expérimentation, nous avons installé docker sur la odroid-u3 avec le système de fichier AUFS (*Advanced Union File System*) et sur la beaglebone avec le système de fichier vfs. Nous constatons que le bon choix du système de fichier est important au vu de la différence de vitesse perdue entre la beaglebone et la odroid-u3. En effet, la figure 4.21 montre une chute de 18% dans la vitesse d'écriture sur disque pour la Odroid-u3, alors que cette chute avoisine les 60% pour la BeagleBone.

4.7.2.4 Réseau

Pour tester l'impact de Docker sur les interfaces réseau du système, nous utilisons un outil de Benchmark iperf. Cet outil permet de tester le débit entre deux nœuds sur un lien réseau.

Nous testons la communication TCP et la communication UDP. Iperf est sous la forme d'une application client/serveur. Le serveur reste en écoute sur un port et attend le transfert de paquets issus d'un client. Nous utilisons comme serveur un PC avec Ubuntu 12.04, les clients sont les cartes (avec et sans Docker).

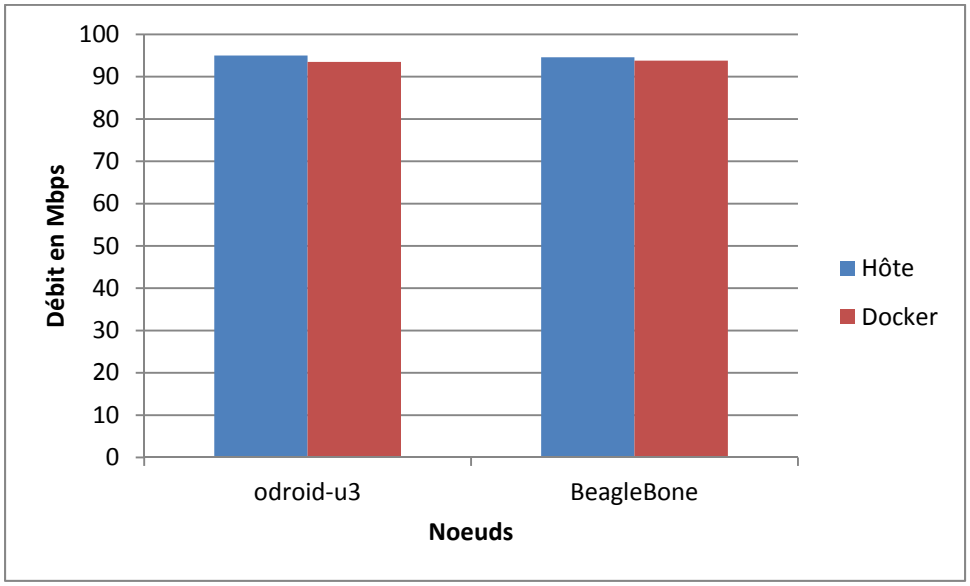

Figure 4.22Débit de transfert TCP.

Les résultats du test de transfert TCP sont montrés par la figure 4.22. On constate que la dégradation est de l'ordre des 1 % pour la odroid-u3 et la beaglebone lorsqu'on utilise Docker.

Le test de transfert UDP permet d'avoir une mesure des paquets perdus dans chaque cas. Le tableau 4.8 montre ces résultats. Pour ce qui est du débit de transfert, comme pour le cas de TCP, docker n'engendre pas de pertes de débits significatives.

4.8 Perte de paquets lors d'un transfert UDP

Carte	Hôte	Docker
Odroid-u3	0.0037%	3.5%
BeagleBone	0.004%	2.7%

Nous constatons une augmentation significative du taux de perte, probablement dû aux couches ajoutées par Docker dans la gestion du trafic réseau.

4.8 Conclusion

Dans ce chapitre nous avons présenté les résultats expérimentaux réalisés sur les ordinateurs mono-cartes ARM. Nous avons en premier lieu présenté les défis techniques que nous avons rencontré pour mettre en place l'environnement de test. Ces défis consistaient à installer les systèmes d'exploitation adéquats pour faire fonctionner les cartes ainsi que la technologie de virtualisation qui n'est pas supportée par défaut.

Les expérimentations sont divisées en deux parties, la première a concerné l'architecture de cluster et les algorithmes implémentés avec différentes configurations. Ces tests nous ont permis d'avoir une estimation de la capacité des clusters formés avec des cartes ARM à traiter des tâches de calcul des algorithmes de sécurité. Nous avons aussi présenté notre solution pour la distribution de la charge à travers le cluster qui s'est avérée efficace dans le cas des algorithmes utilisant la CPU.

Les expérimentations ont permis d'évaluer la performance du cluster selon sa configuration et démontré qu'avec le bon choix de configuration, ces appareils ARM peuvent au moins délivrer la même performance qu'un ordinateur de bureau. De plus, les expérimentations ayant été effectuées en utilisant MPI et PVM, nous ont permis de déterminer que MPI permettait d'avoir de meilleurs résultats que PVM.

La deuxième partie de ce chapitre a concerné la virtualisation sur les ordinateurs mono-cartes ARM. Nous avons motivé le choix de la technologie qui est Docker, puisqu'elle est la plus populaire et la plus rapide du marché en ce moment et qu'elle offre une facilité de déploiement remarquable. Des difficultés techniques liées aux images fournies avec les cartes nous ont restreints dans le déploiement de Docker sur toutes les cartes. Nous avons toutefois réussi à le faire fonctionner sur deux cartes : Odroid-U3 et BeagleBone. Afin de déterminer l'impact des la virtualisation sur ces cartes, nous avons effectué une série de benchmarks afin de tester la CPU, l'accès à la RAM et au disque de stockage ainsi que la communication

réseau. Ces séries de tests ont démontré que la virtualisation ne dégrade pas les performances du système sauf dans le cas de l'écriture sur disque, où cette dégradation varie selon le système de fichier utilisé.

CONCLUSION

La technologie ARM domine le marché des systèmes embarqués, surtout celui des téléphones intelligents et des tablettes. Ceci est principalement dû au fait que les processeurs ARM consomment peu d'énergie comparée aux autres processeurs tels que Intel. Cette faible consommation s'allie bien avec le caractère limité des systèmes embarqués. En effet, ces derniers offrent un environnement avec des ressources limitées et l'autonomie de la batterie est un argument de vente important pour ces appareils.

Cette faible consommation d'énergie commence à attirer l'attention hors du domaine des téléphones intelligents et des tablettes, en effet le domaine du HPC devenant de plus en plus répandu et accessible grâce aux clusters beowulf, intégrer les appareils possédant des processeurs ARM peut s'avérer bénéfique. D'une part, les ordinateurs mono-cartes ARM ne sont pas coûteux (les prix vont de 35$ à 150$)(stromium, 2016), d'autre part, ils consomment peu d'énergie par rapport à la performance délivrée.

Aussi, la technologie de la virtualisation connait un essor fulgurant et ses implémentations sont de plus en plus nombreuses . En combinant la virtualisation et le concept de cluster, il est possible d'obtenir les performances des deux technologies simultanément. En effet, un cluster à la fois physique et virtuel bénéficierait d'une facilité de déploiement de la flexibilité ainsi que de l'isolation apportée par la virtualisation.

Nous avons donc à travers ce travail, étudié la possibilité de l'utilisation des ordinateurs mono-cartes à processeurs ARM dans le domaine du HPC ainsi que la capacité de ces derniers à supporter la virtualisation.

Nous avons présenté dans ce mémoire les notions fondamentales à la compréhension de ce travail dans la première partie, à savoir les clusters, les deux principales techniques de transfert de message au sein d'un cluster, les appareils ARM et leur évolution et finalement le concept et les technologies de virtualisation. Le second chapitre concerne la revue de littérature où nous avons présenté dans un premier lieu les différents usages des clusters Beowulf et notamment ceux composés d'appareils ARM. Dans la partie suivante, nous citons

les études effectuées sur la comparaison des deux mécanismes de passage de messages dans un cluster qui sont MPI et PVM ainsi que les critères de comparaison utilisés. Finalement, ce chapitre s'achève par les études faites sur les techniques de virtualisation et notamment sur les appareils ARM. Les chapitres suivants détaillent la contribution de ce travail.

Nous avons détaillé dans le troisième chapitre l'architecture du cluster ainsi que les algorithmes qui ont été déployés dessus. Nous avons aussi expliqué la mise au point d'un système de monitorage adapté à ce type de cluster. De plus, nous avons présenté un système d'équilibrage de la charge à travers le cluster en vue d'optimiser les performances de ce dernier.

Le quatrième chapitre contient les étapes nécessaires à la mise en place de l'environnement et présente les résultats des expérimentations qui sont divisés en deux grandes parties : les expérimentations sur le cluster et les trois algorithmes implémentés et les expérimentations sur la virtualisation.

Pour ce qui est de l'utilisation des appareils ARM dans un cluster, nous avons conclu en vue des résultats que, en utilisant les bons mécanismes de passage de messages et les bonnes configurations, il était possible d'obtenir de bons résultats. Les résultats ont aussi démontré que MPI dépasse PVM en terme de performance. Notre solution d'équilibrage de la charge a montré qu'il était possible d'améliorer les performances du cluster dans le cas où les nœuds sont chargés différemment.

Les résultats de la virtualisation sur les deux cartes odroid-u3 et beaglebone en utilisant docker ont montré qu'il est possible d'utiliser la virtualisation sans pour autant dégrader la performance du système dans la plupart des tests. Nous avons testé l'effet sur la CPU, la RAM et l'écriture sur disque. Cette dernière a montré une dégradation dans la vitesse d'écriture qui varie selon le système de fichier utilisé.

Ce travail a montré le potentiel des ordinateurs mono-cartes ARM dans le domaine du HPC ainsi que la possibilité d'intégrer la virtualisation. Un travail futur pourrait considérer

l'utilisation d'un cluster de plus grande échelle et adapter les images du noyau Linux utilisées sur les cartes afin de permettre l'utilisation de Docker sur un plus de cartes.

Cependant, les ordinateurs mono-cartes ARM n'étant pas encore destiné au grand public, elles manquent du support et de la stabilité des plateformes traditionnelles. Manipuler et implémenter différentes solutions sur ces cartes requiert des connaissances et des outils nécessaires afin d'installer les environnements et les systèmes d'exploitation dessus.

Comme travaux futurs, il serait intéressant de trouver un moyen pour exécuter docker sur plusieurs appareils ARM, vu que dans ce travail, nous n'avons réussi qu'à l'installer sur deux cartes. Aussi, l'adressage réseau des containers Docker n'est pas encore optimisé pour la virtualisation, difficile d'accéder à un container depuis l'extérieur grâce à son adresse IP sans passer par des modifications qui parfois ne sont pas disponibles sur des images destinées aux appareils ARM.

BIBLIOGRAPHIE

Abdellatif, Manel, Chamseddine Talhi, Abdelwahab Hamou-Lhadj et Michel Dagenais. 2015. « On the Use of Mobile GPU for Accelerating Malware Detection Using Trace Analysis ». In., p. 42-46. < http://dx.doi.org/10.1109/SRDSW.2015.18 >.

Adapteva. 2016. « parellella ». < https://www.parallella.org/ >.

Al Geist, Adam Beguelin, Jack Dongarra. 1994. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press Scientific and Engineering ComputationJanusz Kowalik, Editor.

Arm. 2015a. « architectures ». < http://www.arm.com/products/processors/instruction-set-architectures/ >.

Arm. 2015b. « cortex ». < https://www.arm.com/cortex-A15 >.

Baker, M. A., M. Grove et A. Shafi. 2006. « Parallel and Distributed Computing with Java ». In *Parallel and Distributed Computing, 2006. ISPDC '06. The Fifth International Symposium on*. (6-9 July 2006), p. 3-10.

Barney, Blaise. 2015. « MPI ».

Barroso, L. A., J. Clidaras et U. Hoelzle. 2013. *The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines*. Coll. « The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines ». Morgan & Claypool, 154 p.

Biederman, E. 2006. « Multiple instances of the global linux namespaces ». *Proceedings of the Linux Symposium,* vol. 1, p. 101-112.

Burger, Thomas. 2012. « virtualization technology-in the enterprise ».

Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes et Robert E. Gruber. 2008. « Bigtable: A Distributed Storage System for Structured Data ». *ACM Trans. Comput. Syst.,* vol. 26, n^{o} 2, p. 1-26.

Datti, Ahmad A., Hadiza A. Umar et Jamil Galadanci. 2015. « A Beowulf Cluster for Teaching and Learning ». *Procedia Computer Science,* vol. 70, p. 62-68.

Dean, Jeffrey, et Sanjay Ghemawat. 2008. « MapReduce: simplified data processing on large clusters ». *Commun. ACM,* vol. 51, n^{o} 1, p. 107-113.

Denx. 2016. « The DENX U-Boot and Linux Guide ».

Docker. 2016. « engine ». < https://docs.docker.com/engine/ >.

Felter, Wes, Alexandre Ferreira, Ram Rajamony et Juan Rubio. 2014. « An Updated Performance Comparison of Virtual Machines and Linux Containers ». *Technology,* vol. 25482, p. 171-172.

Floyer, David. 2013. « Nightmare Worsens ». < http://wikibon.org/wiki/v/Intel_Nightmare_Worsens_with_Wearable_Devices_and_64-Bit_ARM_Processors >.

Forum, Message Passing Interface. 2012. *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart; 3.1 edition

Geist, G. a, J. a Kohl et P. M. Papadopoulos. 1996. « PVM and MPI: a Comparison of Features ». *Calculateurs Paralleles*, p. 137-150.

Gropp, William, Ewing Lusk et Computer Science Division. « Goals Guiding Design: PVM and MPI ».

Hamilton, James. 2009. « Cooperative Expendable Micro-Slice Servers (CEMS): Low Cost , Low Power Servers for Internet-Scale Services ». *Power*, p. 1-8.

HardKernel. 2016. « Odroid ». < (http://www.hardkernel.com/main/products/prdt_info.php?g_code=g138745696275&tab_idx=1 >.

Hess, Ken. 2016. « Top 10 Virtualization Technology Companies for 2016 ». < http://www.serverwatch.com/server-trends/slideshows/top-10-virtualization-technology-companies-for-2016.html) >.

Highscalability. 2008. « Collectl - Performance Data Collector ». < http://highscalability.com/product-collectl-performance-data-collector >.

Insidehpc. 2016a. « hpc architecture ». < http://insidehpc.com/hpc101/hpc-architecture-for-beginners >.

Insidehpc. 2016b. « insidehpc ». < http://insidehpc.com/hpc-basic-training/what-is-hpc/ >.

Instruments, Texas. 2014. « beagleboard ». < beagleboard.org >.

Instruments, Texas. 2016a. « OMAP5432 ».

Instruments, Texas. 2016b. « OMAP5432 firt-boot.sh ».

Isctruments, texas. 2016. « pandaboard ». < (http://pandaboard.org/content/resources/references) >.

Kaewkasi, C., et W. Srisuruk. 2014. « Optimizing performance and power consumption for an ARM-based big data cluster ». In *TENCON 2014 - 2014 IEEE Region 10 Conference*. (22-25 Oct. 2014), p. 1-6.

Katta, Dileep. 2014. « Pandaboard-ES Rev ».

Kendall, Wes. 2016. « MPI send and receive ». < http://mpitutorial.com >.

Kivity, Avi, Dor Laor, Glauber Costa et Pekka Enberg. 2014. « OSv—Optimizing the Operating System for Virtual Machines ». *Proceedings of the 2014 USENIX Annual Technical Conference*, p. 61-72.

Layton, Jeff. 2016. « Monitoring HPC Systems ». < http://www.admin-magazine.com/HPC/Articles/Processor-and-Memory-Metrics >.

Le, Deguang, Jinyi Chang, Xingdou Gou, Ankang Zhang et Conglan Lu. 2010. « Parallel AES algorithm for fast data encryption on GPU ». *ICCET 2010 - 2010 International Conference on Computer Engineering and Technology, Proceedings,* vol. 6, p. 1-6.

Levy, By Markus, et Convergence Promotions. 1990. « The History of The ARM Architecture : From Inception to IPO ». *Electronics*, p. 14-19.

Marshall, David. 2016. « benefits of server virtualization ». < http://www.infoworld.com/article/2621446/server-virtualization/server-virtualization-top-10-benefits-of-server-virtualization.html >.

Veuillez sélectionner un type de document autre que « Generic » afin de faire afficher la référence bibliographique.

Merkey, Phile. 2015. « beowulf history ». < http://www.beowulf.org/overview/history.html >.

Morabito, R., J. Kj, x00E, llman et M. Komu. 2015. « Hypervisors vs. Lightweight Virtualization: A Performance Comparison ». In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. (9-13 March 2015), p. 386-393.

Oberheide, Jon, E. Cooke et F. Jahanian. 2008. « CloudAV: N-version Antivirus in The Network Cloud ». *Proceedings of the 17th conference on Security symposium*, p. 91-106.

Oberheide, Jon, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn et Farnam Jahanian. 2008. « Virtualized in-cloud security services for mobile devices ». *Proceedings of the First Workshop on Virtualization in Mobile Computing - MobiVirt '08*, p. 31-31.

Ou, Z., B. Pang, Y. Deng, J. K. Nurminen, A. Yl, x0E, J., x0E, x0E, ski et P. Hui. 2012. « Energy- and Cost-Efficiency Analysis of ARM-Based Clusters ». In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. (13-16 May 2012), p. 115-123.

Portokalidis, Georgios, Philip Homburg, Kostas Anagnostakis et Herbert Bos. 2010. « Paranoid Android: Versatile Protection For Smartphones ». *Annual Computer Security Applications Conference (ACSAC)*, p. 347-356.

Redhat. 2016. « Virtualization ». < https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Virtualization/pr01s05.html >.

RIDWAN, MAHMUD. 2015. « isolating system with linux namespaces ». < https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces >.

Rouse, Margaret. 2016. « Advanced Encryption Standard ». < http://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard >.

Securityguy. 2014. « virtualisation ». < https://securityguy225.wordpress.com/tag/virtualisation-2/ >.

Senning, jonathan. 2016. « HPC History ».

Sharma, R., et P. Kanungo. 2014. « Dynamic Load Balancing Algorithm for Heterogeneous Multi-core Processors Cluster ». In *Communication Systems and Network Technologies (CSNT), 2014 Fourth International Conference on*. (7-9 April 2014), p. 288-292.

Shilov, Anton. 2014. « ARM Microprocessors ». < http://www.xbitlabs.com/news/mobile/display/20081022084110_Intel_Criticizes_Modern_Smartphones_Blames_ARM_Microprocessors.html >.

stromium. 2016. « rockchip ». < (http://armdevices.net/category/chip-provider/rockchip/) >.

Sunderam, Vaidy S., Zsolt N, #233 et meth. 2001. « A Comparative Analysis of PVM/MPI and Computational Grids ». In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. p. 14-15. 749468: Springer-Verlag.

Vmware. 2016. « virtualization overview ». < http://www.vmware.com/ca/en/virtualization/overview >.

Warwick. 2015. « Why is hpc important ». < http://www2.warwick.ac.uk/fac/cross_fac/hpc-sc/importance >.