Université de Strasbourg

École doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur
Laboratoire ICube

THÈSE présentée par :

Étienne Michon

soutenue le : **5 juin 2015**
financée par : Direction générale de l'armement

pour obtenir le grade de : **Docteur de l'université de Strasbourg**
Discipline / Spécialité : Informatique

Allocation dynamique sur cloud IaaS

Allocation dynamique d'infrastructure de SI sur plateforme de cloud avec maîtrise du compromis couts/performances.

THÈSE dirigée par :
Stéphane Genaud, professeur des universités, université de Strasbourg

Encadrant :
Julien Gossa, maitre de conférences, université de Strasbourg

---

Rapporteurs :

Eddy Caron, maitre de conférences, école normale supérieure de Lyon

Christophe Cérin, professeur des universités, Paris XIII

Autres membres du jury :

Laurent Philippe, professeur des universités, université de Franche Comté

Remerciements

De mon master 2 à la rédaction de cette thèse aujourd'hui, cinq années bien remplies se sont écoulées. Je souhaite ici remercier de nombreuses personnes qui m'ont apporté aide et soutien tout au long de ce travail de recherche.

Je tiens tout d'abord à remercier Dr Eddy Caron, Pr Christophe Cérin et Dr Laurent Philippe pour avoir accepté d'évaluer mes travaux de recherche et de faire partie de mon jury de thèse.

J'adresse des remerciements tout particuliers et chaleureux à mes encadrants de thèse, Stéphane Genaud et Julien Gossa pour leurs conseils, mêmes contradictoires, qui m'auront été précieux durant ces quatre années. Merci également à eux pour leur patience dont j'ai usé (et abusé) au moment d'écrire des articles en anglais.

Je tiens également à chaudement remercier les personnes qui m'ont apporté leur précieux soutien : Romaric David et Michel Ringenbach pour l'aide à l'utilisation de Slurm et des traces d'exécution du mésocentre de l'Université de Strasbourg, Vegard Engen et tous les membres de la plateforme BonFIRE, ainsi que la Direction Générale de l'Armement du ministère de la Défense pour le financement de mes recherches.

Je remercie l'ensemble des enseignants de l'IUT d'Illkirch de m'avoir accueilli les bras ouverts au sein de leur équipe pédagogique. Ils m'ont conforté dans ma volonté d'enseigner l'informatique. Je pense ici à tous les enseignants ainsi que le personnel administratif, toujours disponible.

Je souhaite par ailleurs remercier tous les chercheurs de l'ICPS. La bonne ambiance qui règne au laboratoire m'a permis de réaliser cette thèse dans un cadre de travail très agréable. L'initiation au backgammon et les parties postprandiales de jeux vidéo assurent un environnement propice à la réflexion. Je présente mes plus plates excuses à Aravind et Juan pour avoir failli à ma mission de leur faire vivre un mariage français. J'en profite pour féliciter Jean-François d'avoir, de son côté, réussi cette mission !

Je tiens ensuite à remercier ma famille qui me supporte toute l'année : humorus, fraternitus, vitaestcoolus. Merci à Chloé qui m'a réconforté dans les moments les plus stressants de ces années. Merci à Tatie Josette pour sa relecture attentive de l'introduction et ses tentatives inlassables et répétées pour la comprendre.

Finalement, je tiens à remercier tous mes amis : les Sesquidistus pour les tournois à l'autre bout de la France et de l'Europe qui m'ont permis de travailler durant les longues heures de minibus, les Scouts pour le réveil matinal du samedi matin, et les Kêkês parce qu'ils sont la force de l'imagination au service du changement du monde

À l'ensemble de ces personnes, MERCI !

Sommaire

Chapitre 1

Introduction

Au commencement, l'Homme créa l'ordinateur. L'ordinateur était seul et calculait lentement. Au premier jour, John McCarthy énonce pour la première fois l'envie de voir la puissance de calcul informatique vendue comme un service, au même titre que l'eau, l'électricité, ou le téléphone. Autrement formulé, John McCarthy souhaite voir la puissance de calcul vendue à la demande, avec une facturation en fonction de la durée d'utilisation. Nous sommes en 1961.

Au deuxième jour, l'Homme conçoit de grands centres de calcul au sein desquels de nombreux ordinateurs — géographiquement très proches — sont reliés en réseau. L'objectif de ces ordinateurs est de réaliser un calcul similaire sur des données différentes afin d'obtenir un résultat plus rapidement. Le concept de répartir un calcul entre plusieurs processeurs s'appelle le *calcul distribué*. Ces groupements d'ordinateurs s'appellent des fermes de calcul (ou *cluster* en anglais).

Au troisième jour, avec l'amélioration de la capacité des réseaux, l'Homme relie ces fermes de calcul avec un réseau capable de partager les informations entre elles, quelle que soit leur position géographique. Cet assemblage de plusieurs fermes s'appelle la *grille de calcul*, par analogie avec le réseau de distribution de l'électricité aux États-Unis (*power grid*). La grille de calcul est importante au sein de la communauté scientifique. Son principal objectif est de permettre au propriétaire d'un cluster d'accéder à plus de puissance de calcul. À cette fin, il rend son cluster en partie disponible à d'autres membres de la grille en échange d'un accès aux clusters de ces mêmes membres. Ces grilles de calcul forment un ensemble de machines physiques aux caractéristiques *hétérogènes* reliées entre elles par un réseau permettant l'échange de données.

Écrire un programme à exécuter sur une grille est très difficile, particulièrement à cause de sa nature intrinsèquement hétérogène. Pour réaliser cette tâche, un utilisateur peut avoir besoin de spécifier une contrainte sur les caractéristiques physiques de la machine : présence d'une carte graphique, puissance de calcul minimal nécessaire, quantité de mémoire vive disponible… et surtout, présence de logiciels spécifiques. À ces contraintes de l'utilisateur s'ajoutent des contraintes liées aux différences de fonctionnement des différents clusters accessibles : protocole d'accès aux ressources, contrainte sur la sécurité afin de respecter les politiques locales de sécurité. Ces listes de contraintes sont évidemment non-exhaustives. Il se pose, de plus, pour l'administrateur d'un cluster la question de savoir comment les tâches des utilisateurs doivent être ordonnancées, c'est-à-dire quelles tâches

sont exécutées sur quelles machines physiques et comment suivre l'activité de l'ensemble des tâches de la plateforme et des machines physiques. L'ensemble de ces problèmes est résolu par l'utilisation d'un *middleware* de grille qui facilite le développement d'applications dans un environnement distribué. Nous pouvons par exemple citer les middleware Globus Toolkit [20], gLite [34], ARC [18].

Malgré un certain succès, la grille présente des inconvénients dans la difficulté à mettre en place et exploiter le middleware. En effet, une des caractéristiques principales de la grille est sa nature multi-utilisateur et multi-administrée. Chaque cluster est administré indépendamment. Un utilisateur qui souhaite exécuter un logiciel sur une grille doit passer par de nombreux administrateurs avant de pouvoir effectivement réaliser ses expériences. Le middleware doit prendre en charge ces nombreux aspects le rendant très complexe. Il est ainsi une grande source d'erreurs. De plus, la détection et la gestion des erreurs dans un tel environnement ne sont pas aisées. En effet, l'empilement de logiciels qui constituent le middleware et l'accès indirect aux machines réalisant les calculs posent problème pour la détection des erreurs. Cette détection se fait alors généralement à la suite de l'expiration d'un time out qui peut être long par rapport aux besoins applicatifs.

En parallèle du développement de la grille, les technologies de virtualisation se sont affinées. Les systèmes de virtualisation sont des logiciels qui imitent le comportement d'une machine physique. Ils permettent ainsi de faire fonctionner plusieurs machines, dites virtuelles, sur une seule machine physique. Si cela peut sembler contre-intuitif, cette technologie est importante pour simplifier la gestion des clusters. La conséquence de l'utilisation d'une telle technologie est de rendre caduque l'utilisation du middleware. En effet, chaque utilisateur est ainsi administrateur de ses propres machines virtuelles et peut y installer les logiciels qu'il souhaite. La virtualisation est bien moins lourde à mettre en œuvre et bien plus transparente pour l'utilisateur que le middleware de la grille. Celui-ci a alors la possibilité de retrouver un environnement familier pour l'exécution de ses applications.

Au quatrième jour, l'Homme estime que la virtualisation permet de faire encore un pas en avant vers le rêve de John McCarthy de 1961. Depuis 2006, le terme de « cloud computing » a été popularisé. Derrière ce terme grand public se cache une tendance économique et technologique qui concerne tous les niveaux impliqués dans les services entre un fournisseur et un client. Il s'agit aussi bien d'applicatif (le fournisseur exécute l'application pour le client et lui rend les résultats), que d'exploitation du matériel (le fournisseur offre son matériel en location, ainsi qu'une infrastructure logicielle pour l'exploiter). La taxonomie actuelle décline de manière plus précise le cloud computing en trois catégories principales :

Software as a Service (SaaS) désigne des logiciels installés sur des serveurs auxquels les utilisateurs accèdent en général par un navigateur web. Les webmails sont des exemples de SaaS.

Platform as a Service (PaaS) propose une plateforme d'exécution pour les logiciels développés par les utilisateurs. Le PaaS permet de faciliter le déploiement d'une telle application en faisant abstraction de la complexité de l'environnement d'exécution. C'est le fournisseur de PaaS qui gère l'infrastructure physique sous-jacente et son dimensionnement en fonction des besoins de l'application.

Infrastructure as a Service (IaaS) offre une infrastructure matérielle *virtualisée*. Cela permet au client une personnalisation totale du système d'exploitation. Si le matériel se trouve en réalité chez le fournisseur, le client y a, en revanche, accès sans restriction comme si les ressources lui appartenaient.

Nous nous concentrons dans cette thèse sur ce dernier type de cloud.

L'histoire de l'IaaS est faite d'une légende qui remonte à Noël 2005. Pour tenir la charge importante des achats lors de cette période, le site d'e-commerce Amazon aurait acheté de nombreux serveurs. Après cette période, le site web se trouva alors avec quantité de matériel informatique inutile [66]. Ils décidèrent donc de lancer le 25 août 2006 une offre IaaS, Amazon EC2 [2], afin de continuer à rentabiliser l'achat important effectuer quelques mois plus tôt en louant les machines inutilisées. Si nous savions jusque là faire des grilles de calcul ou des clusters, le partage d'une machine physique entre plusieurs utilisateurs a toujours été problématique. Mais la virtualisation est une technologie devenue suffisamment mûre pour résoudre ce problème. La machine virtuelle (VM pour *Virtual Machine*) est le conteneur qui permet de faire fonctionner ces systèmes de façon autonome. Ce modèle de cloud computing est celui de plus bas niveau puisque l'utilisateur peut installer et utiliser les logiciels qu'il souhaite.

Les grilles et les IaaS présentent des différences majeures. Tout d'abord, d'un point de vue économique, les grilles et les IaaS répondent à deux problématiques différentes. Sur la grille, les utilisateurs payent pour investir dans une infrastructure importante pour son fonctionnement. À l'inverse, sur un cloud IaaS, le fournisseur paye pour investir dans l'infrastructure et le client paye simplement pour son fonctionnement. Deuxièmement, les grilles et les IaaS offrent un accès aux logiciels très différents. La grille permet à l'utilisateur de choisir les logiciels qu'il souhaite utiliser alors que les clients du cloud IaaS contrôlent totalement leur environnement logiciel grâce à un accès administrateur aux VM. Troisièmement, les grilles et les IaaS proposent un accès aux infrastructures de calculs de manière différente. La grille expose un accès aux machines physiques provenant de différents clusters et donc, intrinsèquement hétérogènes alors que la virtualisation mise en œuvre au sein des clouds IaaS permet de garantir l'homogénéité des performances des machines louées. Enfin, les grilles et les IaaS se distinguent par la manière dont les utilisateurs peuvent spécifier les caractéristiques des machines à utiliser. Sur la grille, les utilisateurs fournissent une spécification minimale alors que les clients du cloud IaaS fournissent une spécification totale. Autrement dit, la grille permet de spécifier la quantité

minimale de ressources (au moins 1 Go de RAM) alors que le cloud IaaS permet de spécifier la quantité exacte de ressources (exactement 1 Go de RAM) facilitant alors la création d'une infrastructure homogène. Enfin, les grilles et les IaaS se différencient par le nombre d'utilisateurs exploitant l'infrastructure. La grille est multi-utilisateur, elle partage son infrastructure entre de nombreux utilisateurs, alors que le cloud IaaS, grâce à la virtualisation, est mono-utilisateur, il permet aux utilisateurs d'être isolés (du moins d'un point de vue logique) et à l'abri des actions des autres utilisateurs.

L'Homme achève au cinquième jour son œuvre, qu'il avait faite : et il se reposa au cinquième jour de toute son œuvre, qu'il avait faite.

1.1 Fonctionnement de l'IaaS

Au sein de l'IaaS, deux entités collaborent : le fournisseur d'IaaS et son client. Le premier s'occupe de l'infrastructure physique. Il s'agit le plus souvent d'un ou plusieurs clusters que le fournisseur propose à la location sous forme de machines virtuelles. Ses objectifs sont de garantir à ses clients une disponibilité élevée tout en minimisant ses dépenses.

Le second, le client, peut utiliser l'IaaS pour de nombreux cas d'utilisations : du site web au calcul intensif en passant par de l'hébergement de SaaS. L'infrastructure mise en place par le client de l'IaaS reçoit alors des requêtes envoyées par des utilisateurs.

Nous attirons votre attention sur un point de vocabulaire important. Dans cette thèse, nous emploierons le terme de *client* pour désigner l'entité qui loue une infrastructure auprès du *fournisseur* d'IaaS. Le terme *utilisateur* sera lui utilisé pour désigner la personne qui utilise les services hébergés sur l'infrastructure louée par le *client*.

1.1.1 Le *provisioning* de ressources

Du point de vue du client, l'utilisation des ressources IaaS est simple. Techniquement, avant de demander une ressource, le client doit préparer et transférer une image personnalisée de VM chez le fournisseur de l'IaaS. Une image de VM correspond à une image disque. Tout comme une machine physique, une VM a besoin des fichiers concernant le système d'exploitation (p. ex. Windows, Mac OS, Linux) ainsi que des fichiers concernant les applications à exécuter. Par exemple, si le client souhaite héberger un site web, son image de VM contiendra typiquement le système d'exploitation Linux, et les applications Apache, MySQL et PHP utiles pour servir un site web aux utilisateurs. C'est l'ensemble de ces fichiers que nous retrouvons dans l'image de VM.

Le *provisioning* est la suite d'actions qui constitue la gestion du cycle de vie des VM, c'est-à-dire l'allocation des ressources, leur surveillance une fois démarrée, et leur arrêt. Après avoir créé une image de VM, puis l'avoir transférée chez le fournisseur d'IaaS, le client réalise les étapes du *provisioning* suivantes :

1. Lorsqu'il en a besoin, le client démarre autant de VM qu'il le souhaite à partir de cette image. Chacune de ces VM exécute le même ensemble de logiciels. Les VM sont alors prêtes à accepter des requêtes des utilisateurs.

2. Lorsqu'il n'en a plus besoin, le client stoppe les VM qu'il souhaite.

Le fournisseur facture le client en fonction du modèle de tarification, de la puissance, du nombre des VM utilisées et de la durée de location.

Gérer les ressources proposées par le fournisseur de l'IaaS implique deux difficultés majeures pour le client :

- tout comme sur la grille, le client de l'IaaS doit dimensionner son application afin de l'exécuter. Autrement dit, le client doit déterminer la taille du découpage de ses requêtes.

- il doit également s'occuper du dimensionnement de la plateforme dynamiquement en fonction de la charge, c'est-à-dire réaliser le *provisioning*.

Si les clients effectuent le processus de *provisioning* manuellement — c'est-à-dire qu'ils décident par eux-mêmes quand et quelles ressources demander et libérer — le risque est d'aboutir à une décision naïve et sous-optimale, particulièrement lorsque le nombre d'offres IaaS est grand, que le nombre de requêtes augmente et que plusieurs types de ressources sont proposées à différents prix (et avec un modèle économique différent) pour des performances différentes. Il est donc important de trouver une solution pour automatiser ce processus ou aider le client à prendre ces décisions.

1.2 Le calcul scientifique

Bien que souvent associé aux applications web, l'IaaS permet l'exécution de nombreux types d'applications, de l'encodage de vidéos à la mise à disposition de logiciels en passant par l'exploration de données.

D'un autre côté, la grille est largement utilisée par la communauté scientifique pour exécuter des calculs. En effet, cette communauté a un besoin fort en puissance de calcul, quels que soient les domaines. Nous pouvons citer l'exemple de la LHC Computing Grid (LCG) qui a été mise en place, principalement, pour réaliser les calculs liés aux expériences des physiciens du CERN sur le Large Hadron Collider présent en Suisse. Cette grille comprend 180 clusters répartis dans 36 pays.

Afin d'exécuter un calcul sur un grand nombre de machines, il est nécessaire de le découper. Il existe différents types d'applications dans un système distribué. Nous pouvons par exemple citer le *bag-of-tasks*, le *workflow* ou le programme parallèle. L'analyse de traces d'exécution de la grille a montré que plus de 85% des exécutions concernent des

programmes séquentiels [27]. Nous appelons *job* une requête d'exécution de tâche de calcul soumise par un utilisateur. Un *bag-of-tasks* correspond à un groupe de jobs, indépendants les uns des autres. Autrement dit, les jobs qui constituent un *bag-of-tasks* peuvent être exécutés dans un ordre quelconque. À l'opposé, le *workflow* correspond à un groupe de jobs liés par une relation de dépendance. Autrement dit, certains jobs doivent être exécutés après d'autres.

L'exécution de calculs scientifiques sur un IaaS est un sujet de recherche très actif. Deelman et al. [14], Montero et al. [46], Song et al. [63] et Villegas et al. [72] ont montré l'utilité de l'IaaS pour différents types d'applications scientifiques. Ce dynamisme dans les activités de recherche souligne l'importance de trouver des solutions pour l'exécution de calculs scientifiques dans les clouds IaaS.

1.3 Contexte de cette thèse

Cette thèse se place dans le contexte de l'exécution de calculs scientifiques séquentiels sur une infrastructure de cloud IaaS. Plus précisément, nous nous intéressons à la réalisation de la décision du *provisioning* automatique de ressource pour la prise en charge de ces calculs dans le but d'aider les clients de l'IaaS. La résolution de ce problème est importante, car la réalisation d'un *provisioning* sous-optimal présente un grand risque de surcout pour le client.

Ce problème est difficile pour deux raisons principales. Premièrement, du point de vue du client, deux métriques contradictoires sont à prendre en compte pour réaliser le *provisioning*. D'un côté, raccourcir le temps d'exécution des jobs nécessite l'utilisation de ressources plus nombreuses et plus puissantes, et donc de payer plus cher. D'un autre côté, un faible cout implique généralement de sacrifier de la puissance de calcul et donc du temps.

Deuxièmement, dans ce contexte, le problème du *provisioning* des ressources est typiquement *online* : lorsque l'utilisateur soumet des jobs, le client doit immédiatement prendre la décision d'utiliser de nouvelles VM ou de réutiliser celles déjà démarrées.

De ces difficultés nait la nécessité de rendre les décisions de *provisioning* automatiques. Nous avons donc conçu une vingtaine de stratégies de *provisioning*. L'utilisateur choisit une stratégie pour l'exécution d'un job. Elle est appelée au moment de la soumission d'un job afin de décider si le nouveau job doit être exécuté sur une nouvelle VM ou sur une VM existante.

Nous présentons en introduction de cette thèse, notre travail de conception de stratégies de *provisioning*. Dans la Section 1.4, nous posons les hypothèses du problème ainsi que son énoncé. Nous présentons dans la Section 1.5 une vingtaine de stratégies de *provisioning* que nous avons conçues. Puis nous évaluons leur efficacité dans la Section 1.6. Enfin, dans la Section 1.7 nous évaluons la possibilité d'exploiter le modèle économique

de l'IaaS afin d'accélérer les calculs à cout constant.

1.4 Énoncé du problème

La conception de nos stratégies de *provisioning* se place dans un cadre précis. Nous présentons dans cette section les hypothèses sur lesquelles nous nous basons puis nous introduisons le problème auquel nous souhaitons répondre.

1.4.1 Hypothèses

Afin de concevoir nos stratégies de *provisioning*, nous posons cinq hypothèses.

Tout d'abord, nous supposons que les jobs à exécuter sont indépendants et que nous connaissons leurs durées. C'est par exemple le cas lorsque des utilisateurs soumettent leurs jobs à un système de gestion des ressources sur la grille qui nécessite que l'utilisateur fournisse un temps d'exécution maximal.

Deuxièmement, nous supposons que les jobs ne sont pas préemptibles. C'est-à-dire qu'un job en cours d'exécution ne peut ni être interrompu ni migré.

Troisièmement, nous supposons que chaque utilisateur possède sa propre VM. En d'autres termes, un job soumis par un utilisateur ne peut être exécuté sur la VM d'un autre utilisateur.

Quatrièmement, nous nous plaçons dans le cadre d'un système d'ordonnancement *online* ou *semi-online*. Les jobs sont ordonnancées dynamiquement dès leur soumission. Dans un cadre *semi-online*, les jobs transitent dans des files d'attente. Leur assignation finale à une ressource est calculée régulièrement.

Enfin, dans ce travail, nous nous concentrons sur un modèle économique spécifique, mis initialement en place par Amazon pour sa plateforme EC2. Ce modèle, appelé *on-demand* par Amazon, facture l'utilisateur à l'heure d'utilisation de VM. De nombreux autres fournisseurs ont maintenant adopté ce modèle économique tel que GoGrid, OVH, ou Rackspace. Dans ce modèle, nous appelons la période élémentaire de temps facturée la Billing Time Unit (BTU). Il présente deux caractéristiques intéressantes pour le *provisioning*. Tout d'abord, le cout de déploiement est linéaire, indépendamment du nombre de VM louées. Ainsi, louer une VM pendant deux heures est aussi couteux que de louer deux VM pendant une heure. Deuxièmement, la granularité de la BTU est grossière (une heure), et chaque heure entamée est due. Par conséquent, un job dont la durée n'est pas exactement une BTU laissera, sur la VM déployée, du temps payé, mais non utilisé. Ce temps inutilisé peut être recyclé pour exécuter d'autres jobs.

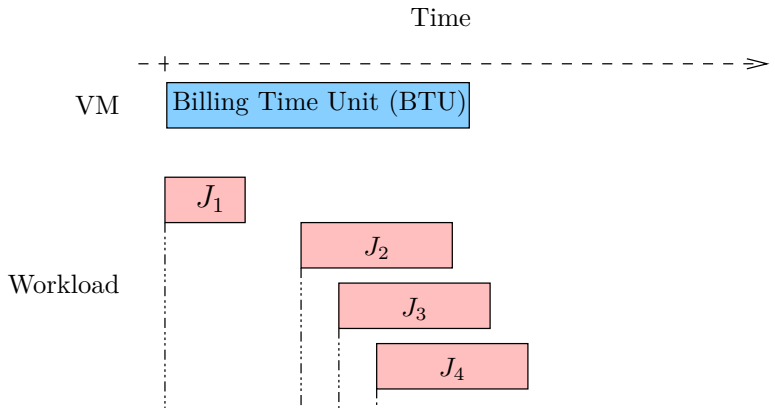


FIGURE 1.1 – Exemple de workload

1.4.2 Problème

Nous introduisons le problème à partir du workload présenté sur la Figure 1.1. Dans cet exemple, le workload est composé de quatre jobs soumis au cours du temps. Nous montrons, en haut de cette figure, la taille d'une BTU. Afin d'exécuter le workload, plusieurs solutions sont possibles. Une solution est d'instancier une seule VM et d'y exécuter les jobs séquentiellement dans l'ordre dans lequel ils ont été soumis. Des solutions alternatives existent-elles afin de réserver simultanément plusieurs ressources pour exécuter certains jobs en parallèle ? À quel cout ? Et pour quelle amélioration des performances ?

Une des solutions consiste à associer ce problème à celui du *bin packing de taille variable à cout constant* [21]. Dans ce problème, nous avons une infinité de boites de taille variable. L'objectif est de ranger dans ces boites des objets de taille variable, mais définie, en choisissant les boites afin de minimiser la taille totale des boites. Il s'agit d'une généralisation du problème de bin packing dans lequel toutes les boites ont une taille fixe S et chaque objet a une taille $s_i < S$. Ces deux problèmes sont NP-difficiles, mais de nombreuses heuristiques existent [13]. Dans notre contexte, les objets sont les jobs à exécuter et les boites sont les BTU des VM. Cependant, les solutions proposées pour ce problème ne peuvent pas être directement appliquées. Tout d'abord, plusieurs de ces heuristiques ne sont pas utilisables à cause de la contrainte *online* que nous avons. Par exemple, les algorithmes de type *FitDecrease* nécessitent de trier les objets par taille et requièrent donc d'avoir l'ensemble des objets à priori. Ensuite, le problème classique de bin packing ne prend pas en compte le côté temporel de notre problème de *provisioning* de ressources. En effet, une boite a toujours la même taille restante tant qu'aucun objet n'y a été ajouté. Au contraire, une VM est, par analogie, constamment remplie depuis sa création jusqu'à sa destruction, que nous y affections des jobs ou non. De plus, comme expliqué en introduction, notre problème a deux objectifs. Nous ne souhaitons pas seulement minimiser le nombre de BTU utilisées, mais également le temps d'achèvement des jobs. Finalement, nous n'adapterons que les heuristiques du problème de bin packing qui s'applique au cas *online* dans le but de minimiser le cout du *provisioning*.

1.5 Des stratégies de *provisioning*

1.5.1 L'algorithme commun

Afin de modéliser l'état du système, nous avons principalement besoin de prendre en compte, à chaque instant, les VM que nous avons démarrées. Nous maintenons également une file d'attente de jobs assignés à chaque VM. Les notations utilisées sont :

- V : Ensemble des VM démarrées
- q_v : File des jobs des $v \in V$ (*queue*)
- b_v : Date de démarrage des $v \in V$ (*boot*)
- s_v : Date de fin des $v \in V$ (stop)
- i_v : Date quand $v \in V$ devient inoccupée (*idle*). C'est-à-dire lorsque q_v est vide.
- J : Ensemble des jobs
- r_j : Temps d'exécution de $j \in J$ en secondes (*runtime*)
- w_j : Temps d'attente de $j \in J$ en secondes (*wait time*)
- $c(x)$: Cout d'une VM pour x secondes d'activité. Dans le cas d'Amazon EC2, $c(x) = pph \times \lceil \frac{x}{BTU} \rceil$ avec $BTU = 3\,600$ s et pph son cout horaire (*price per hour*)
- C : Ensemble des VM candidates pour l'exécution d'un job ($C \subset V$).

Dans ce modèle, les dates sont exprimées en secondes.

Les stratégies que nous proposons peuvent être exprimées par des algorithmes partageant une structure commune. Ces algorithmes ont deux phases principales :

1. Une phase de *déploiement* appelée à chaque soumission de job. Cette phase décide (1) si une nouvelle VM doit être démarrée ou non, et (2) à quelle VM active le job doit être assigné. Cette phase est décrite dans l'Algorithme 1.

2. Une phase de *libération* des ressources déclenchée à la fin de la BTU d'une VM. Cette phase n'est pas spécifique aux stratégies. Elle consiste à décider quelles VM actives doivent être stoppées. Chaque VM démarrée est examinée. Si une VM est inoccupée à la fin de la BTU, elle est stoppée. Cette phase est décrite dans l'Algorithme 2.

Algorithme 1 deploy(j,t)

// a new job j is submitted, at date t
$C \leftarrow \emptyset$ // C is the set of candidate VM ($C \subset V$)
for $v \in V$ **do**
  if $eligible(v, j)$ **then**
    $C \leftarrow C \cup \{v\}$
  end if
end for
if $C \neq \emptyset$ **then**
  $v \leftarrow optimum(C)$
else
  $v \leftarrow deploy()$ // Create and run a new VM
  $V \leftarrow V \cup \{v\}$
end if
$enqueue(q_v, j)$ // Map the job to the VM

L'Algorithme 1 utilise les fonctions suivantes :

- $eligible(v, j)$ est vrai si j peut être assigné à $v \in V$,
- $optimum(C)$ retourne la VM à laquelle le job j doit être assigné,
- $deploy()$ démarre une nouvelle VM,
- $enqueue(q_v, j)$ ajoute le job j à la file d'attente q_v.

Les fonctions *eligible* et *optimum* nous permettent de décrire l'ensemble de nos stratégies de *provisioning*. Dans un premier temps, *eligible* filtre l'ensemble des VM actives V en fonction de l'état actuel du système. Si cet ensemble est vide, alors une nouvelle VM est déployée. Dans le cas contraire, *optimum* choisit la VM sur laquelle sera ordonnancé le job.

Algorithme 2 release(v)

At the end of the BTU // Release action at date t
for $v \in V$ **do**
  if $empty(q_v)$ **then**
    $V \leftarrow V - \{v\}$
  end if
end for

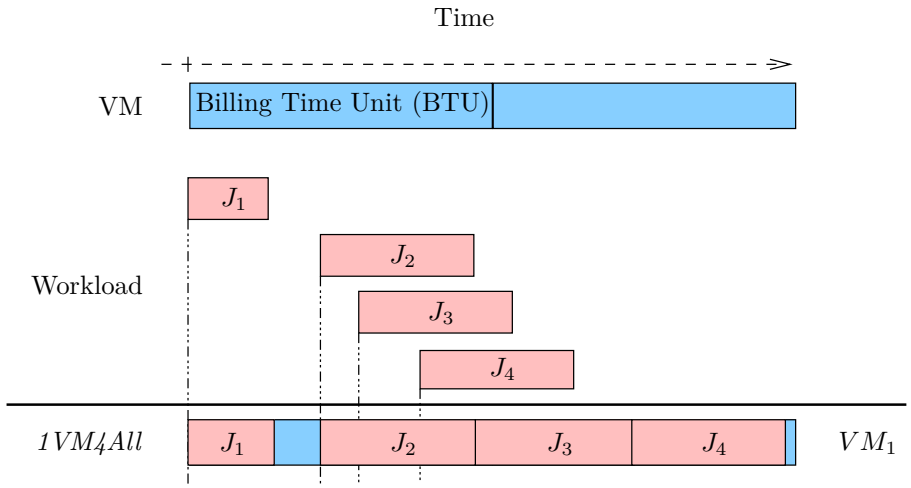


FIGURE 1.2 – Exemple d'application de la stratégie *1VM4All*

TABLE 1.1 – Les paramètres pour l'Algorithme 1 de *1VM4All*

1VM4All	$eligible(v, j)$ returns *true*	always
	$optimum(C)$ returns $v \in C$ such that ...	$v = v_0$
	comment	Slowest/Cheapest

1.5.2 Description des stratégies

Nous proposons quatre familles de stratégies. Nous les décrivons dans cette section au travers de quelques exemples d'applications concrets. La définition des fonctions *eligible* et *optimum* pour chaque stratégie est récapitulée dans l'Annexe B.

1VM4All

La première stratégie provisionne une seule VM et ajoute tous les jobs séquentiellement dans sa file d'attente. Cette stratégie fournit une *borne minimale du cout* pour un workload donné. En effet, le temps d'inactivité de la VM est réutilisé au maximum. Nous présentons dans la Table 1.1 les paramètres pour l'Algorithme 1 pour cette stratégie.

Sur la Figure 1.2, nous présentons un exemple d'exécution de *1VM4All*. Nous montrons que le temps d'attente entre la soumission de chaque nouveau job et son exécution augmente. En revanche, les temps d'inactivité de la VM sont bien réduits au maximum.

Stratégies de type *1VMperJob*

À l'opposé de *1VM4All*, nous avons conçu quatre stratégies couteuses. *1VMperJob* est la stratégie de référence qui minimise le temps d'attente. Elle déploie une nouvelle VM à chaque soumission d'un job, quel que soit l'état des autres VM actives dans le but de *minimiser le temps d'attente* entre la soumission d'un job et son exécution.

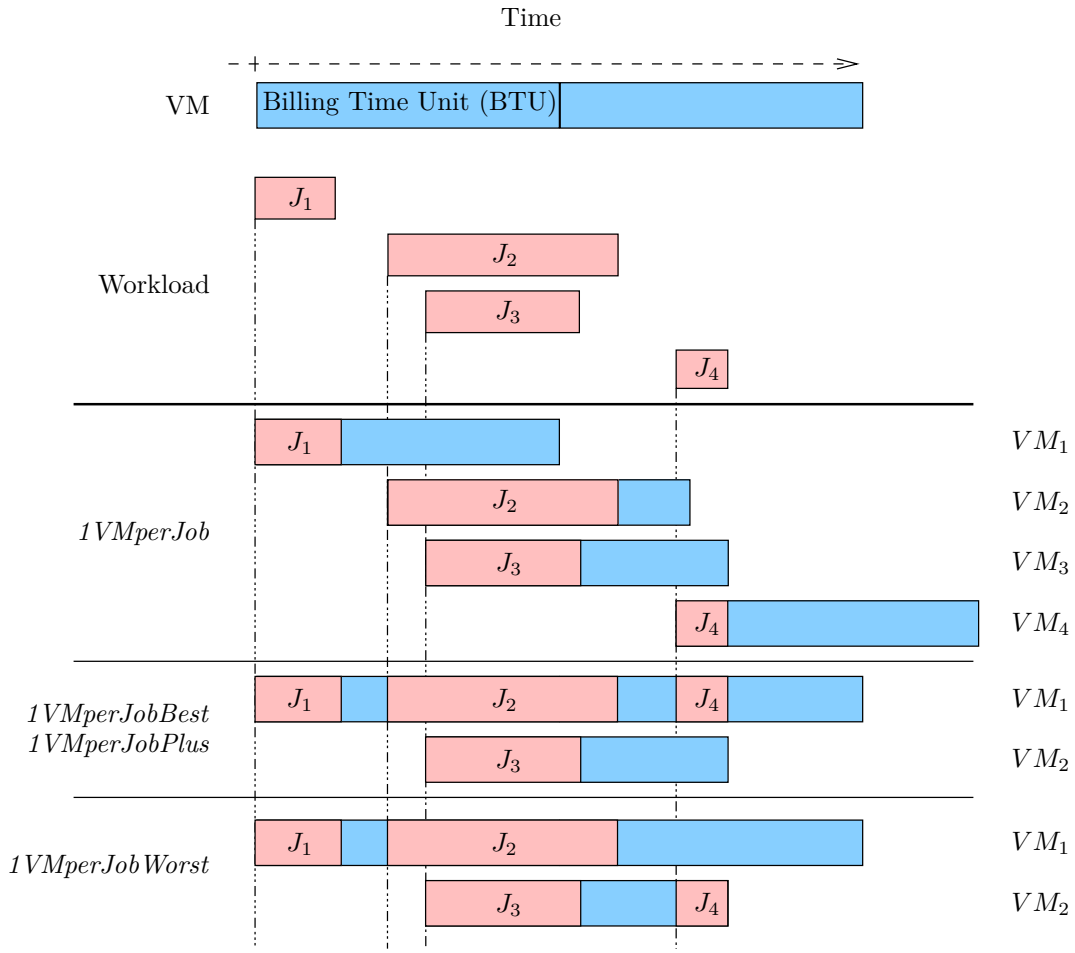


FIGURE 1.3 – Exemple d'application des stratégies de type *1VMperJob*

TABLE 1.2 – Les paramètres pour l'Algorithme 1 des stratégies de type *1VMperJob*

1VMperJob	*eligible*(v, j) returns *true*	never
	optimum(C) returns $v \in C$ such that ...	any
	comment	Most expensive Minimize wait time
1VMperJobPlus	*eligible*(v, j) returns *true*	if $q_v = \emptyset$
	optimum(C) returns $v \in C$ such that ...	any
1VMperJobBest	*eligible*(v, j) returns *true*	if $q_v = \emptyset$
	optimum(C) returns $v \in C$ such that ...	s_v is maximum
1VMperJobWorst	*eligible*(v, j) returns *true*	if $q_v = \emptyset$
	optimum(C) returns $v \in C$ such that ...	s_v is minimum

Nous représentons sur la Figure 1.3 les différentes stratégies de type *1VMperJob*. Nous voyons bien que la stratégie *1VMperJob* déploie de nombreuses VM en parallèle. En contrepartie, les temps d'inactivité des VM sont nombreux et le cout augmente de fait considérablement par rapport à *1VM4All*.

Une amélioration évidente de *1VMperJob* pour le job J_2 est de ne déployer une nouvelle VM que si aucune autre VM n'est inactive au moment de la soumission du job. Ainsi, J_2 sera affecté à la VM_1. *1VMperJobBest*, *1VMperJobWorst* et *1VMperJobPlus* implémentent cette amélioration. Ces trois stratégies se distinguent par le choix de la VM qu'elles font :

***1VMperJobPlus* :** retourne la première VM inactive trouvée, quel que soit l'ordre dans lequel les VM sont rangées dans V. Nous imaginons ici qu'il est difficile de prédire quelle sera la VM la plus intéressante à utiliser et en choisissons donc arbitrairement une. Sur l'exemple de la Figure 1.3, nous supposons que la VM_1 est le premier élément de la liste V.

***1VMperJobBest* :** retourne la VM inactive qui sera stoppée le plus tard. L'objectif est de maximiser le temps d'inactivité restant à la fin de l'exécution du job. En effet, plus il y a de temps d'inactivité sur les VM, plus il y a d'opportunités pour les soumissions futures de trouver une VM sur laquelle s'exécuter à cout constant. Sur la Figure 1.3, lorsque le job J_4 est soumis, la VM_1 et la VM_2 sont inactives et peuvent exécuter ce job. Cette stratégie sélectionne la VM_1.

***1VMperJobWorst* :** retourne la VM inactive qui sera stoppée le plus tôt. L'objectif est de minimiser le temps d'inactivité restant à la fin de l'exécution du job et ainsi réduire le temps payé, mais inutilisé. Sur la Figure 1.3, le job J_4 est donc affecté à la VM_2 et aucun temps d'inactivité ne restera sur cette VM.

Le principal problème des stratégies de type *1VMperJob* est le cout que cette stratégie peut engendrer. Nous montrons sur la Figure 1.4 un cas pathologique pour ces stratégies. Dans cet exemple caricatural, des jobs très courts et d'autres un peu plus longs qu'une BTU sont soumis à la même date, *1VMperJobPlus* déploiera autant de VM que de jobs. Le temps d'inactivité de toutes ces VM sera alors très grand, alors qu'en ordonnançant J_3 et J_4 à la suite de J_1 et J_2, nous obtiendrions un temps d'attente légèrement plus grand pour un cout plus faible.

Stratégie de type *bin packing*

Les deux familles de stratégies proposées précédemment — *1VM4All* et *1VMperJob* — sont respectivement des bornes minimales pour le cout et le temps d'attente. Nous proposons également des stratégies inspirées du problème de bin packing avec pour objectif

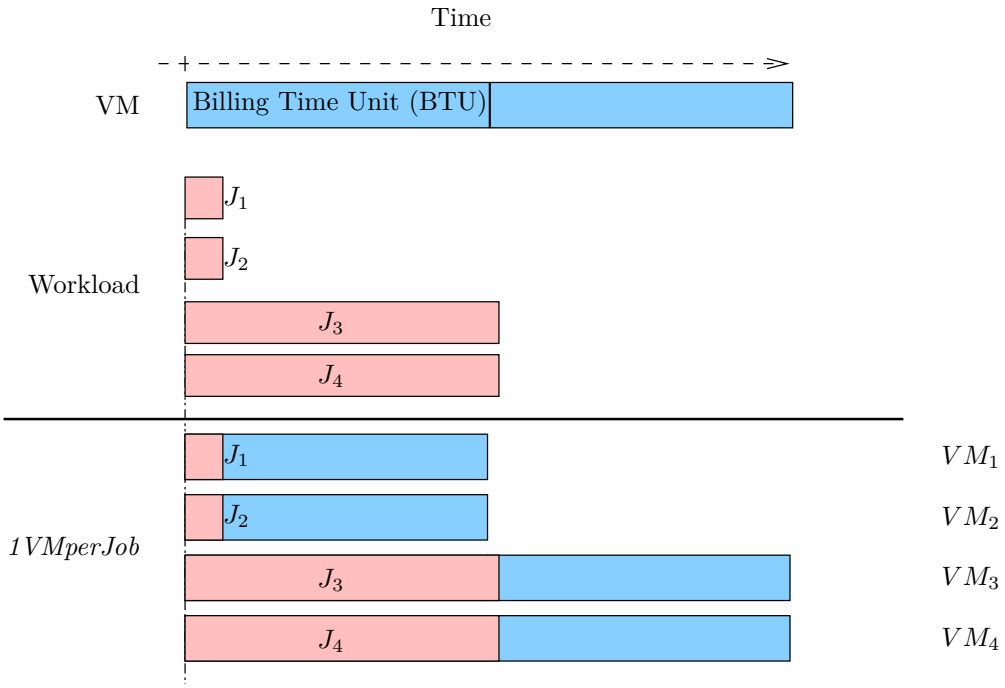


FIGURE 1.4 – Exemple de solution sous-optimale trouvée par *1VMperJob*

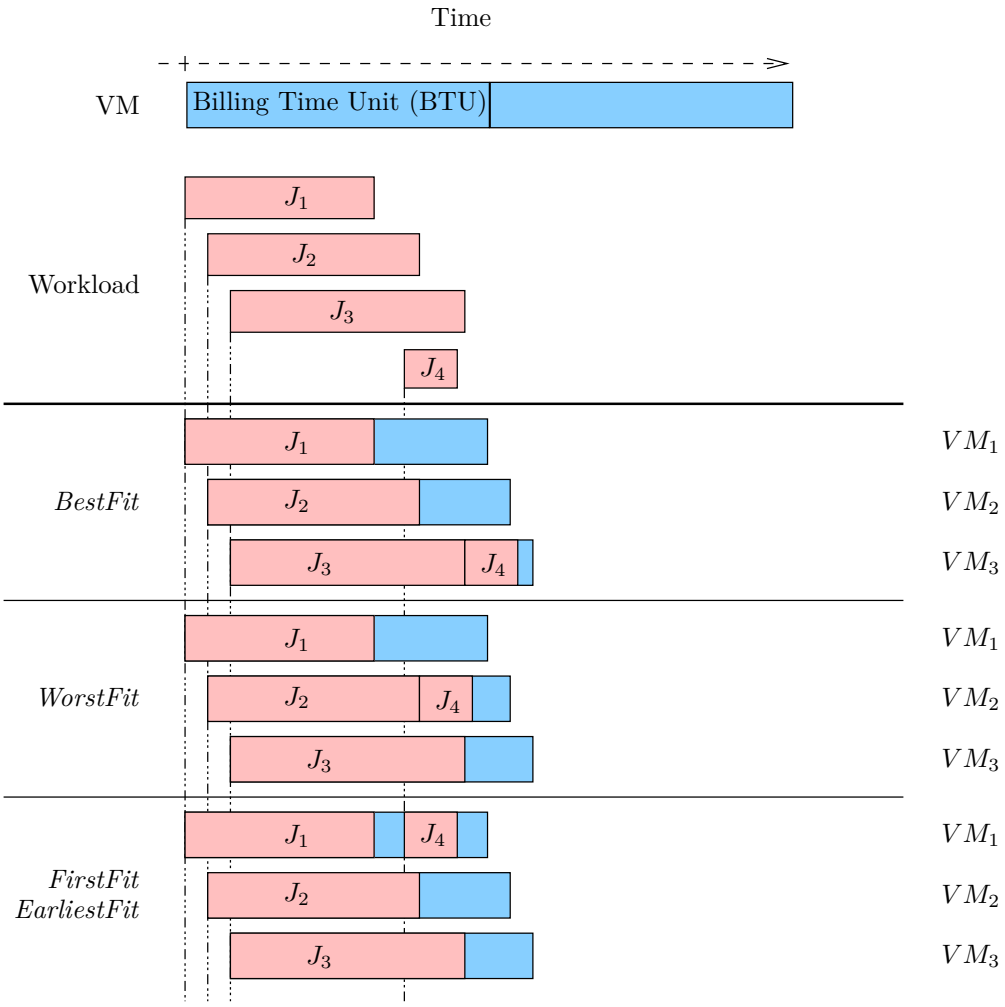


FIGURE 1.5 – Exemple d'application des stratégies de bin packing

TABLE 1.3 – Les paramètres pour l'Algorithme 1 des stratégies de bin packing

FirstFit	*eligible*(v, j) returns *true*	if $c(s_v - b_v) = c(s_v - b_v + r_j)$
	optimum(C) returns $v \in C$ such that ...	any
	comment	Regular bin packing strategy
BestFit	*eligible*(v, j) returns *true*	if $c(s_v - b_v) = c(s_v - b_v + r_j)$
	optimum(C) returns $v \in C$ such that ...	$i_v - s_v$ is maximum
	comment	Regular bin packing strategy to minimize idle time
WorstFit	*eligible*(v, j) returns *true*	if $c(s_v - b_v) = c(s_v - b_v + r_j)$
	optimum(C) returns $v \in C$ such that ...	$i_v - s_v$ is minimum
	comment	Regular bin packing strategy to maximize idle time
EarliestFit	*eligible*(v, j) returns *true*	if $c(s_v - b_v) = c(s_v - b_v + r_j)$
	optimum(C) returns $v \in C$ such that ...	i_v is minimum
	comment	+ wait time optimization

de proposer un compromis entre nos deux bornes : réduire le temps d'attente qu'implique *1VM4All* en limitant l'augmentation du cout qui résulte de l'utilisation de *1VMperJob*.

Trois heuristiques classiques du problème de bin packing [13] ont été implémentées : *FirstFit*, *BestFit* et *WorstFit*. Dans notre contexte, chacune de ces stratégies a pour objectif de sélectionner une VM existante sur laquelle l'exécution du job soumis n'entraînera pas l'ouverture d'une nouvelle BTU. Autrement dit, ces heuristiques ont pour objectif d'exécuter le job à cout constant. Si aucune VM correspondant à ce critère n'existe, nous en déployons une nouvelle.

Nous représentons sur la Figure 1.5 les différentes stratégies de bin packing. Prenons l'exemple de *BestFit*. Sur cette figure, on constate que le job J_2 ne peut être ordonnancé sur la VM_1 à la suite du job J_1 sans nécessiter d'entamer une deuxième BTU. Les stratégies de bin packing démarrent donc une deuxième VM pour l'exécuter. De même pour le job J_3 qui ne peut être exécuté ni sur la VM_1, ni sur la VM_2. Le job J_4 peut être exécuté sur les trois VM démarrées. Ces trois heuristiques diffèrent seulement dans la VM qu'elles choisissent pour exécuter ce dernier job :

***BestFit* :** scanne la liste des VM candidates et affecte le job sur celle qui laissera le plus petit temps d'inactivité. L'objectif est de minimiser le temps restant d'inactivité. Sur la Figure 1.5, c'est la VM_3 qui permet d'exécuter J_4 en laissant le plus petit temps d'inactivité.

***WorstFit* :** scanne la liste des VM candidates et affecte le job sur celle qui laissera le plus grand temps d'inactivité. L'objectif est de maximiser le temps restant d'inactivité.

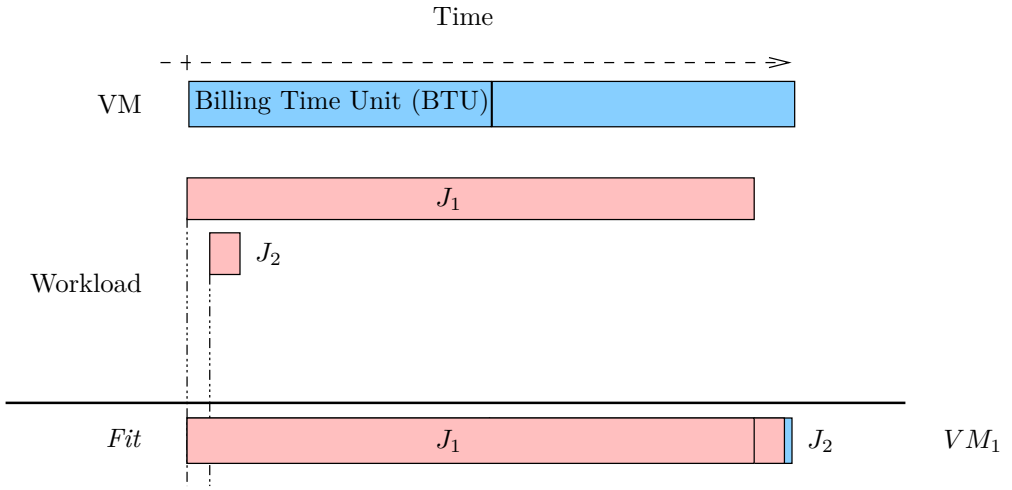


FIGURE 1.6 – Exemple de cas que les stratégies *Fit* ne gèrent pas correctement

Sur la Figure 1.5, c'est la VM_2 qui permet d'exécuter J_4 en laissant le plus grand temps d'inactivité.

***FirstFit* :** affecte le job à la première VM de la liste des VM candidates. Sur la Figure 1.5, nous supposons que la VM_1 est le premier élément de la liste C des VM candidates.

Ces trois stratégies de bin packing ont pour objectif de minimiser le nombre de BTU. Ainsi elles tendent à minimiser le cout global tout en ménageant le temps d'achèvement de l'exécution qui n'existe pas dans le problème original de bin packing. Nous montrons dans la Figure 1.6 un exemple de cas que ces stratégies ne gèrent pas correctement. Dans ce cas, un job court (J_2) arrive immédiatement après un job long (J_1). Il est exécutable à la fin de J_1 sans entamer une autre BTU et est donc ordonnancé sur la VM_1. Le temps d'attente pour le job J_2 est par conséquent très grand.

Ainsi, nous proposons une quatrième stratégie de type *Fit* qui est une première approche pour inclure ce critère :

***EarliestFit* :** sélectionne la VM qui minimise le temps d'attente du job. Sur la Figure 1.5, seule la VM_1 permet d'exécuter J_4 sans aucun temps d'attente.

Stratégies de type *Relax*

Les stratégies de bin packing réduisent le cout global, mais impliquent des temps d'attente qui peuvent être importants. *RelaxFirstFit*x, *RelaxEarliestFit*x et *RelaxLastestFit*x ajoutent une borne au temps d'attente d'un job, exprimée comme un facteur de x. Une nouvelle VM est déployée quand aucune VM démarrée ne peut prendre en charge un job à cout constant ou lorsque le temps d'attente qu'implique l'exécution sur une VM existante dépasserait x fois le temps d'exécution du job. Comme le montre l'exemple sur la Figure 1.7, une valeur basse de x ($x = 0.5$) entraîne un comportement similaire à *1VMperJobPlus*. Ici, une nouvelle VM est déployée pour J_2 puisqu'utiliser une VM déjà déployée

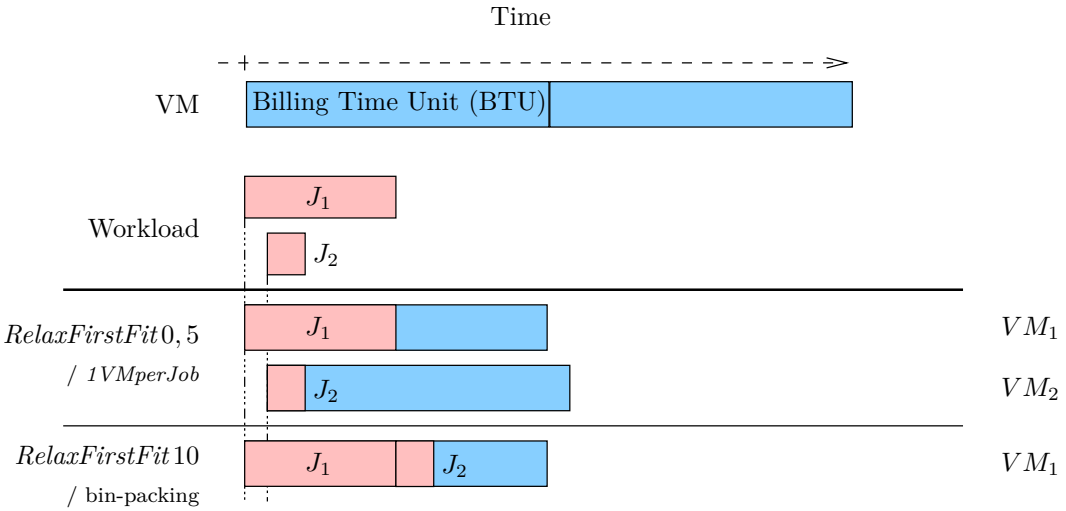


FIGURE 1.7 – Exemple d'application des stratégies de type Relax

TABLE 1.4 – Les paramètres pour l'Algorithme 1 des stratégies de type Relax

RelaxFirstFit x	*eligible*(v, j) returns *true*	if $c(s_v - b_v) = c(s_v - b_v + r_j)$ and $(i_v - t) < (x \times r_t)$
	optimum(C) returns $v \in C$ such that ...	any
	comment	Cost optimization
RelaxEarliestFit x	*eligible*(v, j) returns *true*	if $c(s_v - b_v) = c(s_v - b_v + r_j)$ and $(i_v - t) < (x \times r_t)$
	optimum(C) returns $v \in C$ such that ...	i_v is minimum
	comment	+ max wait time constraint
RelaxLastestFit x	*eligible*(v, j) returns *true*	if $c(s_v - b_v) = c(s_v - b_v + r_j)$ and $(i_v - t) < (x \times r_t)$
	optimum(C) returns $v \in C$ such that ...	i_v is maximum
	comment	+ min wait time constraint

TABLE 1.5 – Caractéristiques des workloads

Trace	#jobs	#CPUs	#utilisateurs	inter-arrivé	walltime
LCG	188 041	24 115	216	5	8 970
AuverGrid	347 611	475	405	78	25 186
NorduGrid	781 370	2 000	387	127	89 273
SharcNet	1 195 242	6 828	412	28	31 964

impliquerait une augmentation du temps d'attente de l'ordre de trois fois le temps d'exécution de J_2. Au contraire, une valeur de x haute ($x = 10$) entraîne un comportement similaire aux stratégies de type bin packing puisque le même temps d'attente est considéré comme acceptable.

Nos trois heuristiques diffèrent seulement dans la VM qu'elles sélectionnent :

RelaxFirstFit x **:** affecte le job à la première VM de la liste des VM candidates.

RelaxEarliestFit x **:** affecte le job à la VM candidate qui sera inactive en premier.

RelaxLastestFit x **:** affecte le job à la VM candidate qui sera inactive en dernier.

Ces stratégies nous permettent donc, grâce au facteur x, de jouer sur le compromis entre le cout et les performances.

1.6 Évaluation

Afin d'évaluer les performances des différentes stratégies, nous avons utilisé quatre workloads correspondant à des exécutions typiques de calculs scientifiques. Ces workloads sont disponibles publiquement sur la Grid Workload Archive [27] et correspondent à des exécutions sur des grilles en production. Notre choix s'est porté sur des workloads de grille car aucun workload de cloud n'existait publiquement au moment de nos expériences. Toutes nos stratégies ont été implémentées dans un simulateur ad hoc. Dans cette section, nous présenterons les workloads ainsi que le simulateur, puis les résultats de l'évaluation.

1.6.1 La Grid Workload Archive

Les caractéristiques principales des workloads sélectionnés sont présentées dans la Table 1.5 : le nombre total de jobs, le nombre de CPU utilisé, le nombre d'utilisateurs, le temps moyen d'inter-arrivé entre les jobs et le temps d'exécution moyen des jobs. Tous les temps sont en secondes.

La première trace correspond à des exécutions sur LCG. Il s'agit d'un centre de stockage et de calcul pour la communauté des physiciens qui utilisent le Large Hadron Collider du CERN. Cette grille en production possède 180 sites avec près de 30 000 CPU. Les

traces collectées incluent seulement les calculs sur les données de physique des particules. Onze jours d'activité à compter du 20 novembre 2005 ont été enregistrés.

La deuxième correspond à AuverGrid, une grille multi site qui fait partie du projet EGEE. Cette grille est principalement utilisée pour des applications biomédicales et de physique des particules. Les traces correspondent à une année d'activité à partir de janvier 2006.

La troisième, NorduGrid, est une grille à destination des chercheurs académiques suédois composée de plus de 75 clusters principalement académiques, mais également en provenance de l'industrie. Les applications qui y sont exécutées proviennent principalement des domaines de la chimie, de la biomédecine et de la physique des particules. Les traces correspondent à trois années d'activité à partir de mars 2003.

La dernière correspond à SharcNet, un consortium des institutions académiques canadiennes qui partagent un réseau d'ordinateurs haute-performance. Les traces correspondent à une année d'activité à partir de décembre 2005.

Les données communes entre ces traces sont, pour chaque job, son identifiant, la date de soumission, le temps d'exécution, l'identifiant de l'utilisateur et du groupe auquel il appartient.

Ces traces sont hétérogènes et représentatives des calculs exécutés sur la grille.

1.6.2 Le simulateur

Le simulateur utilisé pour l'évaluation a été développé au sein de l'équipe avec pour unique objectif de réaliser cette tâche. Il s'agit d'un simulateur à évènement discret développé en Python. Les évènements qui sont utilisés ici sont la soumission d'un job et la fin d'une BTU. Il ne prend pas en compte le réseau, la couche de virtualisation ou les temps de démarrage des VM. Il implémente les files d'attente des VM et la structure de l'algorithme commun à toutes nos stratégies tel que décrit dans la Section 1.5.1.

1.6.3 Évaluation par la simulation

Les résultats de la simulation des différentes stratégies de *provisioning* sont montrés dans la Figure 1.8 et les Figure 1.10, 1.11, 1.12 et 1.13. Nous y présentons deux métriques.

Nous définissons d'abord le temps d'attente d'un job comme étant le temps entre sa soumission et le début de son exécution sur la VM. Il n'inclut pas le temps de démarrage des VM qui n'est pas pris en charge par notre simulateur. La première métrique que nous utilisons est le temps d'attente moyen w défini comme le total des temps d'attente divisés par le nombre total de jobs. Elle est formellement définie dans l'Équation 1.1.

$$w = \frac{\sum_{j \in J} w_j}{|J|} \tag{1.1}$$

La deuxième métrique que nous utilisons est le prix total des ressources défini comme la somme des couts de chacune des ressources. Elle est formellement définie dans l'Équation 1.2.

$$p = \sum_{v \in V} c(s_v - b_v) \tag{1.2}$$

Résultats sur les traces complètes

Dans un premier temps, nous considérons que chaque trace correspond aux soumissions d'un groupe d'utilisateurs à un même système de courtage. Les VM peuvent être réutilisées par tous ces utilisateurs. Les résultats de la simulation d'un tel système sont montrés dans la Figure 1.8.

Nous voyons sur cette figure que les résultats sont similaires sur les quatre plateformes : LCG, AuverGrid, NorduGrid et SharcNet. Comparons les exécutions en utilisant *1VM4All* qui a pour objectif de minimiser le cout et celles en utilisant les stratégies basées sur *1VMperJob* qui ont pour objectif de minimiser le temps d'attente. Ces dernières parviennent à réduire le temps d'attente à 0 s tout en limitant l'augmentation du cout. Par exemple, *1VMperJobPlus* augmente le cout de respectivement 3.5%, 2.5%, 1.1%, et 3.2% pour les quatre workloads. Parmi les stratégies basées sur *1VMperJob*, *1VMperJobBest* est légèrement meilleure que les autres. Elle parvient à limiter l'augmentation du cout à respectivement 2,7%, 2,2%, 1,0%, et 3,0% tout en réduisant le temps d'attente à 0 s.

Ces résultats indiquent qu'il est difficile d'inventer des stratégies de *provisioning* proposant un compromis entre l'optimisation du cout et du temps d'attente des jobs. Une stratégie simple telle que *1VMperJobPlus* permet d'obtenir un temps d'attente nul pour un cout proche du minimum.

Résultats sur les traces individuelles des utilisateurs

Si les stratégies de bin packing ne semblent pas utiles dans le cas des traces complètes, nous souhaitons vérifier si ce résultat est généralisable sur de plus petites traces. Dans un second temps, nous évaluons les stratégies en utilisant les soumissions des utilisateurs extraites de la trace de LCG. Chaque simulation correspond ainsi aux soumissions d'un utilisateur unique à son propre système de courtage. Les utilisateurs ne partagent donc pas entre eux les VM.

Nous avons constaté que les utilisateurs peuvent être regroupés en fonction de la marge de cout entre l'utilisation de *1VMperJobPlus* et de *1VM4All*. Nous définissons cette marge dans l'Équation 1.3. Elle représente l'espace disponible pour trouver un compromis entre la performance, en réduisant le temps d'attente des jobs, et le cout. Nous définissons cette marge de cout comme étant la marge entre le cout de la stratégie la plus couteuse et le cout de la stratégie la moins couteuse. Une marge de 1 signifie qu'il n'y a pas de

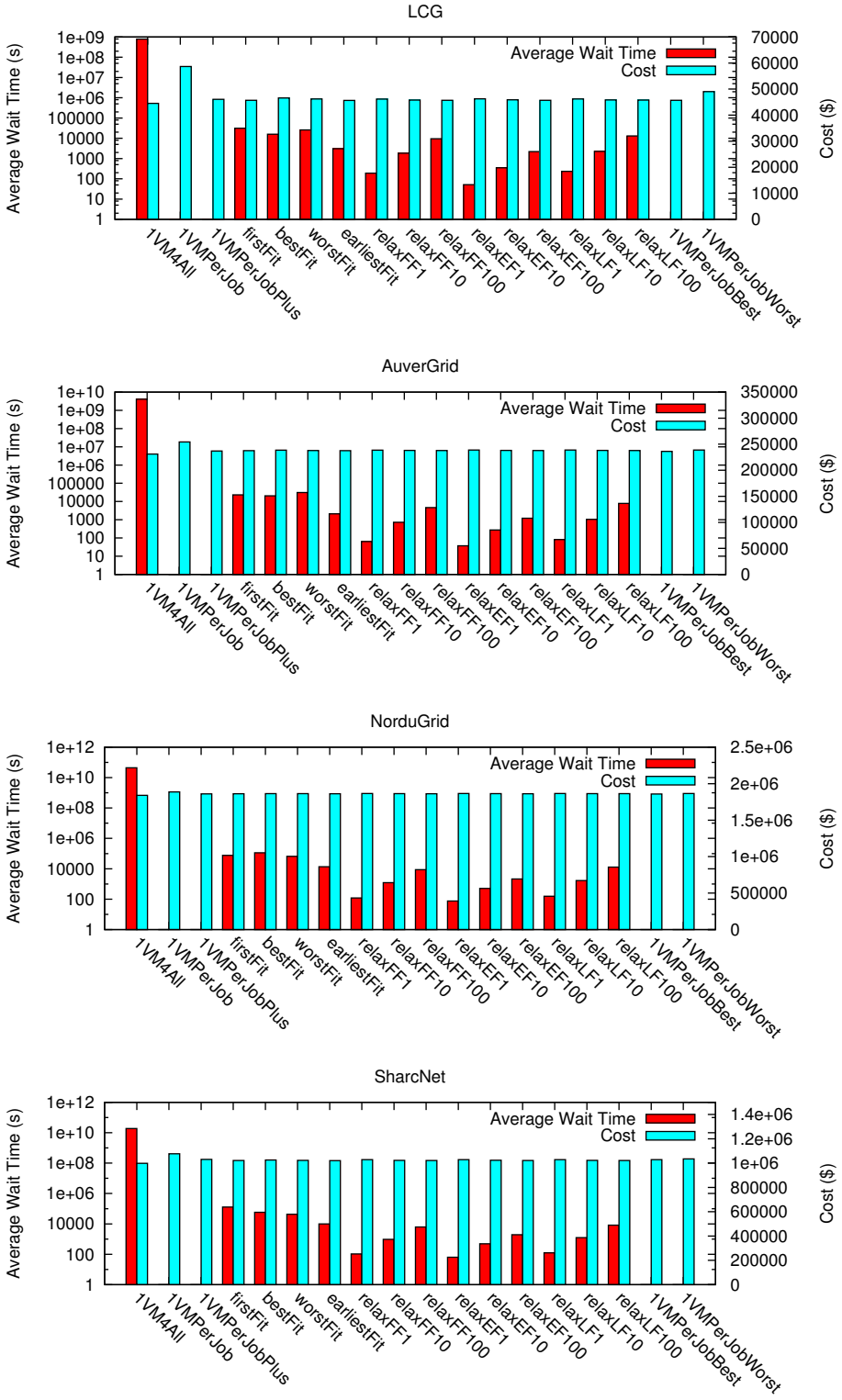


FIGURE 1.8 – Temps d'attente moyen (s) et cout ($), pour chaque stratégie, sur les traces complètes

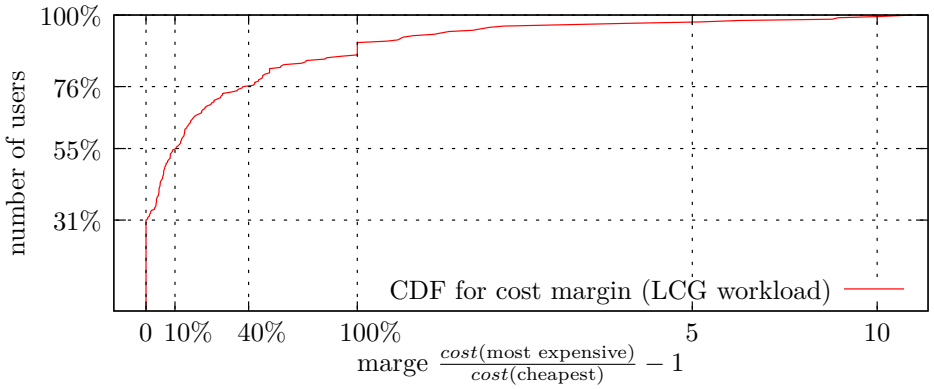


FIGURE 1.9 – Fonction de répartition cumulative des marges de cout sur les traces des utilisateurs

compromis possible puisque la stratégie la plus couteuse et la moins couteuse ont le même cout.

$$\text{marge} = \frac{\text{cost}_{1VMperJobPlus}}{\text{cost}_{1VM4All}} - 1 \tag{1.3}$$

Nous présentons dans la Figure 1.9 la fonction de répartition cumulative des marges de cout des utilisateurs. Un point en (40% , 76%) signifie que pour 76% des utilisateurs, la marge de cout est inférieure ou égale à 40%.

Nous avons arbitrairement découpé la trace LCG en quatre groupes d'utilisateurs en fonction de leurs marges de cout. Pour chacun de ces groupes, nous avons sélectionné un utilisateur représentatif dont nous présentons le cout et le temps d'attente dans les figures 1.10, 1.11, 1.12 et 1.13.

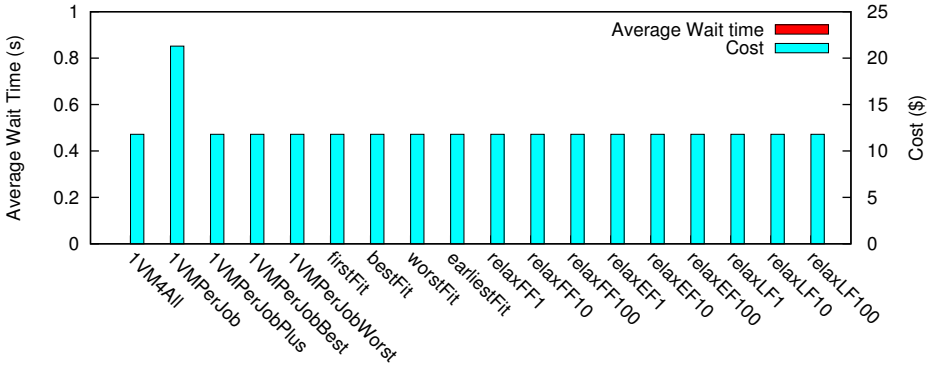


FIGURE 1.10 – Temps d'attente moyen (s) et cout ($), pour chaque stratégie, sur la trace de l'utilisateur LCG 96 (marge = 0,00)

Utilisateurs sans marge de cout 31% des utilisateurs n'ont aucune marge de cout (marge = 0). C'est le cas de l'utilisateur 96 de la trace LCG que nous présentons dans la Figure 1.10. Dans ce cas, il n'y a aucune différence entre le cout de la stratégie la moins couteuse *1VM4All* et celui de la stratégie la plus rapide *1VMperJobPlus*. De plus, il n'y

a pas non plus de différence en terme de temps d'attente pour la plupart des utilisateurs. Ce cas arrive dans des situations particulières lorsqu'il n'y a pas de jobs concurrents, ou que les jobs ont des temps d'exécution supérieurs à une BTU. Ainsi, toutes les stratégies agissent de la même façon. Dans de rares cas, nous constatons une augmentation du temps d'attente pour *1VM4All* ou du cout pour *1VMperJob*. Par conséquent, il est recommandé d'utiliser dans ce cas la stratégie *1VMperJobPlus*.

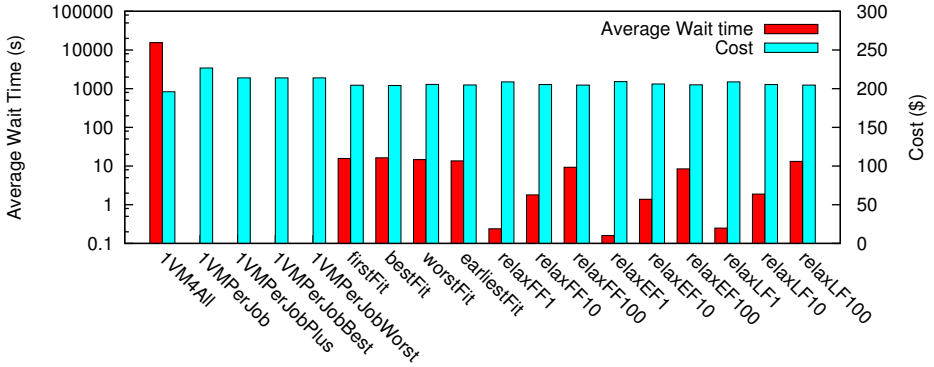
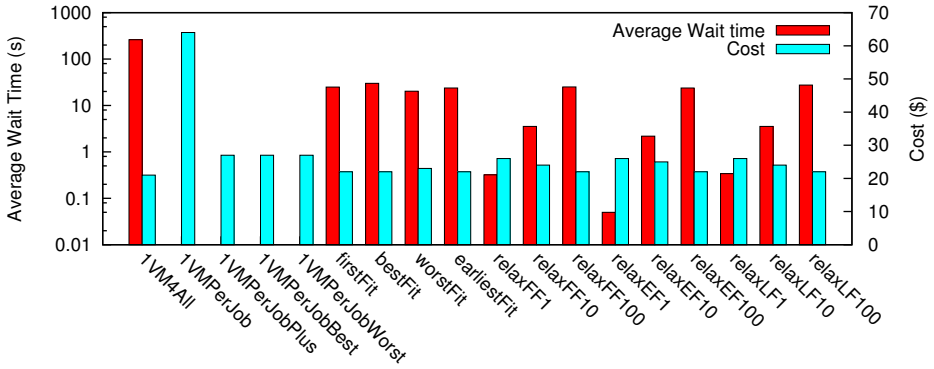


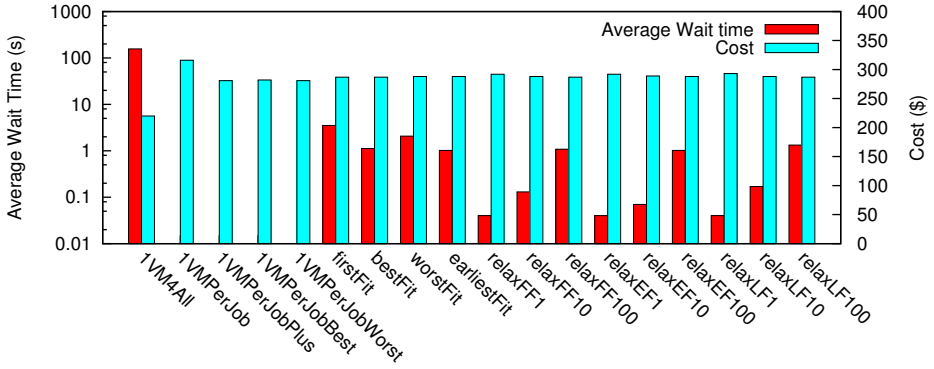
FIGURE 1.11 – Temps d'attente moyen (s) et cout ($), pour chaque stratégie, sur la trace de l'utilisateur LCG 155 (marge = 9%)

Utilisateurs avec une marge inférieure à 10% 24% des utilisateurs ont une marge de cout jusqu'à 10% ($0 < \text{marge} \leq 10\%$). C'est le cas de l'utilisateur 155 de la trace LCG que nous présentons dans la Figure 1.11. Dans ce cas, le résultat est proche de notre conclusion sur les traces complètes. Pour moins de 10% d'augmentation du cout, le temps d'attente peut être réduit à 0 s avec *1VMperJobPlus*. Les autres stratégies ne sont donc d'aucun intérêt puisqu'elles impliquent des temps d'attente largement supérieurs pour une augmentation du cout du même ordre de grandeur.

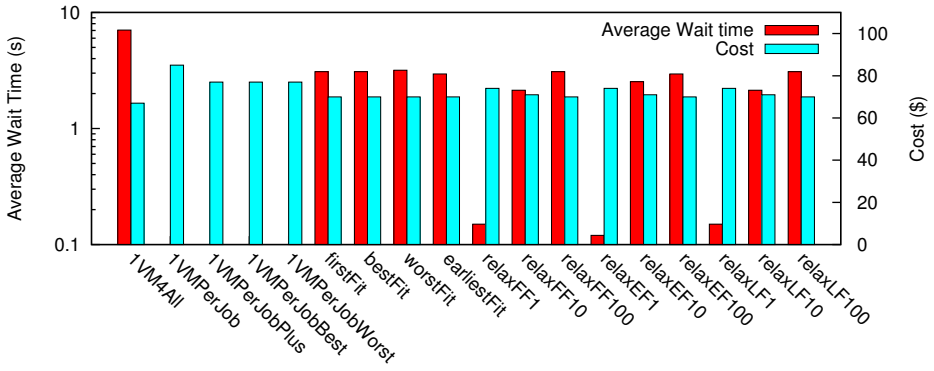
Utilisateurs avec une marge inférieure à 40% 21% des utilisateurs ont une marge de cout de 10% à 40% ($10\% < \text{marge} \leq 40\%$). C'est notamment le cas de l'utilisateur 212 de la trace LCG que nous présentons dans la Figure 1.12a. Dans ce cas, il y a de l'espace pour un compromis et les stratégies basées sur le bin packing peuvent être utilisées. Elles permettent en effet d'obtenir le cout de *1VM4All* tout en réduisant drastiquement le temps d'attente qu'implique cette stratégie. De petites différences existent entre les stratégies de bin packing : *BestFit* coute légèrement moins cher que *WorstFit*, mais augmente le temps d'attente et *FirstFit* se situe entre ces deux stratégies. Ce n'est pourtant pas le cas de tous les utilisateurs qui ont une telle marge. Par exemple, nous montrons sur la Figure 1.12b l'utilisateur 152 qui a une marge de 28%. Dans son cas, toutes les stratégies de bin packing ont le même cout, mais le temps d'attente est le moins élevé avec *BestFit* et le plus élevé avec *FirstFit*. C'est *WorstFit* qui se situe entre ces deux stratégies. Les stratégies les plus intéressantes sont cependant les stratégies de type *Relax*. Elles permettent en effet



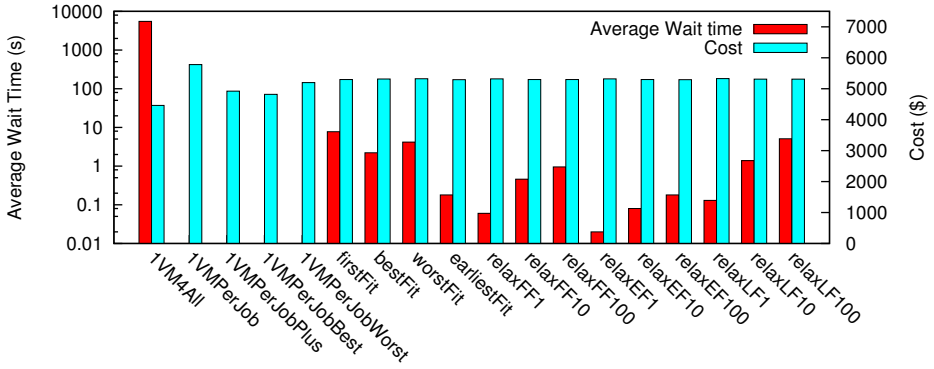
(a) Utilisateur LCG 212 (marge = 29%)



(b) Utilisateur LCG 152 (marge = 28%)



(c) Utilisateur LCG 203 (marge = 10%)



(d) Utilisateur LCG 127 (marge = 15%)

FIGURE 1.12 – Temps d'attente moyen (s) et cout ($), pour chaque stratégie, sur les traces des utilisateurs 212, 152, 203 et 127

de contrôler le compromis grâce au facteur de tolérance x. Nous n'avons cependant pas trouvé de caractéristique qui permette de donner une règle générale quant à la valeur de ce facteur. De la même façon, *RelaxFirstFit*x, *RelaxEarliestFit*x et *RelaxLastestFit*x ont des conséquences légèrement différentes, mais aucune caractéristique commune n'a été trouvée entre les utilisateurs. Nous montrons sur la Figure 1.12c l'utilisateur 203 et sur la Figure 1.12d l'utilisateur 127 qui ont respectivement une marge de cout de 10% et 15%. Pour le premier utilisateur, *RelaxEarliestFit*10 a un temps d'attente supérieur à celui de *RelaxFirstFit*10. En revanche pour le second utilisateur, c'est le temps d'attente de *RelaxFirstFit*10 qui est supérieur à celui de *RelaxEarliestFit*10.

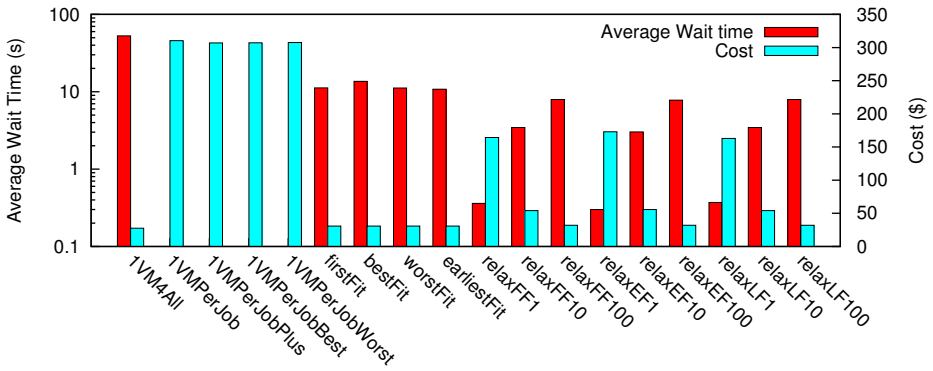


FIGURE 1.13 – Temps d'attente moyen (s) et cout ($), pour chaque stratégie, sur la trace de l'utilisateur LCG 10 (marge = 11,12)

Utilisateurs avec une marge supérieure à 40% Enfin, 24% des utilisateurs ont une marge de cout de plus de 40% (marge > 40%). C'est le cas de l'utilisateur 10 de la trace LCG que nous présentons dans la Figure 1.13. Les mêmes observations que celles faites dans le cas précédent s'appliquent également ici. Les stratégies de bin packing sont encore plus intéressantes et devrait être préférées à *1VMperJobPlus*.

1.6.4 Discussion

Les stratégies démarrent et stoppent dynamiquement des VM. Nous appelons le nombre de VM démarrées à un instant donné le *diamètre* du *provisioning*. Nous avons trouvé une relation directe entre ce diamètre et la marge de cout. Nous montrons sur la Figure 1.14 la relation entre la marge de cout et le diamètre maximum pour chaque utilisateur de LCG. Nous observons que la marge de cout est limitée lorsque le diamètre est élevé. En effet, aucune marge n'excède 50% lorsque le diamètre est supérieur à 170. Lorsque le diamètre augmente, le nombre d'opportunités pour réutiliser du temps d'inactivité des VM augmente également. Les stratégies de type *1VMperJob* sont dans ces cas très efficaces.

Cela explique pourquoi il n'y a que peu d'espace pour trouver un compromis avec des stratégies plus complexes lorsque nous regardons les traces des plateformes complètes sur

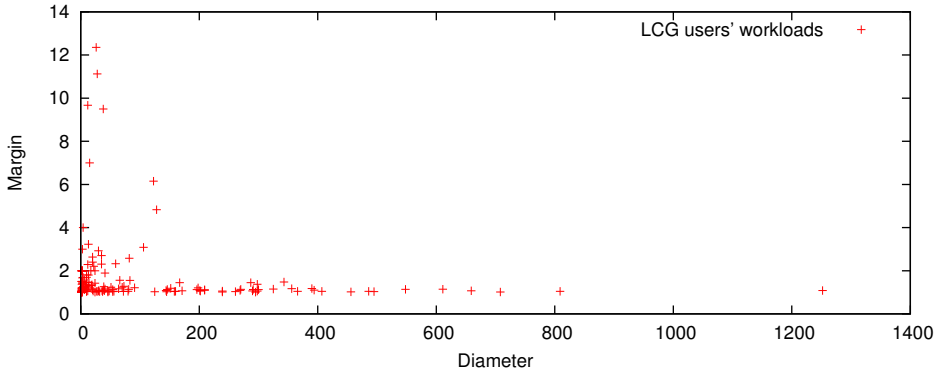


FIGURE 1.14 – Marge de cout et diamètre du *provisioning* sur les traces des utilisateurs

lesquelles le diamètre est naturellement élevé. Ainsi, lorsque le nombre de jobs concurrents est élevé, la meilleure stratégie de *provisioning* est *1VMperJobBest*.

Cependant, si ce nombre diminue, les stratégies de bin packing sont efficaces et permettent de trouver un compromis. Les différences entre les versions d'un même type de stratégie (c'est-à-dire *First*, *Best* ou *Early*) ne sont pas claires. À l'exception de cas très précis, la version *First* devrait être utilisée en raison de sa simplicité.

Les stratégies de type *Relax* sont cependant également intéressantes puisqu'elles permettent de contrôler plus précisément le compromis à l'aide du facteur de tolérance.

Nous recommandons donc à une grande organisation qui souhaite utiliser l'IaaS pour réaliser ses calculs de mutualiser sa plateforme virtuelle entre tous ses utilisateurs et d'utiliser une stratégie de *provisioning* de type *1VMperJob*.

1.7 Surcout de l'élasticité et de la puissance CPU

Dans le contexte du modèle économique d'Amazon EC2, nous mettons en lumière deux propriétés intéressantes : l'élasticité et la puissance CPU sans surcout. Dans cette section, nous étudierons dans quelle mesure il est possible de bénéficier de ces propriétés sans cout supplémentaire et pour quelles caractéristiques de workload.

1.7.1 Énoncé du problème

En prenant le modèle économique d'Amazon EC2, une VM qui s'exécute pendant deux heures a le même cout que deux VM qui s'exécutent pendant une heure. Le cout de déploiement est donc linéaire en fonction des BTU, indépendamment du nombre de VM démarrées. Nous appelons cette propriété l'*élasticité gratuite*.

Amazon exprime la puissance des CPU à louer en EC2 Compute Unit (EC2CU), 1 EC2CU étant équivalant à la puissance d'un cœur de Xeon cadencé à 1,7GHz. Dans le modèle économique d'Amazon EC2, une VM de 1 EC2CU qui s'exécute pendant 8 heures a le même cout qu'une VM de 8 EC2CU qui s'exécute pendant 1 heure. Par exemple, chez

Amazon, une instance de type *small* coute 0,08$/h pour une puissance de 1 EC2CU. Une instance *extra-large* coute 0,64$/h pour un total de 8 EC2CU[1]. Ces deux types d'instances partagent le même cout par heure par EC2CU. Le cout de déploiement est donc linéaire en fonction du nombre de cycles CPU, indépendamment de la vitesse du CPU loué. Nous appelons cette propriété la *puissance CPU gratuite*.

Dans cette section, nous allons nous concentrer sur deux types d'instances à la demande : les instances *small* avec 1 EC2CU et les instances *medium* avec 2 EC2CU. Nous avons choisi ces types d'instances, car ils fournissent des VM simple cœur et permettent ainsi une comparaison directe avec la puissance de leurs CPU. Nous avons normalisé la durée des jobs selon la puissance des instances *small*. Ainsi, durant une BTU, une instance *small* peut exécuter 3 600 s de calcul alors qu'une instance *medium* peut en exécuter 7 200 s. En d'autres termes, un job de 3 600 s s'exécute durant la totalité de la BTU d'une instance *small* mais seulement durant la moitié d'une BTU sur une instance *medium*. Leur prix est respectivement de 0,08$/h et 0,16$/h. Ainsi, pour 0,16$/h, un utilisateur peut obtenir :

a) deux BTU d'une instance *small* ;

b) une BTU de deux instances *small* ;

c) une BTU d'une instance *medium*.

Le choix le plus intéressant semble être b) ou c), car les calculs finiront plus tôt et particulièrement c), car les calculs vont plus vite. Cependant, afin de bénéficier de cette propriété sans surcout, il faut être capable de garder la BTU occupée jusqu'à sa fin. Par exemple, exécuter un job de 1 000 s coutera 0,08$ sur une instance *small* et 0,16$ sur une instance *medium*.

Notre objectif dans cette section est donc de vérifier si cette condition peut être satisfaite et ce que cela implique en terme de caractéristiques des workloads.

1.7.2 Évaluation

L'évaluation est basée sur l'analyse de cinq workloads provenant de différentes grilles en production et sur trois stratégies de *provisioning* et d'ordonnancement. Dans cette section, nous présentons dans un premier temps les stratégies et les workloads. Puis nous introduisons les objectifs et métriques de l'évaluation avant de passer à l'évaluation à proprement parler.

1. Les prix utilisés correspondent à ceux annoncés par Amazon en mai 2012, mais le rapport entre ces prix reste vrai en mars 2015

TABLE 1.6 – Caractéristiques des workloads

Trace	#jobs	#CPUs	#utilisateurs	Inter-arrivée	Walltime
LCG	188 041	24 115	216	5	8 970
AuverGrid	347 611	475	405	78	25 186
NorduGrid	781 370	2 000	387	127	89 273
SharcNet	1 195 242	6 828	412	28	31 964
Unistra	306 605	1 000	74	117 006	51 906

Les stratégies de *provisioning*

Nous avons décrit dans la Section 1.5.2 une dizaine de stratégies de *provisioning*. Dans le cadre de ce travail, nous nous concentrerons sur les trois stratégies de *provisioning* les plus intéressantes : *1VM4All*, *1VMperJobPlus* et *FirstFit*. Si *1VMperJobBest* semble être une meilleure stratégie que *1VMperJobPlus* dans la majorité des cas, l'amélioration est de faible importance et nous avons vu qu'il est impossible de détecter les cas où l'une sera plus intéressante que l'autre. De plus, *1VMperJobBest* nécessite de connaître la durée des jobs ce qui n'est pas le cas de *1VMperJobPlus*. C'est pourquoi nous avons choisi de garder *1VMperJobPlus* dans la suite de nos travaux.

Les workloads

Afin d'évaluer la possibilité de bénéficier de ces deux propriétés de l'IaaS sans surcout, nous reprenons les workloads présentés dans la Section 1.6.1. Nous y ajoutons un workload provenant du mésocentre de l'Université de Strasbourg que nous décrivons dans la Table 1.6. Les temps sont des valeurs moyennes en seconde.

Ce workload contient les traces d'exécution de mars 2009 à mars 2011 provenant de différents laboratoires de l'université. Il se compose de jobs réservant un ou plusieurs processeurs, certains parallèles (p. ex. MPI). Afin de respecter nos hypothèses, nous considérons qu'un seul job s'exécute par processeur réservé.

Objectifs de l'évaluation

L'objectif de l'évaluation est de déterminer si réduire le *makespan* en utilisant des types d'instances plus puissants (CPU) ou plus d'instances simultanément (élasticité) est possible sans surcout. Dans cet objectif, nous utilisons le même simulateur ad hoc que présenté dans la Section 1.6 dans lequel nous avons implémenté les stratégies de *provisioning* et d'ordonnancement. Nous avons dans un premier temps extrait les soumissions de chaque utilisateur individuel (1 306 utilisateurs). Puis nous avons simulé l'exécution des stratégies de *provisioning* sur chacun de ces workloads. Nous avons ensuite calculé deux métriques correspondant aux deux objectifs que les stratégies de *provisioning* cherchent à satisfaire :

Total cost : cout de la location des ressources provisionnées en $ définit dans l'Équation 1.2.

Slowdown : représente la satisfaction de l'utilisateur. Cette métrique part du principe qu'un utilisateur est prêt à attendre plus longtemps pour l'exécution d'un job long que pour l'exécution d'un job court. Il s'agit du ratio moyen entre le temps d'attente et le temps d'exécution de chaque job. Il est défini par Iosup et al. [26] de la façon suivante : $sd_j = \frac{r_j+w_j}{r_j}$. Par exemple, en prenant un job j, $sd_j = 1$ signifie que le temps d'attente du job est nul ($w_j = 0$). Et $sd_j = 3$ signifie que le temps d'attente est deux fois plus long que le temps d'exécution ($w_j = 2 \times r_j$).

Dans la suite de cette section, nous notons S_{1x} et S_{2x} l'exécution de la stratégie de *provisioning* S en utilisant respectivement des instances *small* ($1x$) et *medium* ($2x$).

Regroupement des utilisateurs

Dans la suite de l'évaluation, nous nous intéressons aux solutions *quasi* optimales en cout que nous définissons comme étant les solutions ne dépassant pas l'optimal de plus d'un facteur ε. Ainsi, nous considérons ces solutions comme étant ε-gratuite.

Notre but ici est de présenter un découpage des 1 306 utilisateurs en groupe en fonction du cout supplémentaire induit par l'utilisation de l'élasticité ou de VM plus puissantes. Nous utilisons $\mathit{1VM4All}_{1x}$ comme stratégie de référence. En l'utilisant avec des instances de type *small*, elle nous fournit donc une borne minimale pour le cout.

Notons r le cout de référence : $r = (1 + \varepsilon) \times \text{cost}(\mathit{1VM4All}_{1x})$. Ce cout est ensuite utilisé pour séparer les utilisateurs en catégories exclusives d'après les prédicats suivants :

Les trois premiers prédicats concernent la propriété de bénéficier de façon ε-gratuite de l'élasticité (e) des clouds IaaS.

$$e = \text{cost}(\mathit{1VMperJobPlus}_{1x}) \leq r \tag{1.4}$$

Nous notons e le prédicat correspondant aux cas où l'utilisation de la stratégie *1VMperJobPlus* sur des instances *small* est moins couteuse que le cout de référence r.

$$e' = \neg e \wedge (\text{cost}(\mathit{FirstFit}_{1x}) \leq r) \tag{1.5}$$

Nous notons e' le prédicat correspondant aux cas où l'utilisation de la stratégie *FirstFit* sur des instances *small* est moins couteuse que le cout de référence r en excluant les cas rentrant dans le prédicat e.

$$\bar{e} = \neg(e \vee e') \tag{1.6}$$

Nous notons $\bar{e}$ le prédicat ne correspondant à aucun des deux prédicats précédents : e et e'.

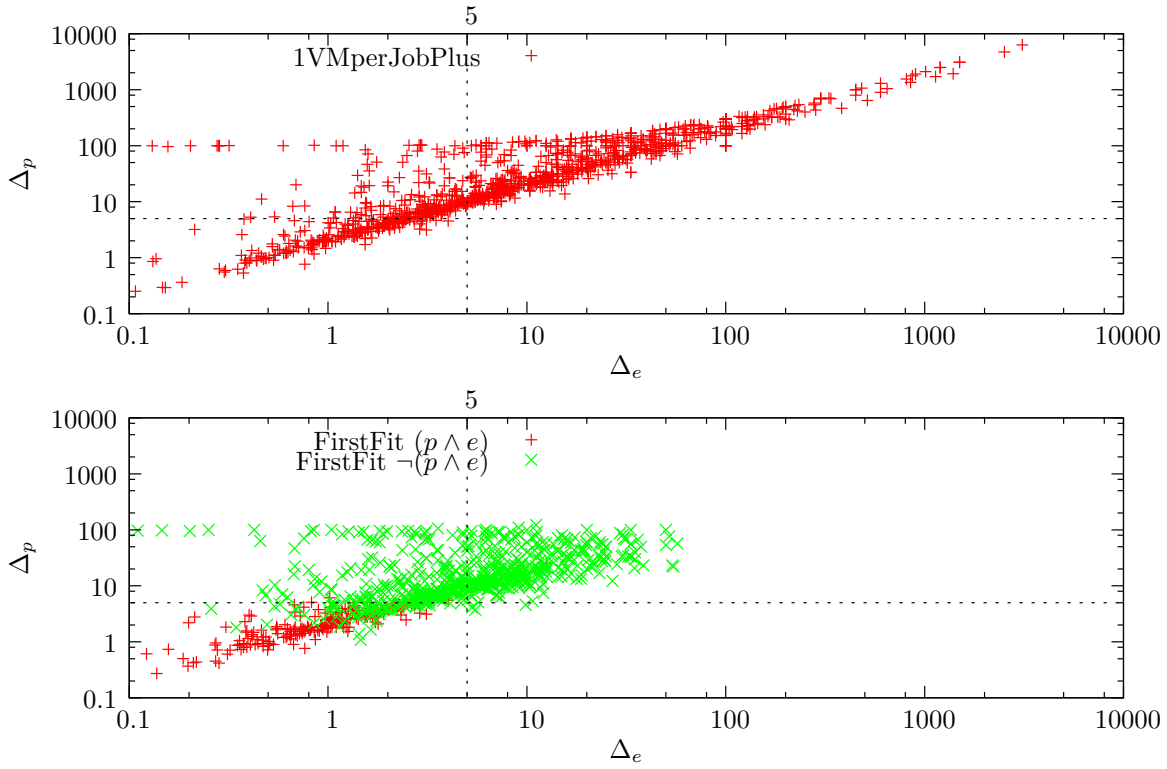


FIGURE 1.15 – Distribution des utilisateurs en fonction de la différence entre la stratégie de référence et *1VMperJobPlus* (haut) et *FirstFit* (bas)

Les trois derniers prédicats concernent la propriété de bénéficier de façon ε-gratuite de la puissance CPU (p) des clouds IaaS.

$$p = \text{cost}(\mathit{1VMperJobPlus}_{2x}) \leq r \tag{1.7}$$

Nous notons p le prédicat correspondant aux cas où l'utilisation de la stratégie *1VMperJobPlus* sur des instances *medium* est moins couteuse que le cout de référence r.

$$p' = \neg p \wedge (\text{cost}(\mathit{FirstFit}_{2x}) \leq r) \tag{1.8}$$

Nous notons p' le prédicat correspondant aux cas où l'utilisation de la stratégie *FirstFit* sur des instances *medium* est moins couteuse que le cout de référence r en excluant les cas rentrant dans le prédicat p.

$$\bar{p} = \neg(p \vee p') \tag{1.9}$$

Enfin, nous notons $\bar{p}$ le prédicat ne correspondant à aucun des deux prédicats précédents : p et p'.

Sur la Figure 1.15, nous montrons, pour chaque utilisateur, un point aux coordonnées (Δ_e, Δ_p). Δ_e représente l'augmentation du cout par rapport à la référence qui résulte de l'utilisation de la stratégie S sur des instances *small*. Δ_p représente l'augmentation

du cout par rapport à la référence qui résulte de l'utilisation de la stratégie S sur des instances *medium*. Ces augmentations de cout sont représentées comme des pourcentages par rapport à la stratégie de référence : $1VM4All_{1x}$. Sur la Figure 1.15, ces résultats sont montrés pour $S = 1VMperJobPlus$ (gauche) et $S = FirstFit$ (droite). Plus formellement, ces augmentations de cout s'expriment de la façon suivante :

$$\Delta_e = \frac{\text{cost}(S_{1x}) - r}{r} \times 100\,;$$

$$\Delta_p = \frac{\text{cost}(S_{2x}) - r}{r} \times 100\,;$$

avec $S \in \{1VMperJobPlus, FirstFit\}$.

Les lignes pointillées représentent le seuil ε que nous avons défini pour cette analyse.

Prenons par exemple un point en $(10, 100)$. Cela signifie qu'un utilisateur aurait une augmentation de *10%* du cout s'il augmente l'élasticité, c'est-à-dire s'il utilise *1VMperJobPlus* ou *FirstFit* avec des instances *small*. Il aurait de plus une augmentation de *100%* s'il utilise des VM deux fois plus puissantes (instances *medium*).

Nous observons dans un premier temps que la grande majorité des utilisateurs restent dans un budget moins de deux fois supérieur à la référence ($\Delta_x \leq 100\%$). Cependant, *1VMperJobPlus* entraîne, pour quelques utilisateurs, des couts de l'ordre de dix fois l'optimal (max. = 3 100%). En revanche, *FirstFit* est moins couteuse puisque le cout est systématiquement inférieur à deux fois la référence.

Deuxièmement, bien que non visible sur ces figures, nous constatons que *FirstFit* réduit le cout pour la plupart des utilisateurs (93,49%) par rapport à la même exécution avec *1VMperJobPlus*.

Enfin, nous voyons que pour tous les utilisateurs, le cout supplémentaire dû à l'utilisation d'un CPU plus puissant (Δ_p) est systématiquement supérieur à l'augmentation du cout dû à l'élasticité (Δ_e).

Dans la suite de cette section, comme aucun utilisateur n'a une augmentation de 0% du cout, nous considérons arbitrairement qu'une augmentation de 5% est raisonnable. Nous nous concentrons donc sur les cas où $\varepsilon = 0,05$.

Nous présentons dans la Table 1.7 le découpage des utilisateurs en groupe en fonction des prédicats pour $\varepsilon = 0,05$. Les colonnes représentent les catégories d'utilisateurs qui obtiennent de l'élasticité ε-gratuite : pour les utilisateurs de la colonne e, *1VMperJobPlus* est suffisant ; pour les utilisateurs de la colonne e', *FirstFit* doit être utilisé ; pour les utilisateurs de la colonne $\bar{e}$, aucune de nos stratégies ne permet de profiter de l'élasticité ε-gratuite. Les lignes représentent les catégories d'utilisateurs qui obtiennent de la puissance CPU ε-gratuite : p si *1VMperJobPlus* est suffisant, p' s'il est nécessaire d'utiliser *FirstFit*, et $\bar{p}$ si aucune stratégie n'est satisfaisante.

Chacune des cellules représente la part des utilisateurs appartenant à la catégorie validant les prédicats correspondant à la ligne et à la colonne. Les valeurs entre parenthèses

TABLE 1.7 – Regroupement des utilisateurs pour $\varepsilon = 0.05$. Part des utilisateurs (et valeurs absolues) des catégories.

CPU \ Elasticity	e	e'	$\bar{e}$	Σ
p	**17,08%** (223)	**0,08%** (1)	**0%** (0)	**17,15%** (224)
p'	**4,29%** (56)	**1,68%** (22)	**0,23%** (3)	**6,20%** (81)
$\bar{p}$	**31,16%** (407)	**23,28%** (304)	**22,21%** (290)	**76,65%** (1001)
Σ	**52,53%** (686)	**25,04%** (327)	**22,43%** (293)	**100%** (1306)

sont les valeurs absolues. Par exemple, la cellule à gauche de la deuxième ligne indique que 4,29% des utilisateurs (soit 56 utilisateurs) peuvent bénéficier de l'élasticité ε-gratuite en utilisant la stratégie *1VMperJobPlus*$_{1x}$ ou de la puissance CPU ε-gratuite en utilisant la stratégie *FirstFit*$_{2x}$. Dans les deux cas nous obtenons un cout inférieur à ε fois le cout de la stratégie de référence. Bien que n'apparaissant pas dans ce tableau, il est important de noter que ce découpage est indépendant de la provenance de la trace.

En additionnant les valeurs de la première colonne, nous remarquons que 52,53% des utilisateurs ont un intérêt à utiliser *1VMperJobPlus*$_{1x}$, et ainsi obtenir de l'élasticité ε-gratuite. En ajoutant les valeurs des cas pour lesquels l'élasticité ε-gratuite s'obtient en utilisant *FirstFit*$_{1x}$ (deuxième colonne), cette valeur grimpe à 77,57% des utilisateurs qui peuvent bénéficier de l'élasticité ε-gratuite.

En lisant la table en suivant les lignes, nous constatons qu'un petit nombre d'utilisateurs peut utiliser des instances *medium* pour le cout des *small*. Seuls 17,15% des utilisateurs peuvent en bénéficier sans ajouter de temps d'attente, et 23,35% en bénéficient si nous ajoutons *FirstFit*. À partir de cette table, on en déduit que *FirstFit* est intéressant dans deux cas : elle permet à 23,28% des utilisateurs d'obtenir un cout quasi-optimal avec des instances *small*, et à 6,2% $(4,29\% + 1,68\% + 0,23\%)$ des utilisateurs d'utiliser des instances *medium* sans cout supplémentaire.

Analyse en fonction des caractéristiques

Nous essayons maintenant de répondre à la question : « Existe-t-il une caractérisation des workloads qui nous permette de prédire l'effet des stratégies ? » Dans la Section 1.6, nous avons évalué nos stratégies avec l'objectif de mettre en avant celles qui nous permettent d'obtenir un bon compromis entre le cout et la performance. Nous avons trouvé une relation directe entre le diamètre, le temps d'exécution et le cout. Nous souhaitons donc examiner si le temps d'exécution et le diamètre peuvent également être déterminants dans ces catégories. Pour cette analyse, nous laissons de côté les catégories ne contenant que peu d'utilisateurs ($p \wedge e'$, $p \wedge \bar{e}$ et $p' \wedge \bar{e}$).

Nous montrons dans les figures 1.16 et 1.17 les temps d'exécution moyens et le diamètre pour les six catégories d'utilisateurs restantes. La catégorie des utilisateurs qui ne peuvent

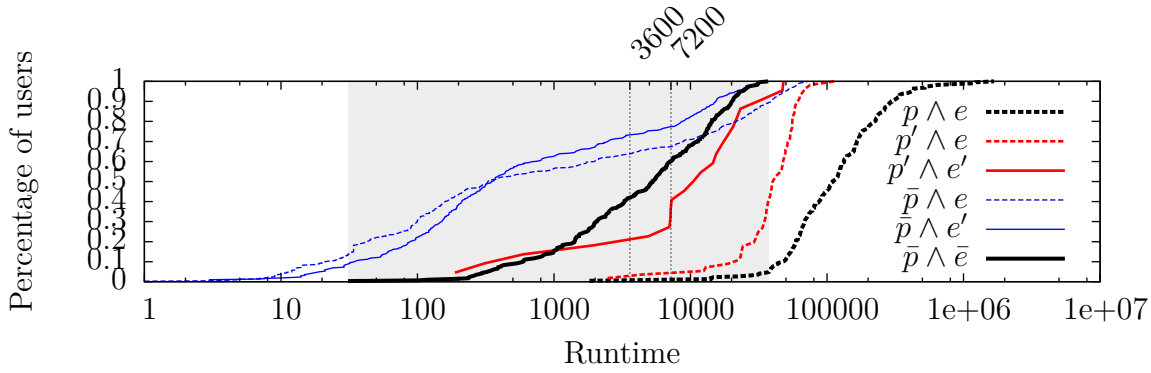


FIGURE 1.16 – Distribution cumulative des utilisateurs en fonction du temps d'exécution moyen pour les catégories d'utilisateurs

bénéficier ni de l'élasticité ε-gratuite, ni de la puissance CPU ε-gratuite ($\bar{p} \wedge \bar{e}$), a un fond grisé qui délimite les valeurs observées pour cette catégorie. Sur la Figure 1.16, chaque courbe est une fonction de répartition cumulative des temps d'exécution moyens de chaque utilisateur. Par exemple, 70% des utilisateurs appartenant à la catégorie $\bar{p} \wedge e'$ — qui ne bénéficient pas de la puissance CPU ε-gratuite mais bénéficient de l'élasticité ε-gratuite avec *FirstFit* — ont des jobs avec un temps d'exécution moyen plus petit ou égal à 3 600 s.

La première observation, sur la Figure 1.16 est que les utilisateurs appartenant à la catégorie $p \wedge e$ — des utilisateurs qui bénéficient à la fois de l'élasticité ε-gratuite, et de la puissance CPU ε-gratuite — se caractérisent par des temps d'exécution très longs (160 610 s en moyenne) qui permettent de garder les instances *medium* occupées pendant toute la durée des BTU.

Deuxièmement, les utilisateurs appartenant à la catégorie $p' \wedge e$ — des utilisateurs qui bénéficient de la puissance CPU ε-gratuite en utilisant $FirstFit_{2x}$ et de l'élasticité ε-gratuite avec $1VMperJobPlus_{1x}$ — ont des temps d'exécution légèrement plus courts (42 813 s en moyenne). De plus, 40% des utilisateurs ont un temps d'exécution moyen dans la zone grisée. Cela implique que les VM ne peuvent pas être gardées occupées en utilisant uniquement $1VMperJobPlus_{2x}$, car les jobs sont trop courts par rapport à la taille de la BTU des instances *medium* (7 200 s), ce qui entraîne du temps d'inactivité dans la BTU. Cependant, *FirstFit* parvient à réutiliser ce temps dans certains cas.

Troisièmement, nous constatons que le temps d'exécution n'est pas déterminant pour les quatre autres catégories. Quel que soit son temps d'exécution moyen, l'utilisateur peut être dans la catégorie $\bar{p} \wedge \bar{e}$, $p' \wedge e'$, $\bar{p} \wedge e$ ou $\bar{p} \wedge e'$.

De la même manière, la Figure 1.17 montre la fonction de répartition cumulative des diamètres moyens de chaque utilisateur, c'est-à-dire le nombre de VM s'exécutant simultanément. Par exemple, 90% des utilisateurs appartenant à la catégorie $\bar{p} \wedge e'$ — qui ne bénéficient pas de la puissance CPU ε-gratuite mais bénéficient de l'élasticité ε-gratuite avec *FirstFit* — ont un diamètre inférieur ou égal à 10.

Nous pensions qu'un diamètre élevé augmenterait les chances de réutiliser le temps

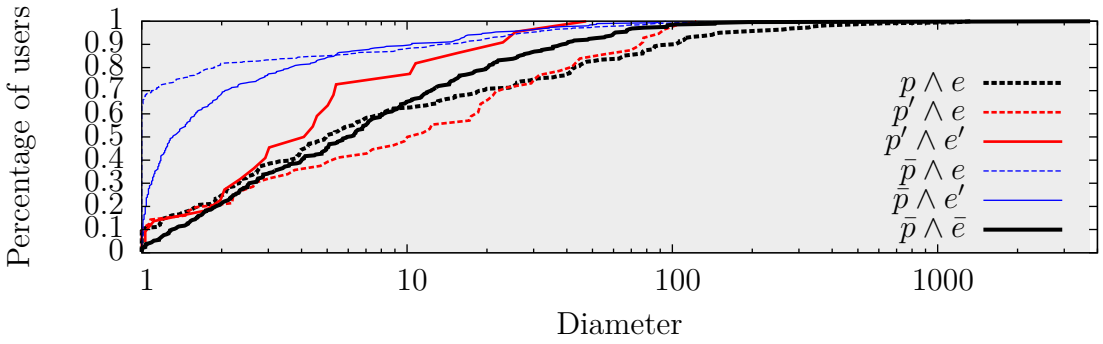


FIGURE 1.17 – Distribution cumulative des utilisateurs en fonction du diamètre pour les catégories d'utilisateurs

d'inactivité de la VM. Mais cette figure indique que, quel que soit le diamètre, nous ne sommes pas capables de prédire les résultats de l'emploi d'une stratégie et d'un type d'instance. Par exemple, l'utilisateur SharcNet-U111 a un diamètre très élevé de 3 753, mais ne peut bénéficier ni de l'élasticité ni de la puissance CPU ε-gratuite. La seule exception est le diamètre de 1. Dans ce cas, toutes les stratégies fournissent le même *provisioning*. Nous recommandons donc d'utiliser la stratégie $1VMperJobPlus_{1x}$ qui est la plus simple à mettre en œuvre.

L'observation la plus intéressante est qu'il est impossible de prédire dans quelle catégorie d'utilisateur un workload sera, sauf pour $1VMperJobPlus_{2x}$ qui concerne les temps d'exécution très longs. De plus, la plupart des catégories ont des workloads partageant les mêmes caractéristiques. C'est notamment le cas de la catégorie $\bar{p} \wedge \bar{e}$ — des utilisateurs qui ne bénéficient ni de la puissance CPU ε-gratuite ni de l'élasticité ε-gratuite — qui ne peut être séparée des autres comme le montre la zone grisée sur les figures 1.16 et 1.17.

Nous avons mené une étude complète sur un grand nombre de workloads d'utilisateurs afin d'analyser le comportement de stratégies de *provisioning* au regard des caractéristiques des soumissions. Nous avons étudié un grand nombre de métriques tel que l'hétérogénéité des temps d'exécution et des diamètres. Cependant, nous n'avons pas réussi à trouver une corrélation entre ces métriques et les performances des stratégies de *provisioning*. Il nous semble donc difficile, voire impossible, de prédire les performances des stratégies de *provisioning* en se basant uniquement sur des métriques agrégées. Ceci s'explique par les effets de seuil dus à la grande taille de la BTU imposent une gestion et une analyse fine des soumissions. Pour illustrer, nous présentons dans la Figure 1.18 un cas particulier. Considérons deux jobs de 3 601 s et 3 599 s. Le second correspond parfaitement à la BTU d'une instance *small* alors que le premier double le cout. De plus, s'ils sont soumis simultanément, *FirstFit* se comportera efficacement, mais pas *1VMperJobPlus* ; s'ils sont soumis avec quelques secondes d'écart, *1VMperJobPlus* se comportera efficacement, mais pas *FirstFit* ; mais aucune des deux stratégies ne fonctionne avec un écart de 1 s entre les soumissions.

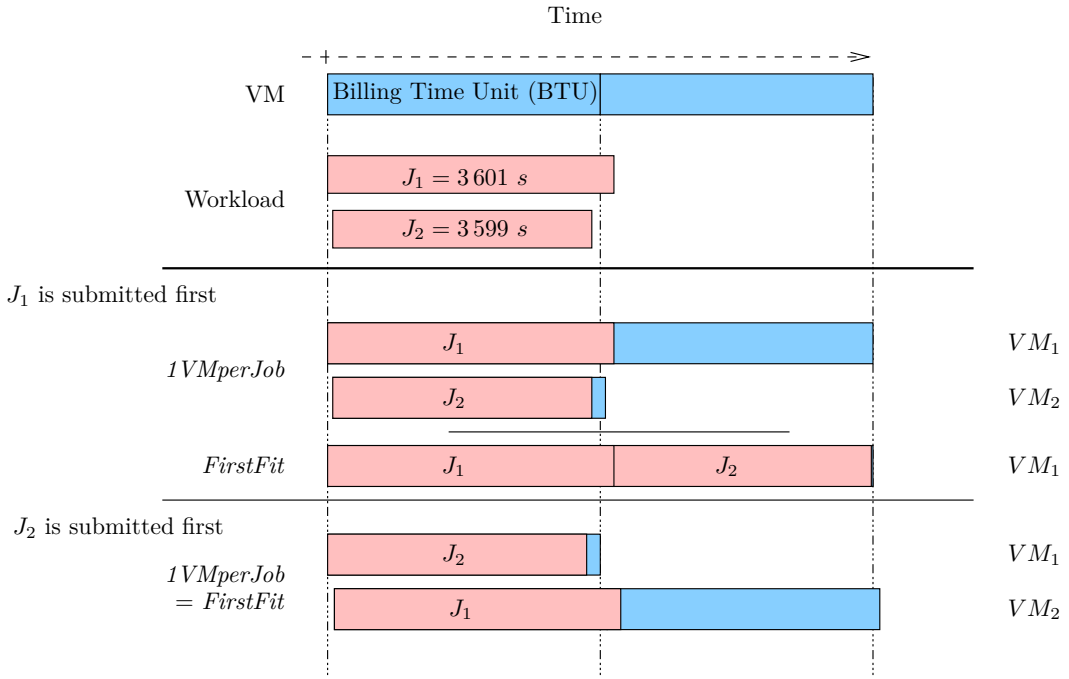


FIGURE 1.18 – Effet de seuil rendant impossible la prédiction du comportement des stratégies

1.7.3 Conclusion sur l'élasticité et la puissance CPU

Alors qu'obtenir de l'élasticité gratuite est aisé, bénéficier de la puissance CPU gratuite est plus compliqué. Seuls des workloads avec des temps d'exécution très longs, de l'ordre de 100 000 s, peuvent obtenir les deux en utilisant $1VMperJobPlus_{2x}$. Les utilisateurs avec un temps d'exécution légèrement moins élevé, de l'ordre de 40 000 s, peuvent en bénéficier en utilisant $FirstFit_{2x}$.

En utilisant des instances *small*, l'élasticité gratuite est possible pour la plupart des utilisateurs en utilisant $1VMperJobPlus_{1x}$ et $FirstFit_{1x}$ fonctionne pour la plupart d'entre eux.

Cependant, prédire les conséquences du choix d'une stratégie de *provisioning* et d'un type d'instance est impossible en n'utilisant que des caractéristiques globales des workloads. Notre recommandation est donc d'utiliser $1VMperJobPlus_{2x}$ si des caractéristiques très particulières du workload sont réunies telles que des temps d'exécution très longs. Dans le cas contraire, nous avons besoin d'utiliser un outil d'aide à la décision. Il peut, par exemple, s'agir d'un simulateur de système distribué permettant d'obtenir avant l'exécution réelle, des informations sur les exécutions avec les différentes stratégies. La réalisation d'un tel simulateur est importante pour l'analyse et la compréhension du comportement d'applications sur des systèmes distribués. Ce problème est cependant difficile à résoudre, car la simulation d'un cloud nécessite la prise en compte de nombreux facteurs tels que la virtualisation, les entrées-sorties au niveau du disque ou du réseau ainsi que les surcouts dus à Schlouder, au cloud kit et au *batch scheduler*.

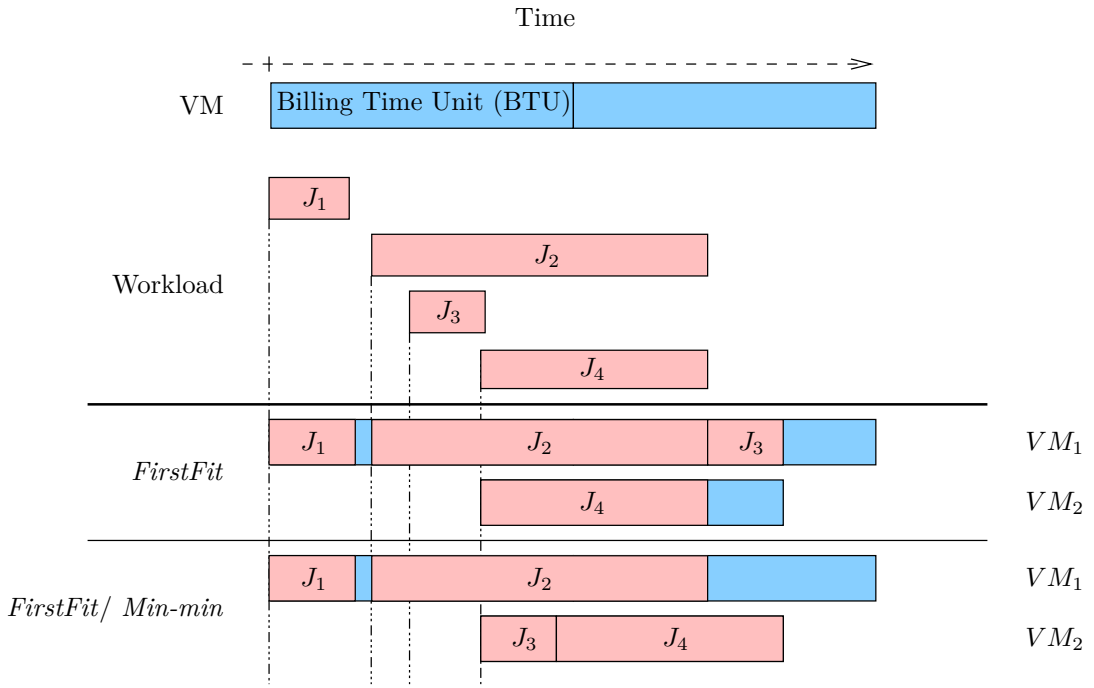


FIGURE 1.19 – Exemple d'application de l'algorithme d'ordonnancement *Min-min* en complément de *FirstFit*

1.8 Ordonnancement semi-online

Nous avons vu que la stratégie *FirstFit* permet d'obtenir une augmentation du cout raisonnable, cette stratégie implique souvent une augmentation du *slowdown* importante. Cet effet peut être minimisé en utilisant, en plus de *FirstFit*, l'algorithme d'ordonnancement *Min-min*. En effet, nous avons vu que *FirstFit* prend une décision de *provisioning online*. En d'autres termes, *FirstFit* décide sur quelle VM le job sera exécuté dès sa soumission. Ce n'est pas nécessairement la meilleure décision et c'est pourquoi nous observons un temps d'attente considérable sur certains workloads. Une meilleure décision peut être prise en relâchant cette contrainte online. Lorsque les jobs patientent dans la file d'attente d'une VM, nous pouvons recalculer leur ordonnancement, à chaque fois qu'un nouvel évènement arrive. Nous appelons ça de l'ordonnancement *semi- online*. Cela se traduit par la réduction du *slowdown* sans impact sur le cout.

Les heuristiques présentées par Maheswaran et al. [38] peuvent être utilisées en complément de *FirstFit*. Parmi ces heuristiques, seule *Min-min* a été testé. L'objectif étant principalement de découvrir si l'utilisation d'un ordonnancement semi-online est utile. L'objectif n'est pas de trouver le meilleur algorithme. L'algorithme *Min-min* favorise l'exécution des petits jobs en les ordonnançant sur les VM où ils finiront leur exécution le plus tôt. Nous montrons un exemple d'application de *Min-min* sur la Figure 1.19. En appliquant *FirstFit*, le job J_3 est ajouté à la file d'attente de la VM_1. Lorsque J_4 est soumis, une seconde VM est démarrée. À ce moment, l'exécution de *Min-min* réordonnance tous les jobs en attente (J_3 et J_4). J_3 est plus petit que J_4 et finira son exécution plus tôt sur

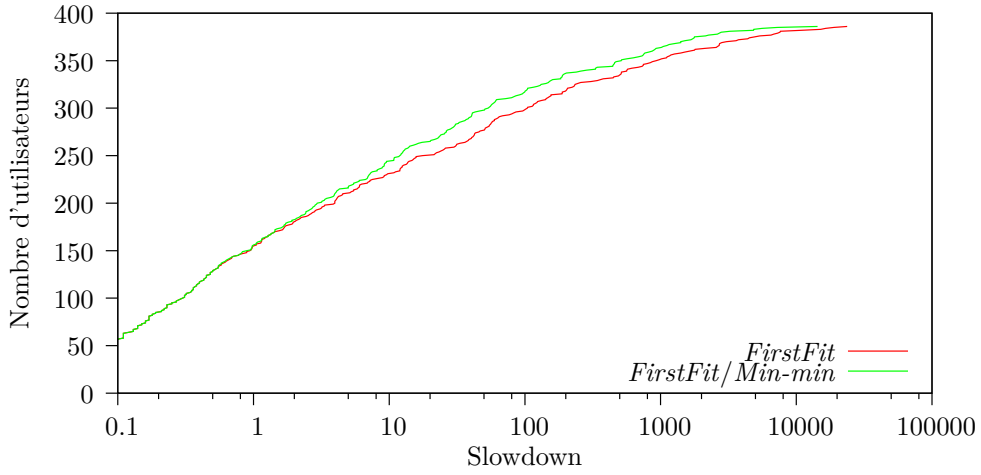


FIGURE 1.20 – Comment l'utilisation de *Min-min* impacte le *slowdown* ?

la VM_2. Il y est donc ordonnancé afin de réduire le *slowdown*.

Cette stratégie est plus complexe d'un point de vue algorithmique. Sa mise en œuvre peut donc être difficile dans un environnement online où un grand nombre de jobs est soumis.

Nous avons effectivement constaté que *Min-min* permet de réduire le *slowdown*. La Figure 1.20 représente la fonction de répartition cumulative du *slowdown* pour les utilisateurs en utilisant *FirstFit* et *FirstFit* avec la stratégie d'ordonnancement *Min-min*. Parmi les 386 utilisateurs concernés (ceux dans les cases grisées de la Table 1.7), 252 ne voient aucune amélioration en utilisant *Min-min*, alors que pour les 134 utilisateurs restants (34%) le *slowdown* est réduit de 32% en moyenne. Mais seuls 21 utilisateurs (5,4%) ont un *slowdown* qui passe en dessous de 3. Par conséquent, afin de rester dans une borne raisonnable de 5% d'augmentation du cout, les utilisateurs de ces catégories doivent accepter des temps d'attente considérables. Il est donc possible de raffiner les résultats obtenus avec les stratégies de bin packing. Ce n'est cependant pas l'enjeu de notre recherche et nous ne poursuivrons donc pas plus loin les recherches dans ce domaine. Nous nous concentrons plutôt dans cette thèse sur le problème de comprendre comment le choix d'une stratégie de *provisioning* peut se faire.

1.9 Conclusions sur les stratégies de *provisioning* et perspectives

Nous avons dans un premier temps souligné l'importance du calcul scientifique et des systèmes distribués pour l'amélioration des performances de ces calculs. L'avènement du cloud IaaS entraîne une nouvelle façon de voir l'infrastructure de calcul dans un système distribué. Les problèmes liés au *provisioning* des ressources dans un contexte IaaS sont très importants car :

- l'élasticité intrinsèque de l'IaaS rend possible la modification de la taille de l'infrastructure à tout moment. La taille de l'infrastructure conditionne les performances de l'application à exécuter ainsi que le prix à payer.
- malgré le succès de la grille au sein de la communauté scientifique, sa maintenance et son utilisation par le middleware sont difficiles et pas toujours fiables. L'IaaS et particulièrement la virtualisation apporte une isolation permettant de simplifier le middleware et est prometteur pour l'exécution de calculs scientifiques.

En outre, les problèmes liés au *provisioning* des ressources dans un contexte IaaS sont très difficiles car :

- la réalisation manuelle du *provisioning* risque d'aboutir à une décision naïve et sous-optimale à cause du grand nombre de paramètres à prendre en compte.
- des métriques contradictoires sont à prendre en compte dans la décision de *provisioning* tel que raccourcir le temps d'exécution et réduire les couts de l'infrastructure.
- l'ordonnancement dans ce contexte est *online* : les décisions de *provisioning* et d'ordonnancement sont prises au moment de la soumission des tâches.

Nous avons présenté une quinzaine de stratégies de *provisioning*. En nous basant sur le modèle économique à la demande d'Amazon EC2, nous proposons trois familles de stratégies : une borne minimale du cout qui implique un grand temps d'attente, une borne minimale en temps d'attente mais plus couteuse et des stratégies basées sur les heuristiques qui proposent des solutions au problème de bin packing.

Notre contribution présentée ici est double. Tout d'abord, nous avons montré qu'il existe un lien entre le diamètre du *provisioning* et la marge de cout qui permet de trouver un compromis entre le cout et la performance (réduction du temps d'exécution d'un workload). En effet, lorsque le diamètre augmente, le nombre d'opportunités pour réutiliser du temps d'inactivité des VM augmente également. Nos stratégies permettent donc quand c'est possible de réduire le cout tout en ayant des performances optimales. Dans les autres cas, nos stratégies offrent la possibilité de trouver un compromis entre le cout et la performance.

Deuxièmement, nous avons mis en avant deux propriétés intéressantes des clouds IaaS utilisant le même modèle économique qu'Amazon EC2. Ces propriétés permettent l'utilisation de l'élasticité ou de VM plus puissantes sans surcout dans la majorité des cas. Cependant, nous mettons en lumière qu'il est impossible prédire les conséquences du choix d'une stratégie de *provisioning* et d'un type d'instance est impossible en n'utilisant que des caractéristiques globales des workloads.

Cette étude, qui se base sur un simulateur, est une première brique dans l'utilisation des clouds IaaS pour l'exécution de calculs intensifs. Nous montrons la possibilité de trouver

un compromis entre le cout et la performance en utilisant des stratégies de *provisioning* variées. À partir de ces constats, nous formulons deux objectifs principaux qui constituent notre fil rouge dans la résolution du problème de *provisioning* automatique des ressources dans un environnement de type IaaS.

Premièrement, nous proposons l'installation d'un système de courtage entre le client et le fournisseur d'IaaS. Ce système aura en charge la réalisation du *provisioning* pour le client. Au regard de la diversité des possibilités de compromis entre le cout et la performance, ce système doit posséder une architecture ouverte permettant la création et l'intégration aisée de nouvelles stratégies de *provisioning*.

Deuxièmement, nous avons mis en lumière la nécessité d'utiliser un outil afin d'aider les utilisateurs à choisir la bonne stratégie de *provisioning*. Nous ouvrons donc ici un nouveau problème, difficile à résoudre, car un tel outil nécessite la prise en compte de nombreux facteurs tels que la virtualisation, les entrées-sorties au niveau du disque ou du réseau ainsi que les surcouts dus à Schlouder, au cloud kit et au *batch scheduler*. Cet outil doit également s'adapter aux différentes plateformes et aux différents workloads qu'un utilisateur exploite. Cet outil doit fournir un retour clair en terme de couts et de performances.

Ces deux outils forment les briques de base de notre approche et sont présentés dans les prochains chapitres. Dans le Chapitre 2, nous présentons les travaux connexes. Nous présentons ensuite dans le Chapitre 3 notre système de courtage. Nous proposons finalement dans le Chapitre 4 un outil qui permet d'obtenir un retour sur les couts et performances de toutes les stratégies de *provisioning*.

Chapitre 2

État de l'art

L'objectif de cette thèse est de proposer un système de courtage proposant à l'utilisateur :

- des stratégies de *provisioning* et d'ordonnancement automatique ;
- un outil afin d'aider l'utilisateur à choisir une bonne stratégie de *provisioning*.

C'est pourquoi notre état de l'art s'articulera autour des trois sections suivantes.

Dans un premier temps, nous présenterons dans la Section 2.1 les stratégies de *provisioning* qui ont pour objectif d'améliorer les objectifs de l'utilisateur. De manière générale, le cloud IaaS implique, pour l'utilisateur, de décider de combien de ressources il a besoin, et pour combien de temps. Dans la pratique, les utilisateurs réalisent souvent l'ajustement du nombre de ressources manuellement. De nombreuses propositions ont été faites pour automatiser cette allocation de ressources. Ces stratégies ont en général pour objectif de réduire le temps d'exécution ou le cout.

Dans un second temps, nous nous intéressons dans la Section 2.2 aux systèmes réalisant le *provisioning* automatiquement pour les utilisateurs. Ce *provisioning* automatique intervient aussi bien dans un contexte de PaaS que d'IaaS. Nous présentons donc les solutions proposées dans ces deux domaines. Dans le contexte du PaaS, le fournisseur propose une ou plusieurs solutions de *provisioning* automatique. Dans le contexte de l'IaaS, l'utilisateur a un choix plus large. Il peut choisir quel système de *provisioning* automatique et quelles stratégies d'ordonnancement et de *provisioning* mettre en place.

Enfin, dans un troisième temps, nous étudions dans la Section 2.3 les simulateurs de cloud. Cette tâche présente un réel défi dans la simulation de la virtualisation et des phénomènes associés.

2.1 *Provisioning* des ressources du cloud

Notre étude des travaux connexes va dans cette section se concentrer sur les décisions qu'un client de l'IaaS peut faire en cherchant un compromis satisfaisant entre le cout et la rapidité de l'exécution d'un workload. La nécessité de choisir entre plusieurs scénarios est mise en lumière par Deelman et al. [14]. Dans cette étude, il est évalué les avantages et les couts qu'implique l'exécution d'une application scientifique en remplaçant les ordinateurs locaux par des ressources d'Amazon EC2 pour les VM et d'Amazon S3 pour le stockage.

Les stratégies de *provisioning* peuvent être divisées en deux catégories principales : d'un côté, certaines stratégies ne s'intéressent qu'au *provisioning*. L'objectif est alors d'évaluer le bon nombre de VM nécessaire pour exécuter l'ensemble d'un workload au moment de sa soumission. De l'autre côté, il y a des stratégies qui s'intéressent au *provisioning* et à l'ordonnancement. Du fait de la nature élastique des clouds IaaS, les algorithmes d'ordonnancement proposés dans ce contexte doivent considérer différents ordonnancements obtenus lorsque des ressources sont ajoutées ou supprimées.

2.1.1 *Provisioning*

Le *provisioning* est un processus d'adaptation dynamique au changement d'un workload. Ainsi, la plupart des stratégies proposées sont *online*. De plus, la plupart de ces stratégies considèrent les deux objectifs que sont le cout et le temps d'exécution. Néanmoins, avec l'avènement du green computing et l'utilisation de l'IaaS, d'autres métriques sont utilisées telles que l'énergie ou les SLA. Les stratégies uniquement de *provisioning* sont principalement utilisées dans un contexte d'applications web. Nous présentons dans cette section des stratégies de *provisioning* permettant d'optimiser différents types d'objectifs.

Une stratégie largement exploitée dans de nombreux travaux de recherche tels que [23, 39, 69] est *1VMperJob*. Il s'agit d'assigner une VM à chaque job soumis. Ce type de stratégie permet de minimiser le temps d'attente.

L'efficacité énergétique est un sujet d'actualité sur lequel Dougherty et al. [16] travaillent. Ils proposent une solution qui a pour objectif de réduire la consommation électrique d'un workload pour le web. Pour cela, ils demandent au développeur de l'application de fournir la configuration logicielle nécessaire sur la VM, la consommation électrique nécessaire pour chaque type de VM et le cout des VM. Ces entrées sont transformées en un problème de satisfaction de contraintes (constraint satisfaction problem) puis un outil de résolution des contraintes est utilisé pour trouver le nombre de VM optimal qui répond au problème.

Une méthode basée sur le *reinforcement learning* est proposée par Xu et al. [74]. Deux méthodes différentes sont mises en œuvre dans le but d'améliorer, à long terme, le compromis entre une bonne utilisation de l'ensemble des ressources et le respect des Service Level Objectives (SLO) des applications. Ces méthodes utilisent un processus de décision markovien. La première consiste à modifier le paramètre du serveur web permettant de définir le nombre maximum de clients par serveur et la seconde joue sur le nombre de VM. De la même façon, Maurer et al. [40] propose une solution afin d'améliorer le respect des SLO des utilisateurs qui utilisent le *raisonnement par cas* (case based reasonning). Le raisonnement par cas consiste à résoudre un problème en se basant sur les problèmes résolus précédemment. Après chaque prise de décision de *provisioning* dans l'exécution d'un workload, les violations des SLA et taux d'utilisation des VM sont stockés. Cette stratégie

est évaluée en utilisant un simulateur ad hoc et des traces provenant d'exécutions réelles. Caron et al. [10] propose également une prédiction de la montée en charge. Ils se basent sur un *pattern matching* qui utilise l'historique des précédentes soumissions. L'évaluation se fait au travers de l'utilisation de trois traces provenant de différents clouds.

Certaines de ces stratégies ont pour objectif d'étendre un cluster local avec des ressources provenant du cloud. C'est par exemple le cas de de Assunção et al. [4] qui proposent des stratégies de *provisioning* et les évaluent par la simulation de traces réelles. Leurs stratégies proposent d'utiliser dans un premier temps des ressources d'un cluster local et d'un cloud extérieur. Ils testent des stratégies classiquement utilisées dans les gestionnaires des ressources locales (Local Resource Management System ou LRMS) telles que les stratégies de backfilling. Leurs conclusions sont que les stratégies naïves augmentent fortement le cout. De plus, l'utilisation du *selective backfilling* permet d'obtenir un bon ratio entre l'amélioration du *slowdown* et l'argent investi dans l'utilisation de ressources du cloud.

2.1.2 *Provisioning* et ordonnancement

De nombreux travaux de recherche proposent également des stratégies de *provisioning* couplées à un algorithme d'ordonnancement. Du fait de la nature intrinsèquement élastique des clouds IaaS, les algorithmes d'ordonnancement utilisés dans ce contexte doivent prendre en compte l'ajout et le retrait dynamique des ressources. Nous présentons dans cette section les travaux relatifs aux stratégies de *provisioning* et d'ordonnancement et les comparons à nos stratégies.

L'objectif d'étendre un cluster local avec des ressources provenant du cloud concerne également certaines stratégies de *provisioning* et d'ordonnancement. Marshall et al. [39] proposent des stratégies implémentées au sein du logiciel Nimbus [31]. Dans cet article, les stratégies utilisent également les ressources d'un cluster local dans un premier temps puis exportent les calculs supplémentaires sur un cloud. Ces stratégies sont conçues pour répondre à différents patterns d'arrivée des jobs. De la même manière, Perez et al. [54] se placent du point de vue d'un propriétaire de grille qui reçoit des requêtes d'utilisateurs et qui souhaite étendre les capacités de sa grille en louant des ressources sur le cloud. Ils proposent une approche qui utilise le *reinforcement learning* afin de provisionner et ordonnancer les ressources. Leur stratégie de *provisioning* a un double objectif : garantir l'équité entre les utilisateurs et la réactivité des requêtes. L'objectif est de maximiser pour chaque job j soumis, W_j tel que : $W_j = \frac{\text{execution\_time}_j}{\text{execution\_time}_j + \text{waiting\_time}_j}$. L'évaluation se fait par la simulation sur une trace d'exécution provenant de la grille EGEE et une trace générée suivant une loi de Poisson pour une charge du système très élevée de 0,99. En plus d'étendre un cluster, la stratégie de Kijsipongse et Vannarat [33] ajoute une contrainte budgétaire imposée par l'utilisateur. Leurs stratégies de *provisioning* consistent à louer des instances dans la limite d'un certain budget ou d'un nombre maximal d'instances. Ces

deux stratégies se basent sur le nombre de jobs en attente. L'ordonnancement se fait en priorité sur les machines du cluster local disponibles puis sur les VM du cloud.

Les stratégies à contrainte budgétaire ont également été largement étudiées. Ainsi, Silva et al. [61] proposent une heuristique qui a pour objectif de maximiser le speedup en respectant un budget. Oprescu et Kielmann [51] proposent une stratégie pour provisionner les ressources pour l'exécution de *bag-of-tasks* de durée inconnue en respectant une contrainte budgétaire. Ils estiment la durée des jobs en se basant sur une analyse statistique des soumissions précédentes. Fard et al. [19] proposent deux algorithmes de *provisioning* pour les *workflows* qui améliorent le *makespan* ou le cout. Le premier algorithme détermine le *makespan* optimal en louant une VM par job puis en appliquant des algorithmes d'ordonnancement qui jouent sur la date de démarrage des VM et sur le remplissage des trous sur les VM sans affecter le *makespan*, mais en réduisant le cout. Le second algorithme est à contrainte budgétaire avec un *makespan* quasi optimal. Dans un premier temps, on prend la solution optimale pour le *makespan* offerte par le premier algorithme. Puis avec d'autres algorithmes d'ordonnancement, ils sélectionnent la solution qui réduit le plus le cout sans affecter le *makespan*.

Le et al. [35] proposent quant à eux une stratégie de *provisioning* à échéance contrainte. Ils introduisent une métrique appelée *temps d'intervalle moyen* qui indique le temps moyen entre deux jobs retirés de la file d'attente. En utilisant cette métrique, le problème du *provisioning* est modélisé selon une chaîne de Markov dans laquelle chaque VM est une file M/M/1. À la suite de ce *provisioning*, trois algorithmes d'ordonnancement sont exécutés. Le premier est le classique First-come, first-served. Le deuxième donne la priorité aux jobs les plus courts. Et le troisième favorise les jobs dont l'échéance est la plus proche. Leur évaluation par la simulation leur permet de conclure que FCFS est l'algorithme qui permet de respecter les SLA.

Enfin, dans le même esprit de nos stratégies, certaines stratégies de *provisioning* se basent sur les caractéristiques des jobs et l'état du système pour prendre une décision de *provisioning*. C'est le cas de Villegas et al. [72] qui proposent des stratégies de *provisioning* inspirées de celles proposées par Marshall et al. [39]. Ils évaluent leurs stratégies par la simulation ainsi que par l'exécution sur un petit cloud privé. Cette évaluation se fait sur des jobs très courts (47 s en moyenne) qui correspondent à des workloads MapReduce. Duong et al. [17] proposent des stratégies de *provisioning* qui prennent leur décision en fonction soit du temps d'exécution des jobs, soit du nombre de jobs en attente, ou d'un mélange de ces informations.

2.1.3 Conclusion

Nous notons qu'un grand nombre de stratégies est proposé par la communauté scientifique. Ces stratégies de *provisioning* permettent d'exécuter aussi bien des applications

web que des calculs intensifs tels que les *bag-of-tasks* et les *workflows*. Nous mettons en lumière deux grandes catégories de stratégies. La première comprend des stratégies qui ne se concentrent que sur le *provisioning*. Il s'agit pour la plupart de stratégies utilisées pour des applications web ou des calculs très courts qui nécessitent une bonne réactivité. La deuxième correspond aux stratégies qui réalisent aussi bien le *provisioning* que l'ordonnancement. Ces stratégies sont particulièrement intéressantes dans le cas où les utilisateurs connaissent la durée de leurs jobs au moment de la soumission. Nous proposons des stratégies dans ces deux catégories.

2.2 Intégration des stratégies à des plateformes existantes

Nous nous intéressons dans cette section aux solutions existantes permettant la mise en œuvre des stratégies de *provisioning* automatique présentées dans la section précédente. Ce type de travaux peut être découpé en deux catégories distinctes. Dans le contexte du PaaS, le fournisseur doit proposer des solutions pour provisionner automatiquement les ressources afin d'exécuter les codes des clients. Nous présentons dans la Section 2.2.1 les solutions proposées par les chercheurs et les industriels.

Dans le contexte de l'IaaS, le fournisseur n'a pas la nécessité de proposer une solution afin d'aider ses clients à provisionner ses ressources. Nous étudions les nombreuses solutions conçues à cette fin dans la Section 2.2.2.

2.2.1 *Provisioning* automatique dans des systèmes PaaS

De nombreux projets tels qu'Aneka [68], mOSAIC [55], ConPaaS [56], COMP Superscalar [65] et JCloudScale (anciennement CloudScale) [36] ainsi que des plateformes commerciales telles que Google App Engine [24] sont de type PaaS et proposent donc des mécanismes de *provisioning* automatique. Les plateformes de type PaaS présentent deux contraintes pour l'utilisateur qui n'existent pas dans notre proposition.

Premièrement, ces plateformes masquent totalement l'accès aux machines virtuelles. Ceci impacte les utilisateurs qui souhaitent bénéficier des avantages du PaaS tout en gardant un certain contrôle sur la plateforme virtuelle.

Deuxièmement, ces plateformes restreignent l'utilisateur à des API, langage de programmation ou modèle de programmation spécifique.

COMP Superscalar [65] est une plateforme ainsi qu'un modèle de programmation basé sur des annotations Java. Au moment de l'exécution du programme sur cette plateforme, le *provisioning* puis l'ordonnancement sont automatiquement exécutés. Cette plateforme propose une unique stratégie de *provisioning* qui loue une VM par tâche à exécuter. Il est possible de définir un seuil au-delà duquel il n'est plus possible de louer des VM. Les

tâches sont alors mises en attente le temps qu'une VM soit disponible pour l'exécution d'une nouvelle tâche. Cette stratégie n'est cependant pas personnalisable.

Plus récemment, Leitner et al. ont proposé JCloudScale [36], une autre plateforme permettant le déploiement d'application Java. Leur modèle permet à l'utilisateur d'imposer une contrainte de cout à ne pas dépasser lors du *provisioning*. Différentes stratégies de *provisioning* peuvent facilement être implémentées par l'utilisateur en se basant sur les données de monitoring regroupées par JCloudScale (utilisation du CPU et de la RAM, espace libre du disque dur et données réseau transférées).

Aneka [68] est une plateforme pour l'exécution d'applications .NET dans le cloud. Pour cela, l'utilisateur écrit son application à l'aide du SDK fourni. Le langage à utiliser est donc n'importe quel langage supporté par .NET. La stratégie de *provisioning* par défaut est d'utiliser une VM par job soumis. Il est cependant possible de personnaliser cette stratégie. Vecchiola et al. [69] propose par exemple une stratégie à échéance contrainte intégrée à la plateforme Aneka.

mOSAIC [55] est une plateforme permettant l'exécution d'application sur plusieurs clouds. Ces applications peuvent être écrites dans différents langages. La stratégie de *provisioning* est définie par deux fichiers écrits par l'utilisateur. Un premier décrivant les besoins en terme de stockage, communication et calcul. Et un second décrivant les SLA voulus par l'utilisateur écrit au format standard WS-Agreement [3].

ConPaaS [56] est une des parties de la plateforme Contrail. Cette plateforme a pour objectif de former une fédération de clouds et d'offrir des outils open source pour la gestion d'une telle fédération. Au sein de ce projet, ConPaaS est la partie fournissant un environnement PaaS au sein de Contrail. Chaque service ConPaaS correspond à une ou plusieurs VM affectées à un utilisateur. Il n'est pas possible de partager ces VM avec d'autres utilisateurs ou d'autres services du même utilisateur. Mais ConPaaS fournit une stratégie de *provisioning* automatique des ressources afin de garantir des SLA à un cout minimum. ConPaaS est plutôt orienté vers l'exécution d'application web dans le cloud.

Enfin, les solutions commerciales telles que Google App Engine [24] proposent également un *provisioning* automatique des ressources. Cependant, cette plateforme ne supporte que quelques langages de programmation (Python et Java). De plus, en limitant la durée des jobs à 60 secondes[1], elle est principalement orientée vers les applications web.

2.2.2 Systèmes de courtage dans des systèmes IaaS

Le cloud computing a dans un premier temps été largement utilisé pour exécuter des applications web. Ainsi, de nombreux travaux de recherche proposent des systèmes de courtage qui se focalisent sur l'exécution de requêtes web dans le cloud. Nous présentons

1. au 9 février 2015 `https://cloud.google.com/appengine/docs/java/requests#Java_The_request_timer`

ces travaux, car les requêtes web peuvent être assimilées à des jobs indépendants, courts, à forte réactivité, qui peuvent s'exécuter en parallèle.

La communauté scientifique a également été active dans la recherche de systèmes de courtage pour la prise en charge d'applications scientifiques. Ces travaux sont plus proches du nôtre. Nous présentons des solutions provenant aussi bien de la communauté scientifique que de l'industrie.

Pour l'ensemble de ces travaux, nous présentons les systèmes de courtage ainsi que les stratégies de *provisioning* qu'ils appliquent.

Systèmes de courtage orientés pour le web

STRATOS [53] est un broker multi-cloud qui permet de choisir les meilleures ressources dans le meilleur cloud en fonction de la topologie de l'application et d'objectifs utilisateur. Ce broker a été testé sur Amazon EC2 et Rackspace afin de montrer ses capacités multi-cloud. Dans ce cas, le *provisioning* n'est pas complètement automatique. L'utilisateur doit fournir un fichier décrivant la topologie de son application incluant l'ensemble des services qu'il souhaite exécuter sur des machines différentes. La description des objectifs de l'utilisateur se fait en utilisant des métriques décrites par Zachos et al. [76] et ne sert qu'à sélectionner le type d'instance voulu par l'utilisateur.

Kingfisher [60] est également un broker côté client qui se focalise sur le *provisioning* de ressources virtuelles avec une stratégie *cost-aware*. Leur stratégie calcule dans un premier temps le taux maximum de requêtes qu'une VM peut exécuter. Puis ils formulent et résolvent le problème de trouver les bons types de VM et leurs nombres en un problème d'optimisation linéaire.

Federated Cloud Management (FCM) [32] est un système de courtage multi-cloud orienté vers les services web sans états. Ce système prend la décision de *provisioning* en fonction de plusieurs critères : nombre de requêtes, monitoring, et les SLA de l'utilisateur. Il n'est cependant pas possible de personnaliser la stratégie de *provisioning*.

Enfin, CompatibleOne [75] est un modèle pour décrire les besoins de l'utilisateur (CORDS) et un broker pour provisionner et déployer une application (ACCORDS). ACCORDS propose un moyen interopérable de démarrer des instances dans le cloud. L'utilisateur peut personnaliser la stratégie de *provisioning* en donnant un document composé de CORDS et de WebService Agreement [3].

Systèmes de courtage orientés pour les calculs scientifiques

Il existe différentes approches dans la création d'un système de courtage pour le calcul scientifique. Montero et al. [46] présente un environnement pour des calculs de type bag-of-tasks. Ce travail étend les capacités d'un cluster local en utilisant un cloud externe. Alors que l'objectif du travail de de Assunção et al. [4] est de comparer différentes stratégies de

provisioning, ce travail ne propose que d'allouer un nombre fixe de VM afin d'exécuter un benchmark. L'objectif est ici de réaliser un modèle de performance pour l'exécution d'applications HTC sur une telle plateforme.

TorqueCloud [63] est un système de courtage qui ajoute des fonctionnalités d'ordonnancement dynamique des tâches au logiciel Torque. Ce dernier est largement utilisé dans des clusters afin d'y ordonnancer les tâches. Dans ce travail, Song et al. [63] proposent également une stratégie de *provisioning* à échéance contrainte appelée IdleCached. Cette stratégie ordonnance les tâches dont l'échéance est la plus proche sur des VM inoccupées. S'il n'y a pas assez de VM pour traiter l'ensemble des tâches, ils démarrent autant de VM que de tâches restantes.

E-clouds [41] est une plateforme SaaS pour le calcul scientifique qui utilise une infrastructure IaaS pour exécuter ses calculs. Ce système propose à l'utilisateur de choisir, au travers d'une page web, une application scientifique dans une liste finie proposée par les administrateurs. Après configuration de l'application par l'utilisateur, e-clouds présente à l'utilisateur une estimation du cout et de la durée de l'exécution. Enfin, e-clouds démarre l'exécution de l'application en provisionnant automatiquement les ressources. Cependant, il n'existe pas de mécanisme de *provisioning* automatique des VM. Le nombre de VM à démarrer est fixe au cours du temps et déterminé par l'utilisateur.

Villegas et al. proposent SkyMark [72], un outil d'évaluation de la performance de stratégies de *provisioning* et d'ordonnancement. Il est capable de soumettre et de monitorer l'exécution de *bag-of-tasks* sur les clouds compatibles EC2 et OpenNebula. Un simulateur qui duplique les fonctionnalités de SkyMark a également été développé. Il ne possède cependant pas la maturité de SimGrid [11]. Il est de plus important de noter qu'aucun de ces deux outils (SkyMark et le simulateur associé) n'est disponible publiquement.

Des sociétés commerciales proposent également des systèmes de courtage. Rightscale [58] permet aux utilisateurs de déployer et monitorer des VM dans plusieurs clouds. Le client définit les règles qui vont automatiquement déclencher de nouvelles VM en fonction d'un seuil sur différentes métriques. D'autres sociétés proposent le même genre de service tel qu'Amazon Elastic Beanstalk [1], CloudFoundry [12] et Windows Azure [73].

2.2.3 Conclusion

Nous pouvons noter qu'un grand nombre de systèmes de courtage a été conçu, aussi bien pour provisionner les applications web que les applications scientifiques. Cependant, aucun ne remplit complètement les objectifs fixés pour notre courtier. Nous récapitulons dans la Table 2.1 pour chaque projet, quels objectifs sont remplis.

TABLE 2.1 – Comparison of various IaaS and PaaS brokering solutions

	Requires API	*A priori* simulation	Customized *provisioning* strategy	Programming model	Open source / Publicly available
[65]	Yes	No	No	Workflows	Yes
[55]				Bag-of-tasks, Workflows, MapReduce, web apps	Partially (Not the broker)
[24]				web	No / Yes
[56]			Yes (QoS requirements)	web, MapReduce, Bag-of-tasks, High-Throughput Computing	Yes
[68, 69]			Yes	Bag-of-tasks, MapReduce	No / Yes
[36]				Applications Java	Yes
[32, 60]	No	No	No	web	No
[41]				Fixed set of applications	
[46]				Bag-of-tasks (HTC)	
[63]				Bag-of-tasks	
[1, 58, 73]				web	No / Yes
[12]					Yes
[53]			Yes (Weighted objectives)	web	No
[75]			Yes	web	Yes
[72]		Yes	Yes	Bag-of-tasks	No
Schlouder	No	Yes	Yes	Bag-of-tasks, Workflows	Yes

2.3 Simulation des clouds

Malgré la présence de stratégies de *provisioning*, certains systèmes que nous avons présentés laissent à l'utilisateur le choix de certains paramètres qui impactent grandement le résultat du *provisioning*. Ainsi, afin d'aider l'utilisateur à choisir une stratégie de *provisioning* qui corresponde à ses besoins, nous souhaitons lui proposer un simulateur de cloud intégré à notre système de courtage capable de simuler l'exécution du workload rapidement tout en donnant une prédiction correcte du déroulement de l'exécution pour chaque stratégie de *provisioning*. Des simulateurs de cloud sont proposés dans la littérature. Les projets iCanCloud [48], CloudSim [6], DCSim [67] et GroudSim [52] sont orientés vers la simulation d'un cloud complet.

iCanCloud [48] est un simulateur de cloud. Il a pour objectif de prédire le compromis cout / performance d'une stratégie de *provisioning*. Ce simulateur est orienté vers une simplicité d'utilisation en proposant une interface graphique pour faciliter la prise en main de la simulation. La validation se fait en utilisant une application d'astronomie dont les auteurs simulent l'exécution sur la plateforme Amazon EC2. Ils comparent ensuite les résultats de la simulation avec les résultats fournis par la modélisation de l'exécution de l'application. Aucune analyse n'est proposée pour comparer les résultats de la simulation avec ceux d'une exécution réelle.

CloudSim [6] est un simulateur initialement basé sur le simulateur de grille GridSim [64]. Il a depuis été réécrit afin de ne plus avoir de dépendance avec ce projet. Il permet de simuler tous les éléments d'un cloud du point de vue d'un fournisseur de cloud, de l'algorithme de placement des VM à la consommation électrique du centre de calcul. Ce simulateur est très utilisé et possède donc une grande communauté. Ce simulateur est cependant connu pour posséder un modèle réseau peu précis [71] et des problèmes de passage à l'échelle.

DCSim [67] est un simulateur orienté vers le fournisseur de cloud. Son objectif est de permettre à ces fournisseurs de tester les actions classiques de gestion des data centres tels que des algorithmes de consolidation ou de migration des VM.

Enfin, GroudSim [52] est le travail le plus proche du nôtre. Il s'agit d'un simulateur de grilles et de clouds pour applications scientifiques écrit en Java. GroudSim est intégré à la grille AKSALON afin de permettre d'exécuter au choix sur la grille ou sur le simulateur. L'évaluation se fait en comparant l'exécution d'un code sur GroudSim, GridSim puis sur la grille Austrian Grid. La comparaison entre les deux simulateurs ne se fait pas sur la correction des résultats, mais sur la rapidité d'exécution de la simulation pour laquelle GroudSim est bien plus rapide lorsque le nombre de jobs parallèles augmente (jusqu'à 100). La comparaison avec l'exécution réelle sur la grille montre une différence de 10% des temps d'exécution. Ce simulateur n'est pas proposé au téléchargement.

Conclusion

En conclusion sur les simulateurs de systèmes distribués, nous pouvons dire que la plupart d'entre eux possèdent une modélisation du réseau *flow-level*. Velho et al. [71] ont montré que l'implémentation de ce modèle dans ces travaux n'a pas été suffisamment validée. De plus, aucun d'entre eux ne propose de comparaisons approfondies entre les résultats d'une exécution réelle et ceux d'une exécution par le simulateur.

Chapitre 3

Schlouder : un système de courtage pour les provisionner tous

Contributions principales

- Développement d'un système de courtage qui automatise le *provisioning*.
- Comparaison des capacités de notre système de courtage avec l'exécution d'une application sur une grille en production.

3.1 Introduction

Après avoir testé des stratégies de *provisioning* dans un environnement maîtrisable dans le Chapitre 1, nous souhaitons confronter notre travail sur les stratégies de *provisioning* dans un environnement réel. Pour ce faire, nous plaidons pour l'installation d'un système de courtage présent entre l'utilisateur et le fournisseur d'IaaS tel que montré sur la Figure 3.1. Dans ce système, les utilisateurs soumettent leurs jobs au courtier qui s'occupe de faire le lien avec le fournisseur d'IaaS. Ce logiciel connaît l'état de la plateforme virtuelle et est capable d'y ordonnancer les jobs.

Nous présentons dans ce chapitre notre système de courtage nommé Schlouder[1]. Il a

1. `http://schlouder.gforge.inria.fr/`

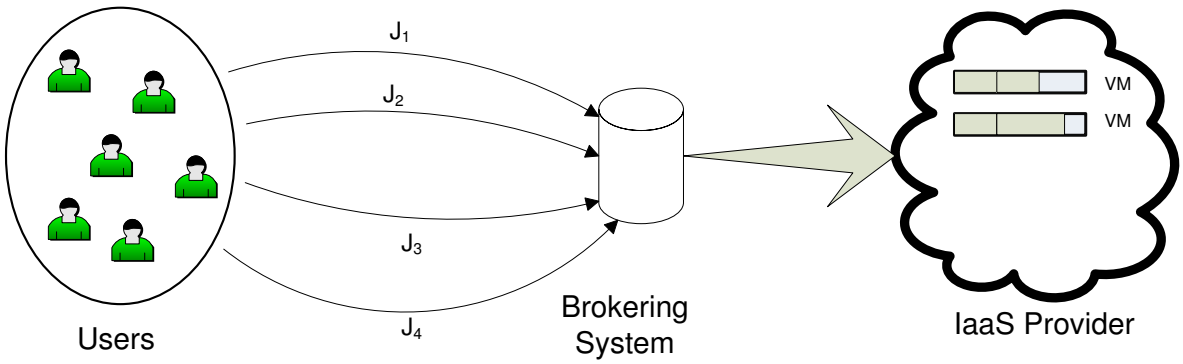


Figure 3.1 – Système de courtage pour la réalisation du *provisioning*

pour principal objectif d'automatiser le *provisioning* et l'ordonnancement pour l'exécution de calculs dans un cloud IaaS. Nous nous plaçons dans un cadre *online* c'est-à-dire que le *provisioning* et l'ordonnancement s'effectuent dynamiquement lorsque les jobs sont soumis, sans connaissance des soumissions futures. Le *provisioning* est calculé dans Schlouder en fonction d'une stratégie choisie par l'utilisateur parmi une liste implémentée au sein du logiciel.

Nous décrivons dans un premier temps l'architecture de notre courtier dans la Section 3.2. Nous y mettons en évidence l'importance d'une architecture ouverte pour ce type de logiciel.

Nous expliquons dans la Section 3.3 le dispositif expérimental mis en place : les stratégies de *provisioning* retenues pour cette évaluation, les applications et les IaaS publics et privés utilisés pour exécuter ces applications.

Enfin, nous présentons les métriques d'intérêt et l'évaluation de Schlouder dans la Section 3.4.

3.2 Schlouder : un système de courtage

Schlouder se positionne entre l'utilisateur et l'infrastructure IaaS (cf Figure 3.2). Du côté utilisateur, Schlouder fournit une interface pour qu'il soumette des jobs, d'une façon similaire à la soumission à travers n'importe quel *batch scheduler*. Du côté infrastructure, Schlouder s'interface avec :

- un *batch scheduler* ;
- et un *cloud kit*. Nous appelons cloud kit le logiciel responsable de la gestion d'un IaaS, privé ou public.

Ces deux interfaces sont très modulaires et peuvent s'adapter à n'importe quel logiciel. En particulier, l'interface « cloud kit » permet d'ajouter plusieurs modules afin de gérer une fédération de clouds IaaS. Dans l'exemple de la Figure 3.2, le *batch scheduler* est Slurm et les clouds kits sont OpenStack et BonFIRE. D'un point de vue technique, ce logiciel a été développé en Perl et comprend 67 modules pour un total de près de 10 000 lignes de codes.

3.2.1 L'architecture de Schlouder

Nous décrivons dans cette section l'architecture de notre système de courtage. Ses fonctions principales sont :

(1) de décider quand démarrer ou stopper des VM, et quand réutiliser une VM déjà démarrée ;

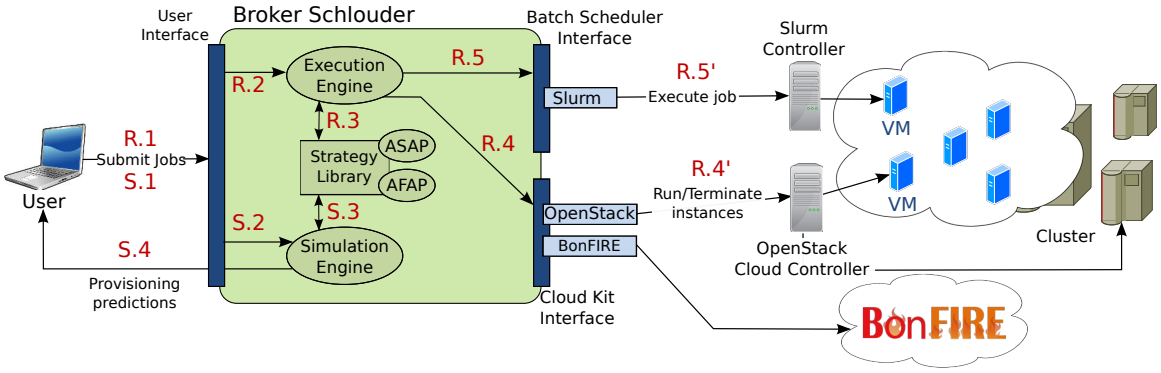


FIGURE 3.2 – L'architecture de Schlouder

(2) d'exécuter les jobs au travers de l'interface du *batch scheduler* ;

(3) de simuler l'exécution des jobs soumis dans le but de fournir à l'utilisateur une prédiction des conséquences du choix des différentes stratégies de *provisioning*.

La Figure 3.2 décrit l'architecture de Schlouder (le carré central) et son environnement. Schlouder a deux modes de fonctionnement : les soumissions réelles, prises en charge par l'*Execution Engine* et les soumissions pour simulation, prises en charge par le *Simulation Engine*. Ces deux moteurs utilisent les stratégies de *provisioning* et d'ordonnancement stockées dans la *Strategy Library*. Les rôles de ces différents composants sont expliqués dans les sous-sections suivantes.

Strategy Library (SL)

Schlouder propose un framework ouvert pour décrire des stratégies de *provisioning* afin de gérer les jobs soumis. Chaque stratégie est une classe Perl qui utilise une API simple pour accéder aux caractéristiques et à l'état de la plateforme virtuelle et des jobs. Nous y avons implémenté l'ensemble des stratégies présentées dans le Chapitre 1.

Execution Engine (EE)

L'Execution Engine est le chef d'orchestre du système de courtage. Lorsqu'une requête pour l'exécution d'un job est reçue, l'EE communique avec les différents modules pour : recevoir les décisions de *provisioning*, modifier la taille de la plateforme et ordonnancer les jobs.

L'utilisateur peut à tout moment demander l'état de la plateforme et de l'exécution des jobs sous deux formes différentes : soit sous la forme d'un fichier au format JSON pour un traitement simplifié par ordinateur (cf. Annexe C.1), soit au format texte pour faciliter la lecture par un œil humain (cf. Annexe C.2).

Simulation Engine (SE)

Le Simulation Engine est un module intégré à Schlouder capable d'exécuter une simulation. Il nécessite le workload à exécuter et une description des ressources de l'IaaS utilisées. Le workload est fourni de manière identique à une exécution réelle. Il est constitué de la liste des jobs à exécuter avec leurs caractéristiques (c'est-à-dire la durée, la taille des entrées/sorties et les dépendances pour tous les jobs).

La sortie principale du simulateur est le cout et le *makespan* pour chaque stratégie de *provisioning* disponible. Cependant, il peut également fournir des informations très précises sur le déroulement de la simulation tel que l'affectation des jobs sur les VM. Ces informations s'avèrent très utiles lorsqu'il s'agit d'analyser et d'améliorer des stratégies de *provisioning*.

Ce module est expliqué en détail au Chapitre 4.

3.2.2 Cas d'utilisation

L'ordre d'appel des composants de Schlouder est indiqué sur la Figure 3.2 et dépend du type de soumission : exécution réelle ou simulation. Les étapes préfixées par S sont utilisées dans le cas d'une simulation : soumission par le client de la description du job à Schlouder (S.1) ; transfert de la requête au SE et initialisation de la simulation (S.2) ; simulation de l'exécution en utilisant les stratégies de la SL (S.3) ; les prédictions sont renvoyées à l'utilisateur (S.4).

Les étapes préfixées par R sont utilisées dans le cas d'une exécution réelle : la soumission par le client de la description du job est identique à l'étape S.1 (R.1) ; transfert de la requête à l'EE (R.2) ; calcule de la décision de *provisioning* et d'ordonnancement en utilisant la stratégie choisie par l'utilisateur (R.3) ; l'EE demande au cloud kit d'ajuster la taille de la plateforme (démarrage ou arrêt des VM) conformément à la décision de *provisioning* (R.4) ; l'EE demande au *batch scheduler* d'ordonnancer les jobs sur les VM conformément à la décision d'ordonnancement (R.5).

3.2.3 Aspects pratiques

Pour compléter cette présentation générale de l'architecture de Schlouder, nous expliquons maintenant comment l'utilisateur interagit avec le système d'un point de vue pratique. Tout d'abord, un utilisateur doit préparer son environnement en (1) incluant les exécutables de l'application dans l'image de VM, (2) écrivant un script shell qui utilise les services de l'IaaS (p.ex. le stockage) et appelle les exécutables de l'application (tout comme il le ferait avec des systèmes traditionnels de *batch scheduler*). Puis l'utilisateur communique avec Schlouder au travers du script client. La soumission s'effectue avec la commande :

```
schlouder-client [--se] [--monitoring[=all|network-only|cpu-only]] <workload>
```

avec l'option `--se` qui indique au serveur d'utiliser le SE à la place d'effectuer une soumission réelle sur les nœuds de calcul. L'option `--monitoring` indique au serveur d'exécuter le job de monitoring sur l'ensemble des VM démarrées.

L'argument `workload` est un fichier texte. Chaque ligne contient le nom du script shell à exécuter sur une VM avec ses arguments. Un exemple de description de workload est montré dans le Listing 3.1. L'en-tête de chaque script shell contient diverses informations utilisées par le *batch scheduler*, la stratégie de *provisioning* ou le SE : le nom du job et le temps d'exécution sont obligatoires. L'utilisateur peut éventuellement indiquer la quantité de données en entrée et en sortie, la prédiction du temps d'exécution de l'application, et les dépendances du job. Un exemple d'en-tête est montré dans le Listing 3.2.

```
/tmp/run_1.sh hrs /tmp/input/X001825CYC+LM_1.mgf
/tmp/run_2.sh hrs /tmp/input/X001825CYC+LM_2.mgf
/tmp/run_3.sh hrs /tmp/input/X001826CYC+LM_1.mgf
```

Listing 3.1 – Exemple de description de workload

```
#!/bin/bash
#SBATCH --job-name=hrs_X001825CYC+LM_1.mgf
#SBATCH -D /tmp
#SBATCH --time=188
#PSM --data_in=20365749
#PSM --data_out=8960672
#PSM --runtime_prediction=188
```

Listing 3.2 – Exemple d'en-tête de script

À la fin de l'exécution, qu'elle soit réelle ou simulée, un fichier détaillé au format JSON est fourni à l'utilisateur. Nous en présentons un exemple en Annexe C.1. Les clés en italiques correspondent à des informations qui n'ont pas été ajoutées par Schlouder. Elles ont été ajoutées post-mortem par un script d'analyse des logs d'exécution. Nous y trouvons des informations concernant :

- l'expérience : la date de début de l'expérience, la version de Schlouder et un texte libre de description de l'expérience.
- les VM (nœuds) : les identifiants de la VM fournis par le fournisseur de l'infrastructure et par Schlouder ainsi que l'IaaS sur lequel la VM s'exécute. Sont également présents les temps de démarrage prédits et réels, ainsi que les dates de démarrage et d'arrêt de la VM. Chaque VM a également une liste des jobs qui y ont été exécutés.

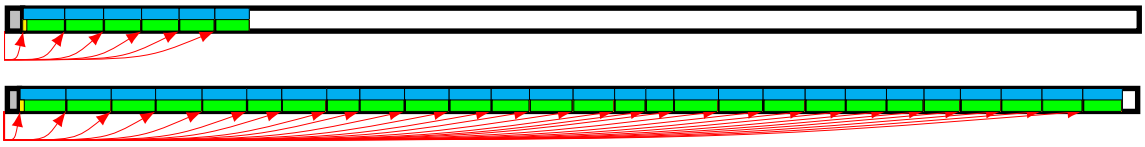


FIGURE 3.3 – Exemple de figure créée par le script de visualisation des BTU

- les jobs : Schlouder fournit les identifiants du job en interne et du point de vue du *batch scheduler* ainsi que la ligne de commande utilisée pour appeler le job. Sont également présents les *walltimes* prédits et réels, les données en entrée du Simulation Engine ainsi que les différentes dates de changement d'état du job. Enfin, Schlouder fournit la stratégie de *provisioning* utilisée et la liste des dépendances du job. Un script analyse également les logs afin d'ajouter les informations en italiques au fichier en sortie. Il s'agit des différents temps du cycle de vie d'un job (cf. Section 3.4.1) et de la taille des fichiers en entrée et en sortie du script.

Schlouder propose également un script de visualisation des BTU produisant un diagramme de Gantt contenant l'ensemble des VM ainsi que les jobs qui se sont exécutés dessus. La Figure 3.3 en montre un exemple. Chaque ligne est une VM et le rectangle grisé au début de chaque VM est le temps de démarrage. Les autres rectangles sont les jobs qui s'exécutent sur la VM. Le point de départ des flèches est la date de soumission des jobs et l'autre extrémité est le début de son exécution. Cette figure est très utile pour comprendre les détails de l'exécution dans le but d'affiner les stratégies de *provisioning* ou comprendre le déroulement de l'exécution une fois celle-ci terminée.

3.2.4 Schlouder dans l'écosystème du cloud

Nous répondons dans cette section à la question du positionnement de Schlouder dans l'écosystème du cloud computing. Le cloud computing est un écosystème qui comprend en réalité trois catégories de services : l'IaaS, le PaaS et le SaaS. Chaque service de cet écosystème s'appuie sur des composants gérés par le client dans la couche précédente. L'IaaS isole complètement les clients et permet un accès bas niveau à l'administration d'une infrastructure virtuelle. Dans le PaaS, les clients n'ont pas accès à l'infrastructure qui est gérée par le fournisseur. L'isolation n'y est cependant pas aussi forte que dans l'IaaS puisqu'il n'y a pas de garantie qu'une application ne partagera pas de ressource (par exemple une VM) avec d'autres clients. Les fournisseurs proposent un environnement pour développer et déployer des applications en utilisant des outils, bibliothèques et services spécifiques. Cacher la complexité de l'infrastructure par l'automatisation peut être adapté dans certains cas. Dans d'autres situations, avoir le contrôle sur la façon dont le *provisioning* s'effectue, ou accéder à des technologies qui n'existent qu'au niveau IaaS est nécessaire.

TABLE 3.1 – Les stratégies de *provisioning FirstFit* et *AFAP* avec leurs paramètres pour l'Algorithme 1.

strategy	$eligible(v, j)$ returns $true$	$optimum(C)$ returns $v \in C$ such that ...	comment
ASAP/1VMperJobPlus	if $q_v = \emptyset$	any	Fastest/Most expensive
FirstFit	if $c(s_v - b_v) = c(s_v - b_v + r_j)$	any	Regular bin packing strategy
AFAP	if $i(i_v + r_j) < i(i_v)$	any	

Schlouder est un hybride plus proche de l'IaaS que du PaaS. Il automatise le *provisioning* des VM, fournit une interopérabilité entre les IaaS, et a un outil de recommandation basé sur la simulation qui aide au choix de la stratégie de *provisioning*. Cependant, contrairement au PaaS, il n'impose pas d'outils ou de plateforme de développement, conserve l'isolation des clients au niveau de la VM, et permet d'utiliser des images de VM personnalisées. À contrario, développer une plateforme de PaaS pour le calcul scientifique est l'objet d'autres travaux tel qu'Aneka [68] et COMP Superscalar [65].

3.3 Validation expérimentale

Après avoir présenté Schlouder, nous souhaitons valider son fonctionnement. Pour cela, nous présentons dans cette section la démarche expérimentale que nous avons suivie. Nous avons sélectionné deux applications scientifiques que nous avons exécutées sur un IaaS privé et un IaaS public. Pour chacune des exécutions, nous choisissons parmi deux stratégies aux objectifs opposés. Nous souhaitons ainsi mettre en évidence que ces deux stratégies produisent des scénarios de *provisioning* et d'ordonnancement contrastés. Nous présentons dans cette section les stratégies de *provisioning* utilisées, les applications et les plateformes sur lesquelles nous avons exécuté nos expériences.

3.3.1 Les stratégies de *provisioning*

Suite à l'étude des stratégies de *provisioning* présentées dans le Chapitre 1, nous avons décidé de garder deux stratégies extrêmes : *1VMperJobPlus* et *FirstFit*. Suite à une modification effectuée sur la stratégie *FirstFit*, nous avons choisi de renommer ces deux stratégies : *As Soon As Possible* (*ASAP*) et *As Full As Possible* (*AFAP*). Une modification a également été apportée à l'algorithme de la stratégie *AFAP*. Son objectif est maintenant de réduire le temps d'inactivité des VM à la fin de l'exécution des jobs, sans empêcher la reconduite d'une VM sur plusieurs BTU. En effet, l'objectif affiché des stratégies de bin packing est d'exécuter le job à cout constant. Cependant, la stratégie *FirstFit* ne permet pas l'ordonnancement de job de plus de 3 600 secondes sur une VM déjà déployée. C'est ce cas que nous souhaitons prendre en compte avec *AFAP*.

Afin de formaliser cette modification de l'algorithme, nous définissons la fonction $i(d)$ qui retourne le temps entre la date d et la fin de la BTU. Nous récapitulons dans la

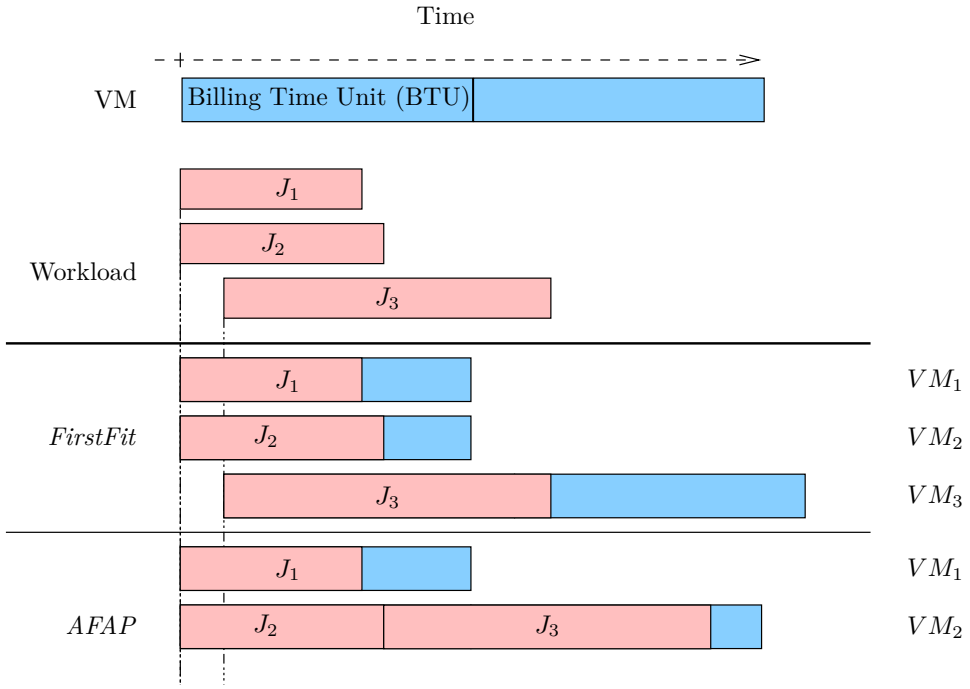


FIGURE 3.4 – Exemple d'application de la stratégie *AFAP* et comparaison avec *FirstFit*

Table 3.1 la définition des fonctions utiles pour l'Algorithme 1.

Nous montrons dans la Figure 3.4 un exemple d'application de cette stratégie que nous comparons avec l'exécution en utilisant la stratégie *FirstFit*. Dans un premier temps, deux jobs, J_1 et J_2 sont soumis et ordonnancés sur deux VM différentes, quelque soit la stratégie. *FirstFit* et *AFAP* se distinguent dans la prise en charge du job J_3. Ce job dure plus de 3 600 secondes. Dans le cas de *FirstFit*, il ne peut pas être exécuté sur des BTU déjà ouvertes. *FirstFit* ordonnance donc ce job sur une troisième VM. En revanche, la stratégie *AFAP* va permettre de réduire le temps d'inoccupation des VM en ordonnançant le job J_3 sur la VM pour laquelle on réduira le plus ce temps. Il s'agit de la VM_2 dans ce cas.

3.3.2 Les applications scientifiques

Nous présentons dans cette section les deux applications que nous avons exécutées. La première, OMSSA, est une application de type *bag-of-tasks* utilisée en protéomique. Nous l'avons sélectionnée, car elle est représentative des applications *bag-of-tasks* suite à une collaboration avec une équipe de protéomistes à l'Institut Pluridisciplinaire Hubert Curien (IPHC) de l'Université de Strasbourg. Des travaux communs avec cette équipe sont présentés dans la Section 3.5. La seconde, Montage, est une application bien connue d'astronomie. Cette application est largement utilisée dans les recherches en informatique sur l'IaaS et les applications scientifiques suite aux travaux précurseurs de Deelman et al. [14].

OMSSA

Il est impossible de parler d'OMSSA, sans introduire d'abord la protéomique, et particulièrement l'analyse MS/MS, technique essentielle pour cette science.

La protéomique est la science qui étudie les protéines des cellules et notamment avec l'objectif d'identifier ces protéines, leur quantité et les éventuelles modifications qu'elles ont pu subir.

La spectrométrie de masse en tandem, également appelée analyse MS/MS, consiste à fragmenter des peptides produits par la digestion enzymatique de protéines. Des milliers de peptides fragmentés sont ainsi obtenus et utilisés pour identifier les protéines présentes dans l'échantillon de départ. Une liste des spectres de masse de tous les peptides est soumise à un algorithme de recherche dans une base de données pour leur identification. OMSSA est un logiciel et un algorithme permettant d'effectuer cette identification. Chaque spectre est ainsi cherché dans une base de données des protéines. Chaque recherche est indépendante des autres. Ainsi, une parallélisation naturelle est de faire de cette application un *bag-of-tasks*.

Nous utilisons quatre jeux de données différents. Ils contiennent jusqu'à 257 762 spectres MS/MS interprétés de la façon suivante : spectrométrie de masse à haute résolution et spécificité enzymatique complète (*hrt*), spectrométrie de masse à haute résolution et semi-spécificité enzymatique complète (*hrs*) et spectrométrie de masse à basse résolution et spécificité enzymatique complète (*brt*). Dans nos expériences, chaque job contient 10 000 spectres pour les recherches *hrt* et *brt* ; et chaque job contient 1 250 spectres pour les recherches *hrs* et *brs*. Dans le but de reproduire nos expériences, l'ensemble des fichiers de configuration que nous avons utilisé est disponible en ligne[2].

La durée des jobs est linéaire en fonction du nombre de spectres présents dans le fichier en entrée, mais dépend fortement du type de recherche. Nous pouvons prédire le temps d'exécution des jobs avec une précision relativement bonne : une régression linéaire nous donne des corrélations de 0,986, 0,985 et 0,970 respectivement pour *brt*, *hrs* et *hrt*.

Montage

La seconde application que nous avons exécutée est Montage Astronomical Image Mosaic Engine [29], appelée plus simplement Montage. L'objectif de cette application est d'assembler des images astronomiques au format Flexible Image Transport System (FITS) en mosaïque. Les astronomes peuvent donc créer une image finale d'une région du ciel d'intérêt, mais trop grande pour être produite en une prise par les outils actuels. Cette application permet également d'utiliser des images provenant de différentes sources pour créer la mosaïque finale.

Les arguments en entrées de Montage sont la région du ciel voulue, la taille de la

2. `http://schlouder.gforge.inria.fr/omssa_configuration/`

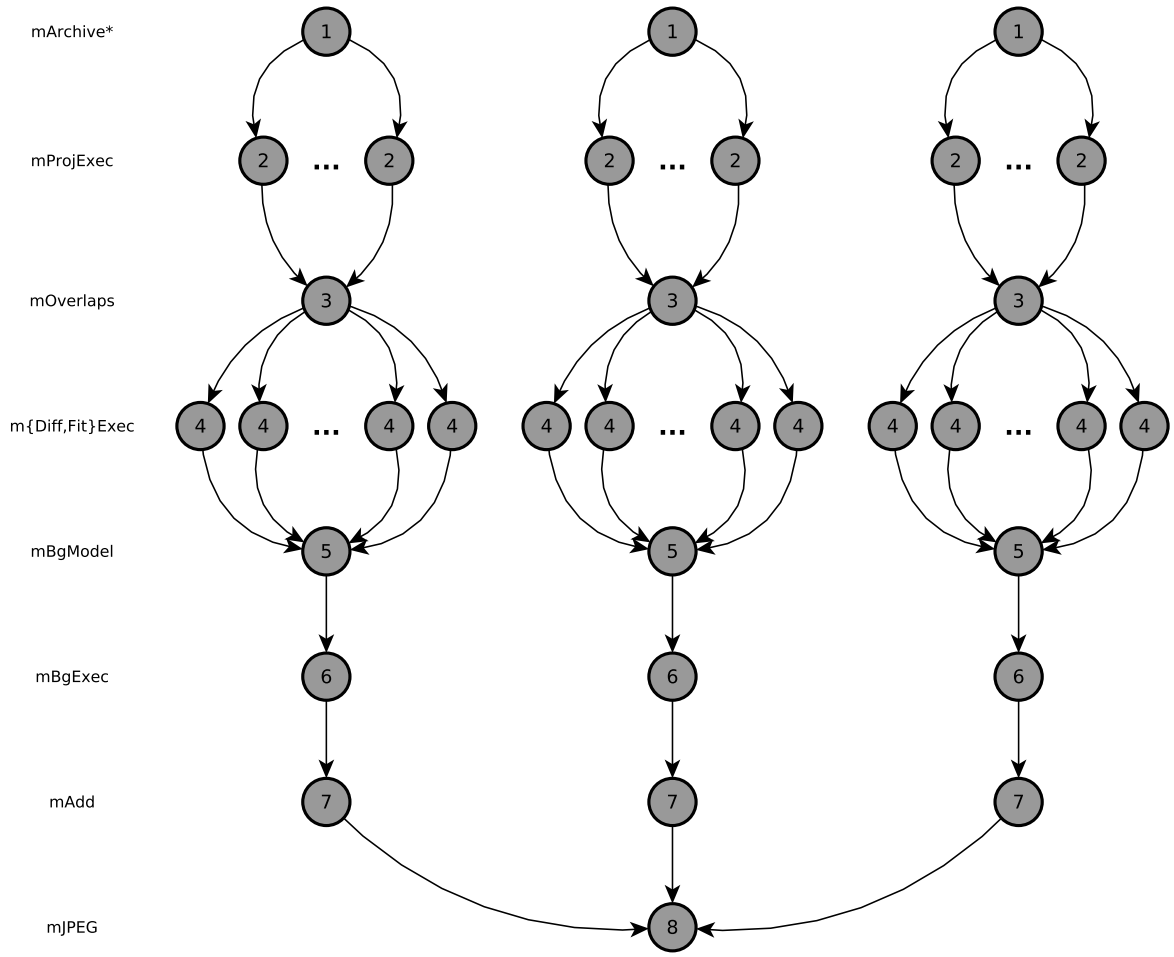


FIGURE 3.5 – Le workflow de Montage

mosaïque en degrés, et d'autres arguments tels que l'archive d'image FITS à utiliser. Dans nos expériences, nous avons utilisé l'archive Two Micron All Sky Survey (2MASS) [28]. Cette application est un *workflow* composé des étapes suivantes. Tout d'abord, les images en entrées sont re-projetées dans l'espace de coordonnées voulues en sortie. Puis, les différences de luminosité sont lissées. Enfin, les images en entrée sont fusionnées pour former l'image de sortie. Dans le but de reproduire nos expériences, l'ensemble des scripts que nous avons utilisés est disponible en ligne[3].

La Figure 3.5 montre un exemple d'un petit *workflow* de Montage. Chaque cercle est un job et chaque arc représente les dépendances de données entre deux jobs. Le chiffre dans chaque cercle représente le niveau du job dans le *workflow*. Tous les jobs à un niveau donné correspondent à un appel d'une commande de Montage, mentionnée sur la gauche de la figure.

Dans nos expériences, nous avons choisi de calculer une mosaïque d'un unique objet astronomique, mais pour différentes tailles. Nous avons arbitrairement décidé de calculer la mosaïque de Pléiades, puisqu'il s'agit de l'objet utilisé par le tutoriel de Montage. Nous

3. `http://schlouder.gforge.inria.fr/montage_configuration/`

TABLE 3.2 – Caractéristiques d'OMSSA et de Montage

		# jobs	walltime (s) min	avg	max	total I/O size	communication to computation ratio	# dependencies
OMSSA	*brt*	33	86	152	398	$1,2GB$	2%	0
	hrt	34	6	16	199	$514MB$	16%	0
	hrs	65	66	148	411	$1,3GB$	1%	0
Montage	1x1	163	4	18	167	$8GB$	94%	288
	2x2	467	6	49	593	$30,9GB$	97%	863
	3x3	993	4	60	1 340	$69,8GB$	98%	1 861

avons choisi de réaliser des mosaïques de tailles 1, 2 et 3. Avec ces tailles, le nombre de jobs parallèles aux niveaux 2 et 4 du *workflow* varie de 21 à 1 479. Montage est une application intensive en données. Les images en entrée, les fichiers intermédiaires créés au cours de l'exécution du *workflow*, et l'image en sortie ont une taille allant jusqu'à $69,8$ GB. Par la suite, nous appelons NxN, avec N un entier de 1 à 3, une exécution de Montage de taille N.

Nous résumons dans la Table 3.2 quelques caractéristiques de ces applications afin de mettre en évidence que nos expériences correspondent à deux types d'utilisation des ressources diamétralement opposés. OMSSA est une application *bag-of-tasks* de calcul intensif alors que Montage est un *workflow* communication intensive. Les *walltimes* correspondent à la calibration de Schlouder sur la plateforme icps-os-cloud. Cette calibration a été utilisée, quelle que soit la plateforme.

3.3.3 Les plateformes d'exécution

L'IaaS privé

Nous présentons sur la Figure 3.6 l'architecture de notre IaaS privé. Il est composé de deux serveurs locaux — icps-gc-1 et icps-gc-2 —, chacun avec 24 Go de mémoire vive et deux processeurs Intel Xeon X5650 de 2,67 GHz de 6 cœurs, soit 12 cœurs hyperthreadés par processeur. Ces deux serveurs possèdent un disque SSD. Cette configuration nous a permis de lancer jusqu'à 25 VM d'un cœur avec 1024 Mo de mémoire vive. Notez que cette plateforme n'est pas plus petite que ce que proposent la plupart des IaaS publics qui imposent une limite au nombre d'instances disponibles par utilisateur. C'est par exemple le cas d'Amazon qui impose par défaut une limite de 20 instances par utilisateur sur EC2. Ces deux serveurs ont un noyau Linux dans sa version 3.2.0–12 avec le module KVM activé.

Nous avons utilisé un troisième serveur — openstack-cc — sur lequel nous avons

installé :

- OpenStack 2012.1.3–dev [50] comme cloud kit. Ce logiciel est activement développé et a une communauté très dynamique pour le support. Le temps de démarrage des VM, optimisé par OpenStack avec un mécanisme de cache, est linéaire en fonction du nombre de VM demandé. Nous avons observé les temps de démarrage induits par la commande $run\_instances(n)$, avec $n \in [10; 60]$ par pas de 10 afin d'obtenir un échantillonnage convenable. Elle est correctement approximée par la régression linéaire $boot\_time = 5,98 \times i + 48,5$ (en secondes) qui prédit quand la i-ème VM aura démarré. Le coefficient de corrélation est de 0,96.

- Slurm 2.5.4 comme *batch scheduler*. Nous avons choisi ce logiciel pour la présence d'une API Perl, nécessaire pour l'utilisation avec Schlouder.

- et le serveur de Schlouder.

Le service central de stockage de l'IaaS est le logiciel Walrus, compatible avec S3 d'Amazon, installé sur le serveur storage. C'est sur ce serveur que se trouvent les images de VM ainsi que les fichiers nécessaires aux applications. L'ensemble de ces quatre serveurs sont situés dans le même LAN.

Nous nommons par la suite cet IaaS privé icps-os-cloud.

BonFIRE

BonFIRE [30] est une plateforme multi-cloud publique orientée recherche. Elle est actuellement présente sur sept sites distribués à travers l'Europe. Nous présentons sur la Figure 3.7 l'architecture de cet IaaS.

Nous avons exécuté nos expériences sur trois de ces sites : de-hlrs, fr-inria, et uk-epcc. Chaque site géographique a une infrastructure physique différente. de-hlrs propose 14 nœuds, chacun avec 16 Go de mémoire vive et deux processeurs quad core Intel Xeon de 2,83 GHz ; 14 nœuds, chacun avec 32 Go de mémoire vive et deux processeurs quad core Intel Xeon de 2,83 GHz ; 6 nœuds, chacun avec 2 Go de mémoire vive et deux processeurs dual core Intel Xeon de 3,2 GHz ; 2 nœuds, chacun avec 196 Go de mémoire vive et quatre processeurs AMD Opteron avec 12 cœurs de 2,6 GHz. fr-inria propose 4 nœuds, chacun avec 64 Go de mémoire vive et deux processeurs Intel Xeon E5-2620 de 2 GHz de 6 cœurs, soit 12 cœurs hyperthreadés par processeur. uk-epcc propose un nœud avec 128 Go de mémoire vive et quatre processeurs AMD Opteron 6176 de 2,3 GHz avec 12 cœurs.

Chaque site est accessible par une API commune basée sur le standard Open Cloud Computing Interface (OCCI) [49]. Le cloud kit utilisé est OpenNebula 3.6 dans une version modifiée pour BonFIRE.

Le nombre maximum de VM qu'un utilisateur peut instancier dépend de l'utilisation actuelle de la plateforme, et d'un quota sur l'espace disque et le CPU. Dans notre cas, avec

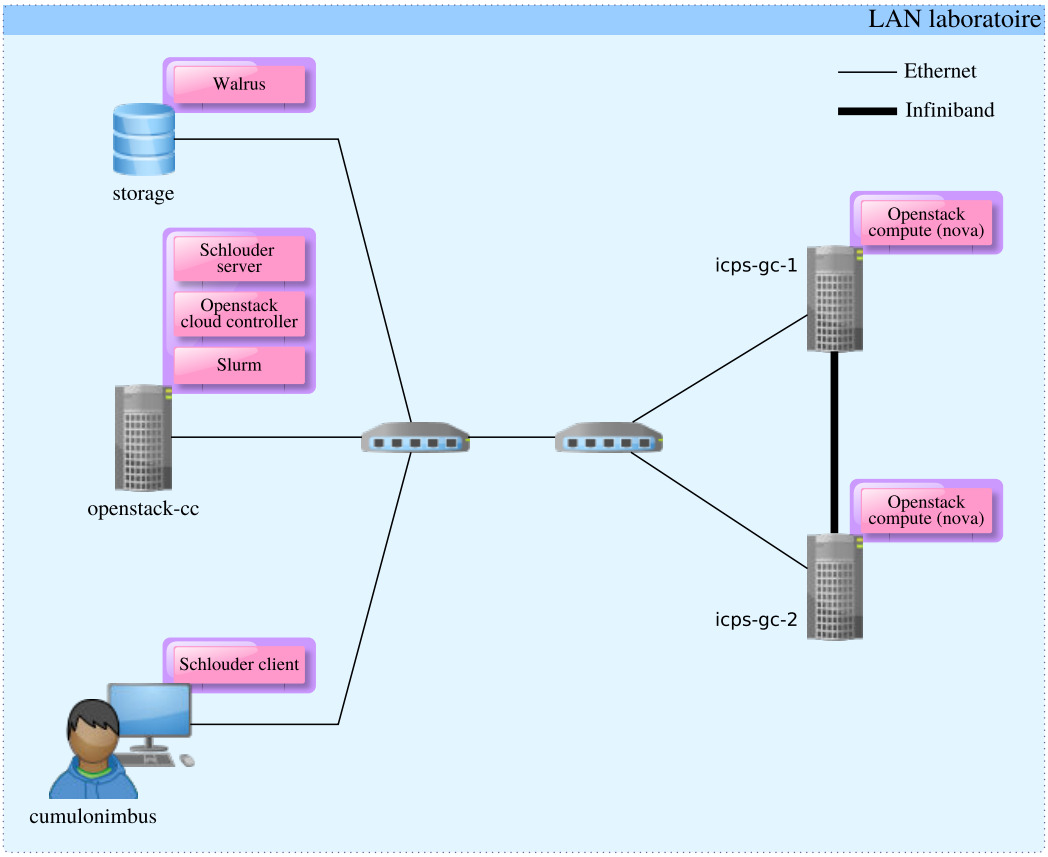


FIGURE 3.6 – Architecture du cloud icps-os-cloud

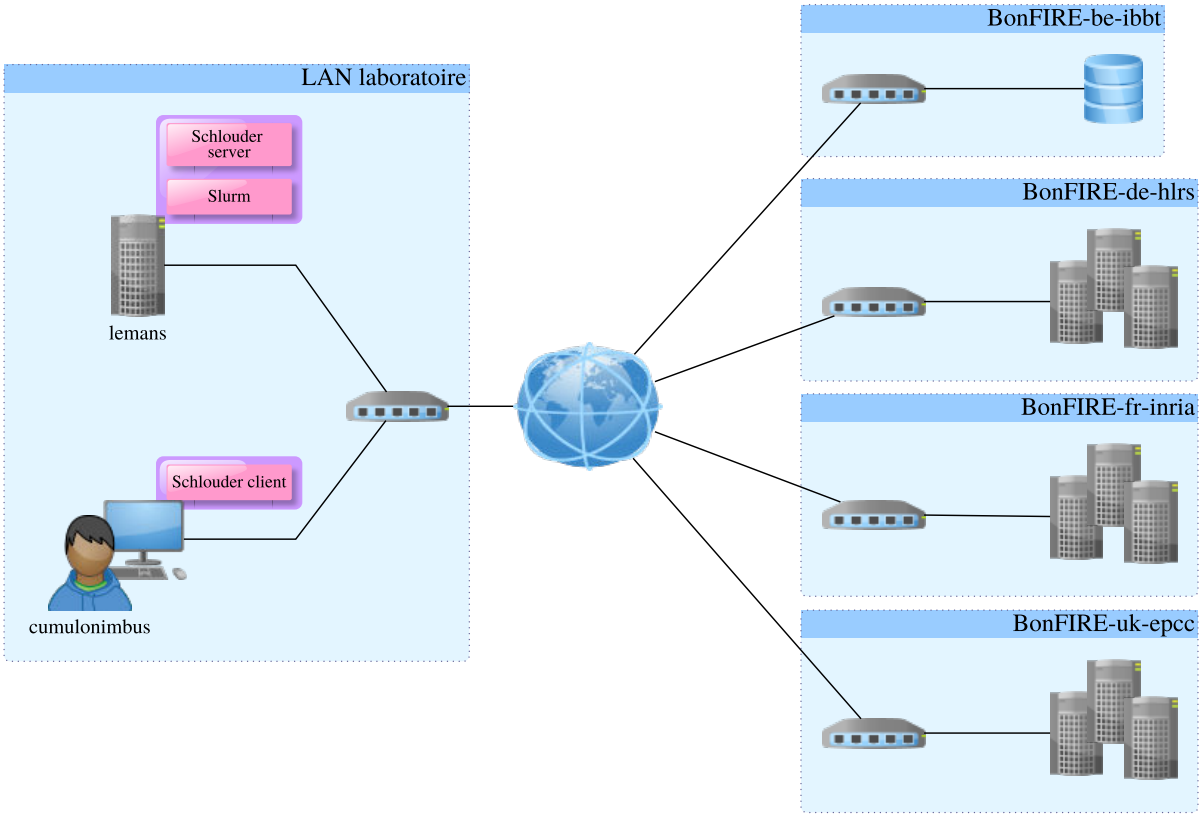


FIGURE 3.7 – Architecture de notre IaaS public

Table 3.3 – Caractéristiques des IaaS

IaaS	# nodes	# cores	CPU power (GHz)			# max VM	hypervisor	storage	boot time (s)		
			min	avg	max				min	avg	max
icps-os-cloud	2	48	2.67	2.67	2.67	25	KVM	Walrus	36	120	255
BonFIRE-de-hlrs	36	344	1.65	2.15	3.40	20	Xen 3.1.2	NFS	34	266	2 097
BonFIRE-fr-inria	4	96	0.93	0.93	0.93	20	Xen 3.2	NFS	123	982	11 084
BonFIRE-uk-epcc	7	176	1.29	1.42	1.54	20	Xen 3.0.3	NFS	130	437	1 047

une image de VM de 10Go déployée sur une VM d'un cœur et de 1024 Mo de mémoire vive, nous avons pu instancier de 20 à 23 VM sur chaque site. Il est important de noter qu'il s'agit d'un nombre de VM du même ordre de grandeur que ce que proposent la plupart des IaaS publics. Nous avons observé les temps de démarrage induits par la commande $run\_instances(n)$, avec $n = 15$. Sur de-hlrs, fr-inria et uk-epcc, les temps de démarrage sont correctement approximés par les régressions linéaires $boot\_time = 3,625 \times i + 41,67$, $boot\_time = 22,13 \times i + 242,65$ et $boot\_time = 42,44 \times i + 86,12$ (en secondes) qui prédit quand la i-ème VM aura démarré. Les coefficients de corrélation sont respectivement de 0,98, 0,76 et 0,99.

L'espace de stockage central fourni par BonFIRE est un partage Network File System (NFS). Le serveur est situé sur le site be-ibbt. Pour accéder à ce service de stockage, une ressource doit être dans le réseau privé de BonFIRE. Le serveur de Schlouder est dans ce réseau par l'utilisation d'un VPN. Dans la suite de ce chapitre, nous nommons ces sites BonFIRE-*site_name*. Comme indiqué sur la figure, il est important de noter que dans cette configuration d'IaaS public, l'ensemble de nos logiciels est installé au sein du laboratoire. Aucune modification au sein de l'architecture de l'IaaS public n'est nécessaire.

Nous récapitulons dans la Table 3.3 les caractéristiques des plateformes d'IaaS privés et publics utilisés au cours de nos expériences. Ils diffèrent en divers aspects : ils ont des tailles et des puissances CPU différentes, et ils utilisent des hyperviseurs différents. Sur BonFIRE, les machines physiques sont hétérogènes, avec des puissances du CPU très différentes. Sur les IaaS privés, le temps de démarrage est plutôt stable et bas alors que BonFIRE montre des temps de démarrage occasionnellement très longs. Ces plateformes représentent deux types très différents d'IaaS : de l'IaaS privé à utilisateur unique à l'IaaS public multi-utilisateur possédant différents sites géographiques.

3.4 Évaluation

Après avoir décrit l'environnement utilisé pour la validation de Schlouder, nous présentons dans cette section l'évaluation de l'efficacité de Schlouder dans l'exécution de calculs scientifique sur l'IaaS. Nous exécutons les deux applications sur différentes plateformes IaaS afin de montrer l'adaptabilité de Schlouder à des situations de calculs variées. Nous présentons dans cette section les métriques utilisées pour notre évaluation puis l'évaluation

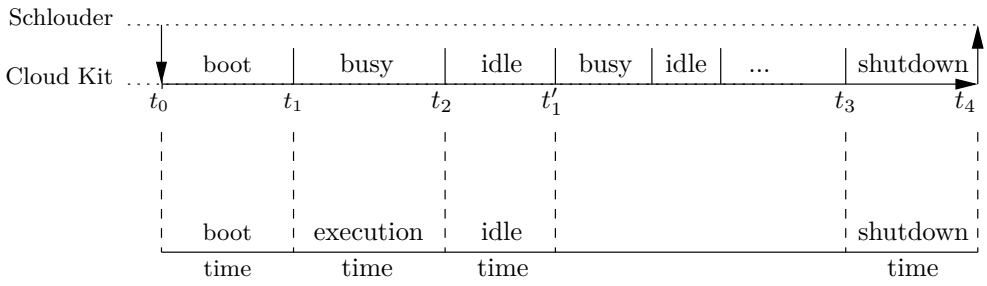


FIGURE 3.8 – Exemple de cycle de vie d'une VM

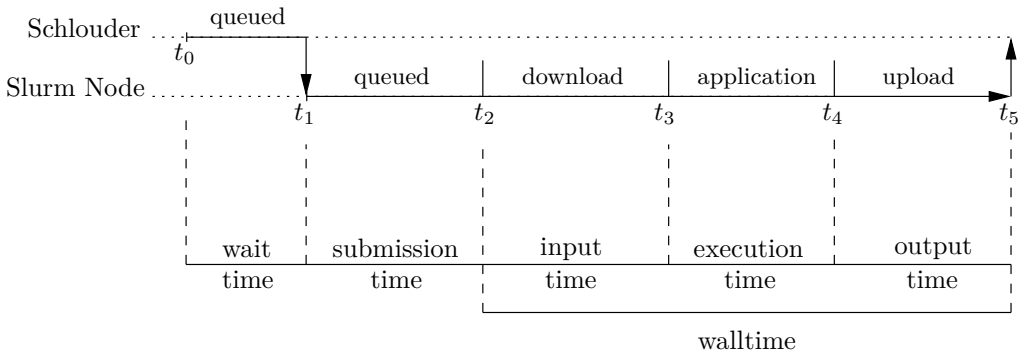


FIGURE 3.9 – Exemple de cycle de vie d'un job

à proprement parler.

3.4.1 Cycles de vie

Nous présentons dans cette section le découpage en différents intervalles de temps du cycle de vie des jobs et des VM. Nous analyserons le comportement de Schlouder dans la prise en charge des workloads en examinant le découpage du temps total d'exécution, aussi bien du point de vue de la VM que du job.

Cycle de vie de la VM

Nous avons identifié quatre intervalles de temps qui caractérisent le cycle de vie d'une VM. Nous montrons sur la Figure 3.8 le cycle de vie d'une VM, du moment où Schlouder demande l'instanciation d'une nouvelle VM (t_0) à l'extinction de cette instance (t_4) lorsque nous avons besoin de réduire la taille de la plateforme. À t_1, la VM a démarré et est prête à exécuter des jobs. Puis, la VM alterne entre exécution et temps d'inactivité. Finalement, la stratégie de *provisioning* réduit la taille de la plateforme et Schlouder demande au cloud kit l'extinction de la VM (t_3). À t_4, la VM est effectivement arrêtée.

Cycle de vie du job

De l'analyse des logs d'exécution, nous avons identifié cinq intervalles de temps qui permettent de caractériser le comportement de l'exécution sur l'IaaS.

La Figure 3.9 montre le cycle de vie d'un job, de sa soumission par l'utilisateur (t_0), à la fin de son exécution (t_5) sur le nœud. À t_1, le *batch scheduler* soumet le job au nœud Slurm sélectionné. Le temps d'attente, $t_1 - t_0$, commence lorsque le job est soumis à Schlouder et finit lorsque Schlouder est capable d'ordonnancer le job (par Slurm) sur la VM sélectionnée. Lorsque le serveur Slurm a réussi à contacter le nœud Slurm, le job démarre son exécution aussi tôt que possible. Cependant, il y a un délai représenté pas le temps de soumission $t_2 - t_1$. Le job démarre réellement son exécution à t_2, avec une première phase qui consiste à télécharger les données en entrée ($t_3 - t_2$). Suit la phase d'exécution à proprement parler de l'application appelée temps d'exécution ($t_4 - t_3$). L'exécution du job se termine par la phase de téléchargement des données en sortie sur le service central de stockage ($t_5 - t_4$). La fin du téléchargement des résultats à t_5, marque la fin du job. Le temps de communication — qui correspond aux deux temps de téléchargement — et le temps d'exécution forment le *walltime* du job.

3.4.2 Comparaison des exécutions sur l'IaaS

Nous regardons maintenant les exécutions d'OMSSA et de Montage sur les IaaS. Pour cela, nous regardons des exécutions sur les trois sites de BonFIRE et sur icps-os-cloud. Nous présentons les résultats dans la Figure 3.10. Nous y montrons les différents intervalles de temps du cycle de vie d'un job et le nombre de BTU (c'est-à-dire le cout) pour l'exécution des six applications[4] : 1x1, 2x2, 3x3, *hrs*, *hrt* et *brt*. Chacune de ces exécutions a été réalisée plusieurs fois pour un total de 218 exécutions. Nous avons découpé le temps d'attente en deux parties : le temps de démarrage des VM et le temps d'attente imposé par l'ordonnancement (dû à la stratégie *AFAP* ou aux dépendances entre les jobs). Le temps de soumission dû à Schlouder a été mesuré, mais est négligeable (en moyenne 0,42% du temps d'exécution). Nous avons donc choisi de ne pas le montrer sur les figures.

Stratégies de *provisioning*

Afin de comparer l'efficacité des stratégies proposées dans Schlouder, nous comparons les exécutions avec les stratégies de *provisioning* *ASAP* et *AFAP*. Sur la Figure 3.10, les graphiques de gauche montrent les résultats pour *ASAP* alors que les graphiques de droite montrent les résultats obtenus avec *AFAP*.

AFAP réduit le nombre de BTU (représenté par les barres de droite dans chaque groupe sur la Figure 3.10), et donc le cout, conformément aux objectifs de cette stratégie : *AFAP* fait économiser de 64% à 94% par rapport à la même exécution avec *ASAP*. Par exemple, le nombre de BTU utilisé en moyenne pour l'exécution de 2x2 est respectivement de 8 et 31 avec *AFAP* et *ASAP*.

4. Notez que suite à une opération de maintenance sur BonFIRE-fr-inria, nous n'avons pas réussi à exécuter 3x3 avec la stratégie *ASAP*

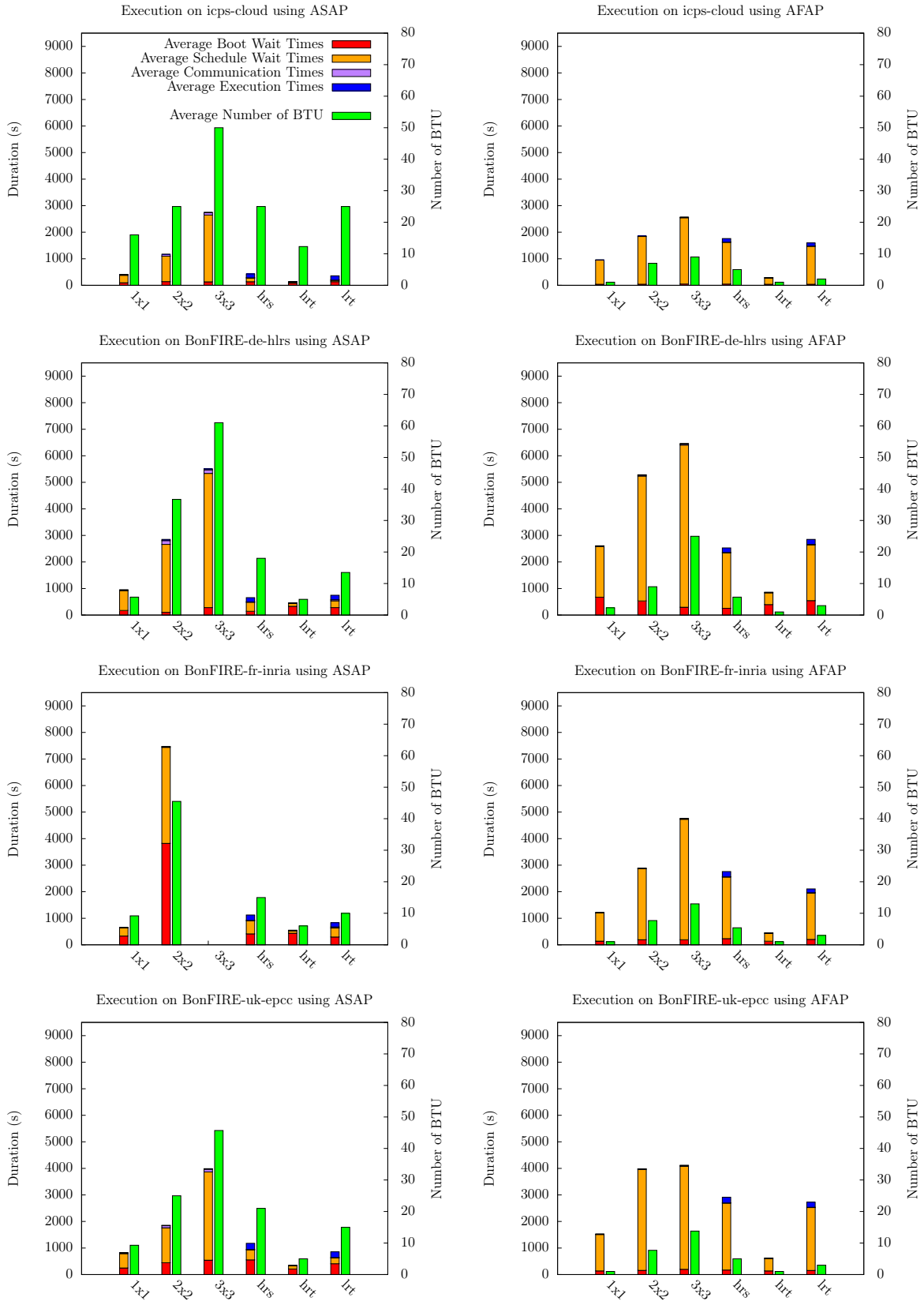


FIGURE 3.10 – Découpage du *walltime* des exécutions d'OMSSA et de Montage sur les quatre plateformes, en utilisant les stratégies de *provisioning AFAP* et *ASAP*

En contrepartie, l'utilisation d'*AFAP* implique des temps d'attente largement supérieurs à ceux qu'implique l'utilisation d'*ASAP*. Par exemple, le temps d'attente moyen pour *hrs* sur BonFIRE-de-hlrs est seulement de 425 s avec *ASAP*, alors qu'il est de 2 427 s avec *AFAP*.

En conclusion, Schlouder réalise le *provisioning* et l'ordonnancement et entraîne le résultat attendu : *ASAP* permet des calculs rapides alors qu'*AFAP* réduit le nombre de BTU, sur toutes les plateformes, pour toutes les applications. Cela montre que Schlouder s'est effectivement adapté à ces deux situations très différentes.

Plateformes

Nous souhaitons maintenant comparer le comportement des exécutions de Schlouder sur les différentes plateformes. Chaque ligne de la Figure 3.10 correspond à l'exécution sur une plateforme différente. Nous constatons dans un premier temps que pour chaque application et chaque stratégie, la répartition du temps passé dans les différentes phases du cycle de vie est très similaire, quelle que soit la plateforme. Par exemple, l'exécution de *hrs* avec la stratégie *AFAP* utilise de 5 BTU à 5,7 BTU en fonction de la plateforme d'exécution.

Le temps d'attente est quasiment constant (environ 90% du temps d'exécution), le temps de communication représente moins de 0,5%, et le temps de calcul est d'environ 7%.

Cela montre que le processus de *provisioning* et d'ordonnancement n'est pas lié à une plateforme donnée. Dans cette expérience, le grand nombre de plateformes (cf. Table 3.3) et les différents types d'applications n'impliquent pas de changement de comportement du système de courtage. C'est un signe encourageant que Schlouder peut être utilisé et produit les résultats attendus dans un grand nombre de situations avec peu ou pas de calibration pour la plateforme cible.

En effet, comme indiqué dans la Section 3.3, nous avons calibré Schlouder en fonction de la plateforme cible de plusieurs manières :

- Afin de prendre sa décision de *provisioning*, Schlouder utilise une prédiction du temps de démarrage des VM sur la plateforme cible. Ce temps peut varier d'une plateforme à une autre en fonction du nombre de machines physiques et de la qualité du réseau.
- Pour certaines stratégies, Schlouder utilise également une prédiction du *walltime* des jobs. Cette prédiction varie en fonction de la puissance de la VM et de la qualité du réseau.

Applications

Nous comparons finalement les exécutions des six applications. Nous observons que le motif en terme de cout, et de temps passé dans les différentes phases est très similaire,

quelles que soient la plateforme et la stratégie. Les couts d'exécution des applications — selon une stratégie — restent ordonnés de la même manière, quelle que soit la plateforme. C'est-à-dire que si le cout d'exécution d'une application est supérieur à celui d'une autre application sur une plateforme avec une stratégie, elle le reste, quelles que soient la plateforme et la stratégie. Pour une stratégie, la proportion de temps d'ordonnancement, de communication et d'exécution est similaire d'une plateforme à une autre.

Ces deux applications représentent des catégories d'applications scientifiques variées, avec des besoins en communication et en puissance de calcul différents (cf. Table 3.2). Il est intéressant de noter que ces expériences montrent une telle constance en fonction des plateformes. Ces expériences montrent que les décisions de courtage de Schlouder ne sont pas changées pour un type d'application. Schlouder montre donc sa capacité à s'adapter à différentes stratégies pour l'exécution d'applications très variées sur des plateformes différentes.

Cas particuliers

Alors que les observations générales formulées précédemment indiquent que Schlouder remplit les objectifs des deux stratégies pour les deux applications sur les IaaS testés, nos expériences présentent des comportements inattendus dans quelques situations particulières. Nous analysons dans cette section les causes de ces particularités.

Divergence des temps de démarrage La première exécution de 2x2 avec *ASAP* (Figure 3.10) implique des temps d'attente considérablement plus long sur BonFIRE-fr-inria (en moyenne 9 400 s par job) comparé à la même exécution sur les autres plateformes (de 1 095 s à 2 660 s). Près de la moitié de ce temps d'attente est en réalité dû aux temps de démarrage. Cette différence importante s'explique par la façon dont BonFIRE-fr-inria instancie de nouvelles VM suite à la requête d'un utilisateur. En comparant le *provisioning* sur BonFIRE-fr-inria et sur BonFIRE-de-hlrs pour une même requête de 22 VM, nous avons découvert que les deux IaaS fournissent au final le même nombre de VM. Cependant, alors qu'elles sont fournies simultanément sur BonFIRE-de-hlrs, elles arrivent en trois lots consécutifs de 8 VM sur BonFIRE-fr-inria : 144 s après le début de l'exécution, puis 3 700 s après le début de l'exécution et enfin 7 286 s après le début de l'exécution. La Figure 3.11 montre la chronologie des exécutions de 2x2 sur ces deux sites en utilisant *ASAP*. Cette figure nous permet de voir ce phénomène.

Cette situation arrive en réalité lorsque l'infrastructure est surchargée. La plupart des plateformes que nous connaissons retournent une erreur dans le cas où elles ne peuvent plus fournir de nouvelle VM, car l'ensemble des machines physiques est occupé. Néanmoins, sur BonFIRE, la requête reste en attente le temps pour l'infrastructure d'être capable de déployer de nouvelles VM. Nous avons gardé ce cas afin de souligner ce problème. Schlouder a néanmoins été corrigé avec un timer afin d'éviter ce type de cas.

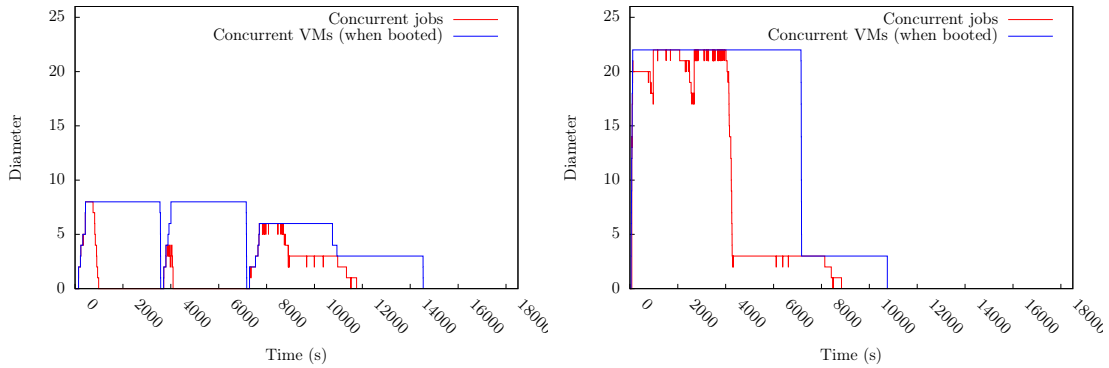


FIGURE 3.11 – exécutions de 2x2 avec *ASAP* sur BonFIRE-fr-inria (gauche) et sur BonFIRE-de-hlrs (droite)

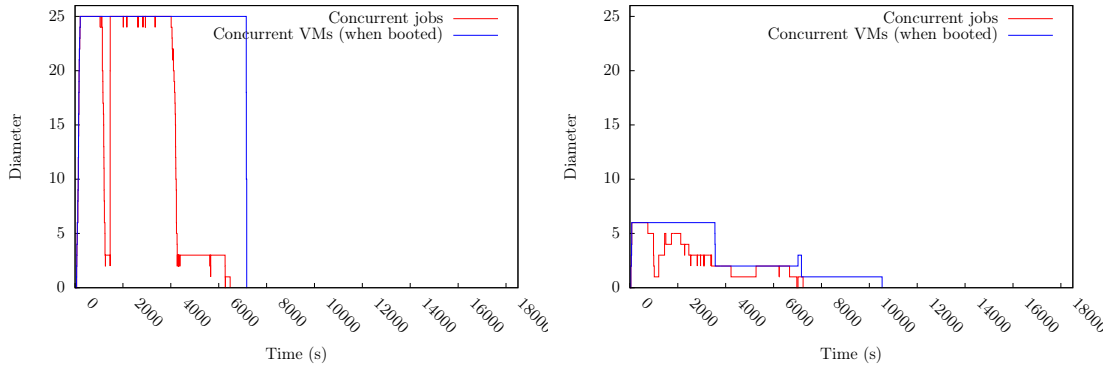


FIGURE 3.12 – exécutions de 3x3 sur icps-os-cloud avec *ASAP* (gauche) et *AFAP* (droite)

***ASAP* n'accélère pas l'exécution de 3x3** Nous montrons sur la Figure 3.12 la chronologie de la première exécution de 3x3 sur icps-os-cloud avec *ASAP* et *AFAP*. Étonnamment, les deux exécutions terminent autour de 6 000 s. La raison pour laquelle *ASAP* échoue à exécuter le workload plus rapidement malgré le plus grand nombre de VM instanciées (25 contre 6 avec *AFAP*) est liée à deux faits :

- Montage est une application intensive en communication (cf. Table 3.2).
- La grande taille des données de 3x3 (69,8 GB). Nous n'observons pas ce phénomène avec 1x1 et 2x2.

En effet, *ASAP* augmente le parallélisme des workloads soumis et augmente ainsi la concurrence entre les communications. La contention qui en résulte rend finalement le temps total d'exécution du workload avec *ASAP* aussi long qu'avec *AFAP*. En conclusion, la robustesse apparente de la stratégie de *provisioning* *ASAP* est remise en cause dans le cas d'application intensive en communication. Les utilisateurs doivent choisir dans ces cas cette stratégie de *provisioning* avec prudence, ou utiliser la simulation au préalable.

Mauvaises prédictions des *walltimes* Les stratégies prennent la décision de *provisioning* en se basant sur le *walltime* du job tel qu'annoncé par l'utilisateur en préambule

de la description du job. Dans le cas de nos expériences, ces données ont été obtenues par calibration des jobs sur la plateforme icps-os-cloud. Malgré les différences de puissance de calcul avec les autres plateformes, nous avons décidé de garder la même prédiction de *walltime* afin d'évaluer la robustesse du *provisioning* en fonction de la précision de cette information.

Nous observons que la granularité grossière de la BTU permet une tolérance aux imprécisions dans la prédiction du *walltime*. Cependant, nous montrons sur la Figure 3.13 un cas problématique. L'exécution de *brt* sur icps-os-cloud, qui est la plateforme de calibration, nécessite deux VM pour la stratégie *AFAP*. L'exécution des jobs termine exactement avant la fin de la BTU. Lorsque nous exécutons ce même workload sur BonFIRE-de-hlrs — où la puissance CPU est moins élevée — la stratégie de *provisioning* prend sa décision en se basant sur un *walltime* sous-estimé (3 435 s au lieu de 4 810 s). La capacité d'une BTU (3, 600 s) n'est alors plus suffisante ce qui entraîne le déclenchement d'une BTU supplémentaire.

Une prédiction précise du *walltime* aurait permit à *AFAP* d'instancier une VM supplémentaire à la place d'une BTU supplémentaire, réduisant ainsi le temps de complétion tout en maintenant le même nombre de BTU total.

La prédiction des *walltimes* est un problème classique dans le champ de l'ordonnancement de jobs, et certains algorithmes sont plus sensibles que d'autres à cette incertitude dans la durée des tâches [8]. Pour *AFAP*— et plus généralement pour les stratégies de bin packing — les utilisateurs devraient fournir une prédiction précise des durées afin de s'assurer que la stratégie s'exécute efficacement. Dans le cas contraire, les utilisateurs devraient se tourner vers des algorithmes moins sensibles tels qu'*ASAP*.

Cependant, de nombreuses techniques permettent d'améliorer la précision de la prédiction. Par exemple, dans notre cas, nous pouvons effectuer la calibration sur toutes les plateformes, apprendre des exécutions précédentes, ou déduire la prédiction en fonction des performances de la plateforme telles que rapportées par le job de monitoring de Schlouder. Il est également possible de réduire la sensibilité d'*AFAP* à la prédiction des *walltimes* en utilisant conjointement un algorithme d'ordonnancement tel que *Min-min*.

3.4.3 Conclusions

Nous avons présenté dans ce chapitre un système de courtage appelé Schlouder qui a été conçu dans le but de faciliter l'exécution de calculs, sous la forme de *bag-of-tasks* ou de *workflows*, dans l'IaaS. Son utilisation a été illustrée par l'exécution de six cas applicatifs représentatifs des applications scientifiques.

Nous avons montré que Schlouder, en utilisant deux des stratégies de *provisioning* que nous fournissons, peut exécuter des calculs sur une plateforme IaaS. Nous avons démontré que Schlouder permet d'adapter la plateforme aux besoins applicatifs et aux préférences

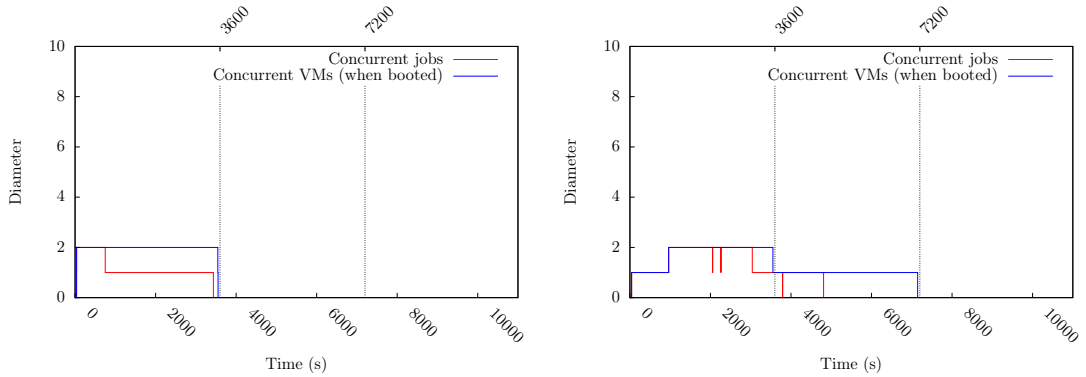


FIGURE 3.13 – exécutions de *brt* avec *AFAP* sur icps-os-cloud (gauche) et sur BonFIRE-de-hlrs (droite)

utilisateurs, quelles que soient la plateforme, l'application et la stratégie de *provisioning*.

Nous avons de plus identifié et analysé plusieurs cas particuliers pour lesquels l'exécution de la stratégie ne s'est pas déroulée comme prévu. Les principales raisons sont : l'instabilité de la plateforme, les problèmes de réactivité, ainsi que l'utilisation d'*ASAP* dans le cas d'une application intensive en communication. Ces analyses ont permis d'une part de faire évoluer Schlouder pour les prendre en compte, d'autre part de produire des recommandations en direction des utilisateurs

3.5 De la grille au cloud : une expérience d'une application de bio-informatique

Après avoir montré que Schlouder est un système de courtage fonctionnel pour l'exécution de calculs dans l'IaaS, nous souhaitons maintenant comparer le comportement de Schlouder dans la prise en charge d'un calcul avec une application existante qui s'exécute sur une grille de calcul.

Les grilles sont un moyen très efficace pour l'exécution de calculs scientifiques constitués de jobs indépendants. Elles offrent un moyen de mutualiser divers clusters dans le but de fournir une interface commune. L'European Grid Infrastructure (EGI) est un exemple d'une telle grille. Elle permet à des milliers de scientifiques d'accéder à une infrastructure de calcul en production.

Dans les sections suivantes, nous nous concentrerons sur l'exécution de l'application de protéomique Open Mass Spectrometry Search Algorithm (OMSSA) [22]. Les protéomistes de l'Institut Pluridisciplinaire Hubert Curien (IPHC) à Strasbourg ont développé un outil qui a pour but de faciliter l'exécution d'OMSSA sur la grille EGI. Il prend la forme d'une interface web appelée Mass Spectrometry Data Analysis (MSDA) [9]. Pour ses utilisateurs, un tel outil automatise l'exécution en parallèle de plusieurs instances d'OMSSA distribuées sur EGI. Cela permet de réaliser des analyses à grande échelle pour les projets

de protéomique.

Nous décrirons comment nous avons porté cet environnement mature, basé sur la grille, sur notre plateforme IaaS privée expérimentale. En analysant en détail les logs d'exécution de quatre cas d'utilisation typiques sélectionnés par les protéomistes sur MSDA, nous comparerons les comportements de ces deux systèmes.

3.5.1 MSDA : exécution d'une application scientifique sur la grille

Nous allons décrire dans cette section l'environnement actuellement en production pour l'exécution d'OMSSA sur la grille EGI ainsi que son comportement lors d'une exécution classique.

Le portail web Mass Spectrometry Data Analysis (MSDA) est un portail développé par les chercheurs de l'IPHC. Son rôle est de prendre en charge les différentes étapes de l'exécution d'OMSSA : de la gestion des données à la soumission pour exécution sur une grille. MSDA copie dans un premier temps les bases de données nécessaires sur le serveur de stockage de la grille, puis il regroupe les spectres dans des fichiers pour soumission à différents nœuds de calcul. Pour chacun de ces fichiers, un job exécutant OMSSA est créé pour effectuer l'identification. Le grain de la parallélisation (c'est-à-dire le nombre de spectres distribués sur chaque nœud) est calculé par MSDA en fonction de la résolution demandée et du nombre de nœuds disponibles. Les protéomistes ont déterminé empiriquement qu'une bonne granularité est 1 250 spectres par job. Cette granularité apporte globalement le plus court *makespan* sur la grille EGI.

Une fois les jobs préparés, MSDA les soumet à un méta-ordonnanceur hébergé localement. JJS [7] a été choisi pour MSDA. La principale raison de ce choix est qu'il propose de bonnes performances dans le processus de soumission : après une première phase d'entrainement, tous les sites de la grille sont classés en fonction de la rapidité de leur réponse ; seuls les plus rapides sont utilisés. JJS lisse les soumissions des jobs en procédant par cycle. À chaque cycle, JJS soumet un certain nombre de jobs au gestionnaire des ressources locales (LRMS pour *Local Resource Management System*) du site distant. Il s'agit par exemple des *batch scheduler* tels que Slurm, SGE ou PBS. JJS monitore ensuite l'évolution des jobs soumis. Les protéomistes ont empiriquement observé que la soumission de 50 jobs par cycle donne de bons résultats sur la grille EGI.

3.5.2 Les plateformes d'exécution sur la grille et un cloud IaaS

Dans cette section, nous présentons tout d'abord la grille sur laquelle s'appuient les protéomistes pour exécuter les calculs soumis par MSDA. Puis nous présentons la plateforme d'IaaS que nous avons utilisée pour exécuter OMSSA par Schlouder.

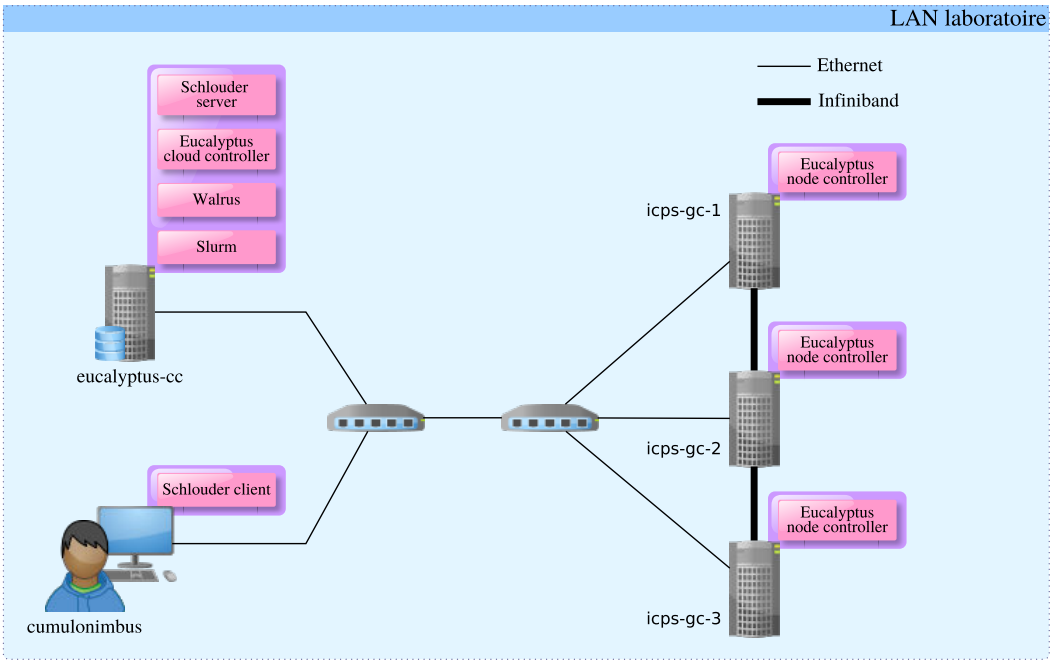


FIGURE 3.14 – Architecture du cloud icps-euca-cloud

La grille EGI

La grille EGI est le lien entre des clusters répartis à travers toute l'Europe sur 34 sites distincts. Elle a pour objectif de fournir des ressources informatiques dans de nombreuses disciplines scientifiques. Les chercheurs sont répartis dans des groupes par intérêt scientifique commun en fonction de leurs besoins. Ces groupes sont appelés des organisations virtuelles. EGI compte aujourd'hui plus de 200 organisations virtuelles. MSDA fait partie de l'organisation virtuelle biomed.

Il est difficile de trouver les caractéristiques des machines utilisées lors d'une exécution sur la grille puisque chaque tâche peut utiliser des ressources différentes. Dans nos expériences, nous avons cependant remarqué que la plupart des machines possèdent un processeur Intel et AMD récent tels que les Opteron K10 d'AMD et les Xeon Nehalem d'Intel.

La base de données utilisée pour l'identification des peptides et les exécutables d'OMSSA sont stockés sur une ressource dédiée au stockage de la grille.

Une fois qu'un site EGI a été choisi par le méta-ordonnanceur JJS, la base de données et les exécutables sont transférés sur les nœuds du site.

Eucalyptus

Nous présentons sur la Figure 3.14 l'architecture de notre IaaS privé. Le cloud IaaS privé présenté ici diffère légèrement de celui présenté dans la Section 3.3.3. La première différence notable est le changement de cloud kit. Nous avons ici utilisé Eucalyptus 2.0.3. Nous

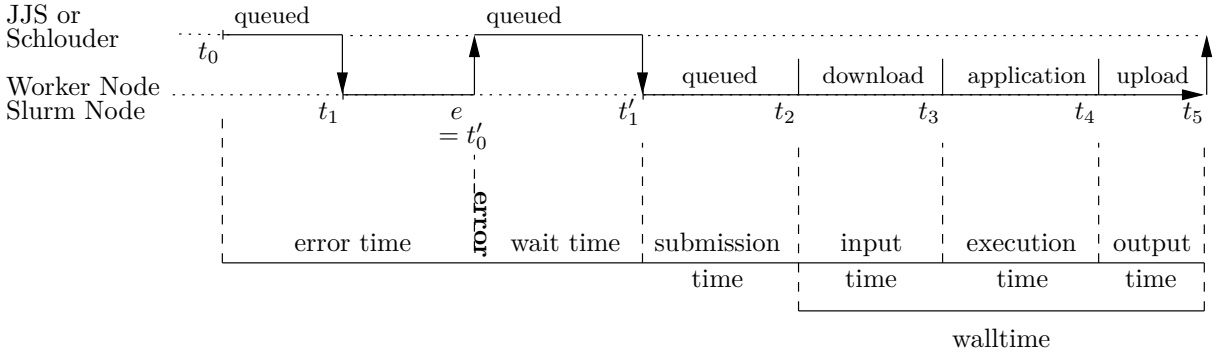


FIGURE 3.15 – Exemple de cycle de vie d'un job sur un cloud IaaS et sur la grille

avons souhaité changer de cloud kit afin de tester la flexibilité de Schlouder. De plus, un troisième nœud, identique aux deux premiers a été ajouté. Avec cette configuration, nous pouvons démarrer jusqu'à 72 VM.

Nous pouvons également noter que le service de stockage du cloud, Walrus, est installé sur la même machine que le cloud kit (eucalyptus-cc).

Nous nommons par la suite cet IaaS privé icps-euca-cloud.

Le temps de démarrage des VM est optimisé par Eucalyptus avec un mécanisme de cache, tout comme le fait OpenStack. Ce temps de démarrage est linéaire en fonction du nombre de VM demandé. Nous avons observé que la commande $run\_instances(n)$, avec $n \in [10; 60]$ par pas de 10, est correctement approximée par la régression linéaire $boot\_time = 5,55 \times i + 50,96$ (en secondes) qui prédit quand la i-ème VM aura démarré. Le coefficient de corrélation est de 0,993.

Notez que l'expérience présentée dans cette section a été réalisée avant celle présentée dans la Section 3.3.

3.5.3 Métriques pour l'évaluation

Nous comparons dans un premier temps une exécution sur l'IaaS avec la même exécution dans l'environnement original sur la grille EGI. La discussion sera appuyée par les observations faites sur MSDA durant deux mois. Pour réaliser une comparaison correcte, nous avons rejoué, sur icps-euca-cloud, l'exécution sur MSDA dont nous avons obtenu les logs.

Cycle de vie d'un job

Nous pouvons associer les intervalles de temps du cycle de vie d'un job exécuté sur un IaaS et présentés dans la Section 3.4.1 avec des intervalles similaires pour une exécution sur la grille. Nous présentons sur la Figure 3.15 le cycle de vie d'un job sur la grille et sur un IaaS, de son admission par le meta-scheduler (t_0) à la fin de son exécution (t_5).

À t_1, le meta-scheduler soumet le job au LRMS du site choisi. Comme des erreurs peuvent arriver sur la grille, nous en montrons un exemple sur la figure. L'erreur est

détectée à la date e. Le temps d'erreur est défini par $e - t_0$. À la détection de l'erreur, le job est immédiatement remis dans la file d'attente du meta-scheduler durant un temps $t'_1 - t'_0$ appelé temps d'attente. Le job est alors soumis à un site différent à t'_1. Une fois sur le site d'exécution, le job est mis dans la file d'attente du LRMS durant $t2 - t'1$, appelé temps de soumission. Le job démarre effectivement à t_2, avec une première phase de téléchargement de la base de données et des exécutables de l'application. La seconde phase est l'exécution de l'application et la dernière phase le téléchargement des résultats sur le service de stockage de la grille. Nous appelons la somme des deux temps de téléchargement le temps de communication.

3.5.4 Comparaison des exécutions

Nous allons dans un premier temps comparer l'exécution sur la grille et sur notre IaaS privé en utilisant la stratégie *ASAP*. Dans les deux cas, l'objectif de l'utilisateur est d'exécuter les jobs le plus rapidement possible, sans considération du nombre de ressources à utiliser.

Par rapport à l'expérience présentée dans la Section 3.3, les protéomistes ont exécuté un quatrième cas d'utilisation d'OMSSA. Il s'agit de la recherche de spectrométrie de masse à basse résolution et semi-spécificité enzymatique complète (*brs*) que nous avons donc également exécutée sur notre IaaS privé.

Les intervalles de temps observés sur la grille et sur l'IaaS icps-euca-cloud avec la stratégie *ASAP* ont été rassemblés dans la Table 3.4. Il s'agit d'une moyenne de 4 à 10 exécutions des mêmes expériences pour chacun des types de recherches (*hrs*, *hrt*, *brs*, *brt*).

Une autre façon d'analyser le comportement de l'exécution est de regarder l'évolution du nombre de jobs et de VM concurrentes pour chaque exécution. Nous appelons ce nombre de jobs ou de VM le *diamètre*. La Figure 3.16 présente la façon dont le diamètre de la plateforme évolue au cours du temps. Nous utiliserons cette figure dans certaines observations.

Temps d'erreur

Nous avons constaté que les exécutions sur EGI doivent s'accommoder d'un nombre très élevé d'erreurs en comparaison avec l'infrastructure IaaS, sur laquelle nous n'en avons constaté aucune. Ce fait est connu et bien mis en valeur par Magoulès [37]. Le temps passé en erreur sur la grille est en moyenne de 30% du temps total sur toutes les exécutions. Nous avons également noté que certains jobs ont subi de multiples erreurs. C'est pourquoi, *hrs* a plus d'erreurs que de nombre de jobs. Le temps perdu à cause des erreurs n'est cependant pas proportionnel au nombre d'erreurs puisque les jobs sont soumis en parallèle. Les raisons de ces erreurs sont multiples : exécution annulée à cause d'une durée trop longue, panne du middleware, connexion refusée, panne matérielle, etc. Quelle que soit l'origine de ces erreurs, la nature hautement distribuée et multi administrée de la grille

TABLE 3.4 – Découpage du temps total sur la grille et sur l'IaaS

	hrs (65 jobs)		*hrt* (65 jobs)	
	Grid	*ASAP*	Grid	*ASAP*
(1) Error time (#Jobs in error)	34.36 (127.6)	0	240.69 (27.25)	0
(2) Wait time	158.53	259.28	159.29	103.20
(3) Submission time	49.14	2.18	110.40	1.19
(4) Communication time	25.69	0.39	13.48	0.27
(5) Execution time	234.15	161.18	33.50	7.22
Total time	1026.80	486.50	768.50	142.50
	brs (223 jobs)		*brt* (223 jobs)	
	Grid	*ASAP*	Grid	*ASAP*
(1) Error time (#Jobs in error)	17.19 (195.88)	0	87.11 (55)	0
(2) Wait time	239.27	1191.40	67.22	209.71
(3) Submission time	72.06	1.36	157.43	1.20
(4) Communication time	26.98	0.53	2.06	0.35
(5) Execution time	816.80	855.67	118.74	24.32
Total time	2438.50	3373.33	887.00	419.25

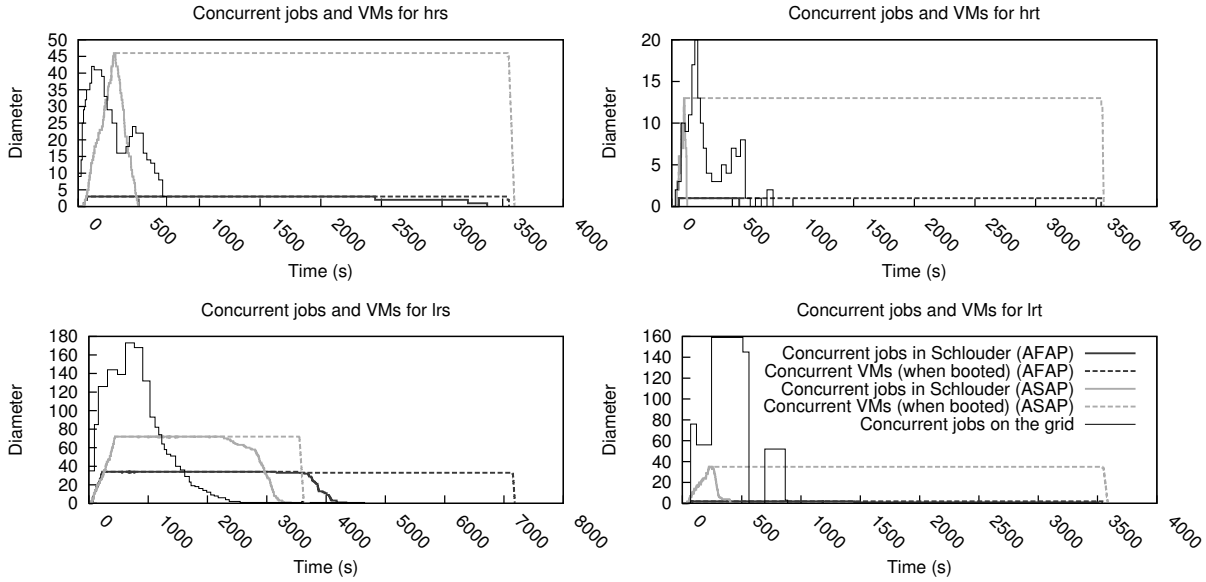


FIGURE 3.16 – Diamètre des jobs et des VM sur l'IaaS et sur la grille

rend l'identification et la correction des erreurs difficiles. Dans la plupart des cas, la détection des erreurs est basée sur l'expiration d'un time out, ce qui explique l'importance du temps d'erreur parmi l'ensemble des intervalles de temps.

Les erreurs peuvent également être observées sur la Figure 3.16 : pour *hrs* et *hrt*, avec 65 jobs, après la première vague d'exécution des jobs, nous observons une seconde vague après environ 300 s qui correspond aux jobs resoumis suite à la détection d'une erreur due à l'expiration du time out.

Temps d'attente

Le temps d'attente nous indique combien de temps est nécessaire pour qu'un job soit accepté sur un site de calcul. Bien que les temps d'attente soient du même ordre de grandeur dans la Table 3.4 pour les deux environnements, la nature de ces temps diffère.

Sur la grille, le temps d'attente est le résultat du fonctionnement de JJS qui ordonnance des groupes de jobs par cycle à intervalle de temps régulier.

Sur Schlouder, deux facteurs entrent dans le temps d'attente : le temps de démarrage des VM, et le nombre limité de VM qui peuvent être démarrées. Le premier facteur est contrôlé par Schlouder qui estime s'il est préférable de démarrer une nouvelle VM ou d'attendre qu'une VM devienne disponible afin d'exécuter le job le plus tôt possible. Par exemple, les cas *brt* et *hrt* impliquent des jobs courts qui peuvent être exécutés séquentiellement sur une seule VM. Ainsi, le nombre de VM démarré est limité par rapport aux 223 jobs à exécuter. Le deuxième facteur est propre à notre plateforme IaaS privée. Une fois que 72 VM sont démarrées, les jobs doivent attendre qu'une de celles-ci soit disponible (nous n'exécutons qu'un job à la fois sur chaque VM). Au contraire, les ressources sur EGI ne s'épuisent jamais.

Nous voyons que ce temps d'attente ne peut pas être évité sur la grille, mais que sur l'IaaS, nous pourrions disposer d'une plateforme déjà prête à exécuter des jobs. Nous aurions par exemple pu proposer une stratégie de pré-*provisioning* dont l'objectif est de démarrer un certain nombre de VM avant le début de l'expérience. Elle aurait permis de réduire le temps d'attente sur l'IaaS.

Temps de soumission

Le temps de soumission peut être vu comme une mesure de la réactivité de l'environnement d'exécution. Il s'agit de l'intervalle de temps entre l'admission du job par le LRMS et le début de son exécution effective sur le nœud de calcul.

Sur la grille EGI, le LRMS est un service d'accès tel que LCG-CE ou CREAM qui se connecte à un ordonnanceur spécifique à un site. Nous avons observé l'utilisation de PBS, LSF, SGE et BQS au cours de nos expériences.

Avec Schlouder, le temps de soumission correspond à l'intervalle de temps durant lequel un job reste dans la file d'attente de Slurm.

Il n'est pas surprenant de voir que ce temps est plus important sur la grille étant donné le niveau élevé de hiérarchie qui y est présent pour pouvoir accéder aux nœuds de calcul. D'un côté, le middleware de la grille prend en charge des systèmes complexes de gestion des ressources tels que la sécurité, l'ordonnancement et le monitoring. D'un autre côté, utiliser le cloud IaaS permet de construire un environnement d'exécution plus léger embarqué dans les VM, qui peut être ajusté aux besoins de l'utilisateur.

Temps de communication

Les temps de communication sont également plus importants sur la grille que sur l'IaaS, même si dans cette expérience la quantité de données à télécharger est faible. Cela reflète bien une des différences essentielles entre ces deux environnements. La nature intrinsèquement distribuée de la grille implique des communications sur de longues distances afin que les données soient téléchargées sur chaque site distant. Au contraire, les IaaS proposent des solutions de stockage centralisées, accessibles au travers d'un système de fichiers distribué local. Ainsi, si un seul site est utilisé, les communications d'un cloud IaaS se font exclusivement sur un LAN, le rendant plus adapté pour des applications data-intensive.

Temps d'exécution

Les différences entre les temps d'exécution sur EGI et sur le cloud IaaS ne sont pas directement exploitables du fait de la différence de matériel exploité sur ces deux plateformes. De plus, les logs de l'exécution sur la grille ne contiennent que les dates de début et de fin de l'exécution telles que rapportées par le LRMS. Il est clair que ces mesures tiennent également compte d'un temps de démarrage des jobs puisque le ratio des temps d'exécution avec l'IaaS sont très grand (de l'ordre de 5 fois) lorsque l'expérience concerne de petits jobs (*hrt* et *brt*) et plus faible (de l'ordre de 1,5 fois) lorsque l'expérience concerne de grands jobs (*hrs* et *brs*).

3.5.5 Conclusions

Après avoir montré que Schlouder est un système de courtage fonctionnel dont le but est de faciliter l'exécution de calcul sur une infrastructure IaaS, son utilisation a été évaluée au travers du portage d'un outil réel d'aide à l'exécution d'une application de protéomique sur une grille. Nous avons comparé Schlouder avec l'exécution de cette application de protéomique dans son environnement d'exploitation initial en production sur la grille EGI.

Schlouder permet de bénéficier des avantages de l'IaaS : l'élasticité, le faible cout, et la fiabilité. D'un point de vue technique, l'IaaS n'a pas la complexité des middleware des

grilles, ce qui permet d'éviter un certain nombre de latences et de surcouts.

En outre, note étude a montré que Schlouder permet de réduire drastiquement le temps nécessaire à la mise en place d'une expérience. En effet, mettre en place cette expérience sur la grille a nécessité de développer un portail dédié (MSDA) capable de monitorer l'exécution, de mettre en place un méta-ordonnanceur dédié (JJS) afin d'ordonner les ressources disponibles, et de tester différentes configurations. D'un autre côté, mettre en place cette expérience sur Schlouder a nécessité de créer une image de VM contenant les fichiers d'OMSSA, et d'adapter le script de soumission de la grille afin d'utiliser le stockage de l'IaaS. Cela a été fait en l'espace d'une heure, sans calibration ou connaissance particulière de la plateforme d'exécution. L'application est, de plus, totalement portable sur un autre IaaS et est adaptable aux préférences de l'utilisateur.

Nous avons cependant également montré que les performances des applications peuvent être impactées par l'utilisation intensive des entrées-sorties. Nous mettons ici en lumière un point important dans le choix de la stratégie de *provisioning*. L'utilisation d'*ASAP* dans un contexte d'application intensive en communication risque de surcharger le réseau et de ralentir, au final, l'exécution. De plus, la location sous forme de BTU d'une heure implique d'être précautionneux dans la gestion des ressources au risque de payer un surcout inutile. Il est donc nécessaire d'avoir des informations précises sur les temps d'exécution des jobs. Malgré les contraintes et erreurs qu'implique l'utilisation d'une grille de calcul, l'exécution de l'application s'est achevée dans des délais proches de l'exécution sur notre IaaS privé. Schlouder s'est montré très simple pour l'exécution d'une application scientifique sur l'IaaS.

Les problèmes à prendre en charge dans la gestion des ressources sont très différents entre la grille et l'IaaS. Les efforts principaux sur la grille visent à organiser le workload et trouver un bon pattern de soumission afin d'exploiter autant de ressources que possible. En effet, les protéomistes ont cherché le nombre optimal de spectres par job ainsi que le nombre optimal de jobs par cycle à soumettre par JJS. Sur l'IaaS, l'effort principal consiste à définir le nombre de ressources qui doivent être provisionnées en fonction des besoins de l'utilisateur. Cet effort est pris en charge par Schlouder. Ainsi, une application sur la grille peut être portée sur l'IaaS sans effort supplémentaire. D'adapter l'application sur la plateforme, à adapter la plateforme à l'application : l'IaaS est un changement majeur dans les problématiques du calcul scientifique.

Dans le cas où une organisation possède une application à distribuer sur une infrastructure, elle doit se poser plusieurs questions. Tout d'abord, si l'application est déjà en production et son exécution distribuée sur une grille, tant que la grille reste stable et n'implique pas un effort d'adaptation constant, l'organisation ne semble pas avoir intérêt à migrer sur une infrastructure IaaS. Dans le cas où l'organisation n'a pas encore distribué l'exécution de son application, l'utilisation d'une infrastructure IaaS est une option envisageable : il s'agit d'une question d'équilibre entre les besoins de calcul et les capacités

budgétaires d'investissement et de fonctionnement. Cependant, si l'organisation prévoit une utilisation intensive de la plateforme il peut être intéressant d'investir dans un cluster à intégrer dans une grille. L'utilisation d'un cloud IaaS est particulièrement indiquée dans le cadre d'une organisation ayant une utilisation sporadique de la plateforme ou d'une organisation n'ayant pas les moyens d'investir, puis de maintenir un cluster local.

Chapitre 4

Simuler l'exécution d'un workload

Contributions principales

- Développement d'un simulateur de cloud IaaS.
- Intégration du simulateur au système de courtage afin d'être utilisé comme outil d'aide à la décision de *provisioning*.

4.1 Introduction

Nous avons montré dans les précédents chapitres que nous disposons d'un outil permettant d'exécuter des calculs sous forme de *bag-of-tasks* et de *workflows* sur des infrastructures IaaS, en utilisant une stratégie de *provisioning* défini par l'utilisateur. Nous n'avons cependant pas encore résolu le problème du choix de la stratégie de *provisioning*.

Comme nous l'avons montré dans le Chapitre 1, il est difficile de prédire analytiquement les conséquences du choix d'une stratégie de *provisioning* dans un contexte d'ordonnancement *online* [23] à cause des effets de seuil induit par l'utilisation des BTU [43].

Si de nombreux travaux connexes conçoivent de nouvelles stratégies de *provisioning*, peu de travaux proposent une solution pour choisir parmi un ensemble de stratégies. Deng et al. [15] propose une solution de type *portfolio scheduling*. Les auteurs construisent un portfolio contenant des stratégies de *provisioning*, dont certaines issues de nos travaux. Ils proposent également une métrique prenant en compte le nombre de BTU, le *makespan*, et le *slowdown*. Le but étant de trouver la stratégie réalisant le meilleur compromis entre ces trois métriques pour un workload donné. Pour cela, ils utilisent le simulateur DGSim [25] pour simuler l'exécution du workload sur l'ensemble des stratégies du portfolio.

Nous pensons que la meilleure approche est celle de proposer un simulateur au sein duquel l'utilisateur peut implémenter ces propres stratégies de *provisioning*. Nous estimons que seul l'utilisateur est à même de choisir la stratégie qui l'intéresse, afin d'optimiser le critère qu'il souhaite. Aucune métrique ne peut faire ce choix à la place de l'utilisateur. Notre objectif étant de lui fournir le cout, le temps total ainsi qu'une vision précise de l'exécution de son workload pour toutes les stratégies de *provisioning*.

La simulation d'infrastructures de calculs distribués est un défi stimulant pour la communauté scientifique. De nombreux travaux proposent des solutions pour la simulation des

grilles telles que GridSim [47] ou OptorSim [5]. Cependant, la modélisation d'un système hétérogène comme la grille avec un modèle réseau valide et efficace est particulièrement difficile et ces travaux ne répondent que partiellement aux contraintes imposées par un tel système[57, 71].

Simuler un IaaS est simplifié par le côté homogène des ressources proposées par les fournisseurs de cloud IaaS, ainsi que par la possibilité offerte par la virtualisation de posséder une plateforme mono utilisateur. Ainsi, nous pouvons proposer aux utilisateurs de Schlouder un outil fiable et efficace pour la sélection d'une stratégie de *provisioning* dans un contexte *online*. Il existe quelques simulateurs d'IaaS tels que iCanCloud [48], CloudSim [6], DCSim [67] et GroudSim [52]. Aucun de ces simulateurs ne présente de travaux comparant les résultats de l'ordonnancement et du *provisioning* effectué par le simulateur avec ce résultat dans la réalité. Nous avons donc choisi de développer notre propre simulateur.

Nous présentons dans la Section 4.2 l'architecture du simulateur présent au sein du Simulation Engine. Puis nous évaluons sa précision dans la Section 4.3 en comparant les résultats fournis par le simulateur avec des exécutions réelles avec Schlouder.

4.2 L'architecture du Simulation Engine

Le Simulation Engine est en réalité un projet séparé de Schlouder appelé SimSchlouder. Nous avons développé une interface de type Adaptateur[1] afin de pouvoir l'appeler depuis Schlouder. SimSchlouder a été développé avec un objectif double :

- fournir à l'utilisateur un outil d'aide à la décision entre toutes les stratégies pour un workload donné ;
- fournir à l'utilisateur l'ordonnancement précis des jobs sur les VM.

La Figure 4.1 montre comment les différentes briques de ce simulateur s'articulent entre elles : SimSchlouder est basé sur SCHIaaS, lui-même basé sur SimGrid [11].

4.2.1 SimGrid

SimGrid est un instrument scientifique pour étudier le comportement des systèmes distribués à grande échelle tels que les grilles, les clouds ou les systèmes pair-à-pair. Il peut être utilisé pour évaluer des heuristiques, des prototypes d'applications, ou même tester des applications MPI réelles.

SimGrid a été conçu comme un outil de mesure scientifique. Ainsi, la validité de ses modèles a été largement étudiée et validée [70]. Nous nous assurons ainsi de baser notre simulateur sur un moteur de simulation efficace, précis et validé.

1. `https://en.wikipedia.org/wiki/Adapter_pattern`

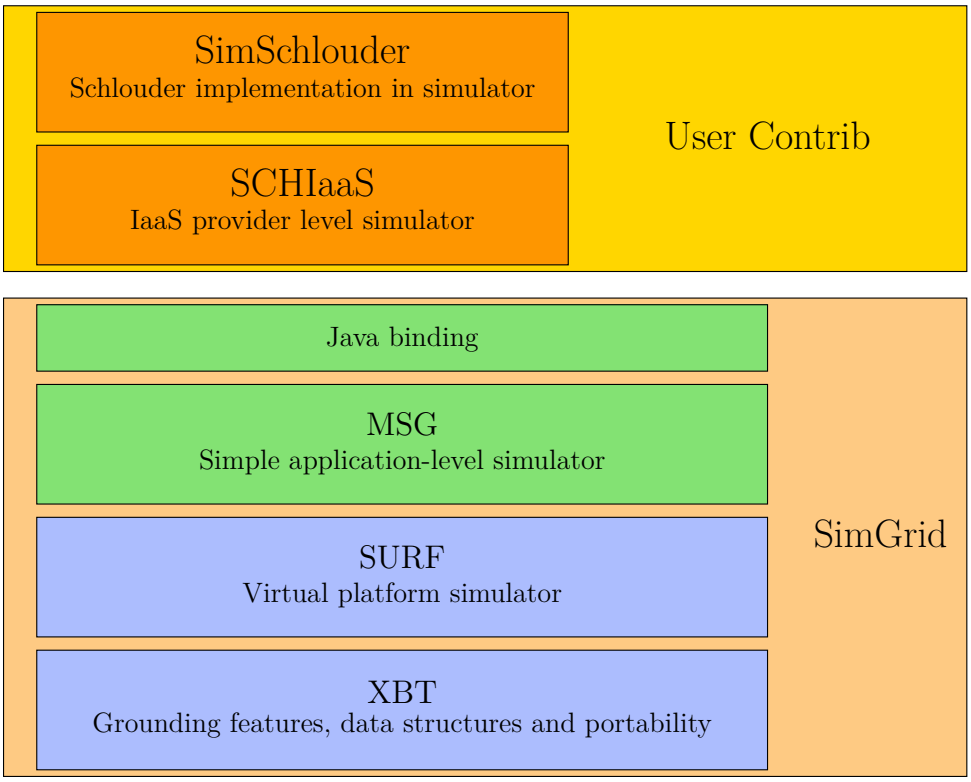


FIGURE 4.1 – L'architecture de SimSchlouder

Il est important de noter que SimGrid n'est pas un simulateur. Il s'agit d'un framework générique facilitant le développement d'un simulateur. Nous n'en expliquerons pas dans cette section l'ensemble des fonctionnalités. Nous nous concentrerons sur les seules fonctionnalités qui nous ont été utiles :

- une boite à outils facilitant le développement d'applications en C appelé XBT. Elle apporte notamment des outils portables concernant l'allocation mémoire et la gestion des chaines de caractères, mais également de nouvelles fonctionnalités telles qu'une gestion des exceptions ainsi que des structures de données classiquement utilisées (pile, file, tas, etc.).

- un moteur de simulation extensible qui implémente des modèles de simulation validés et qui permet la simulation de topologies réseau complexes, de machines de calcul variées, ainsi que des échecs des différentes ressources (SURF) ;

- une interface utilisateur haut-niveau pour permette le développement d'un simulateur en C ou en Java (MSG) ;

Attardons-nous sur le module MSG, support de la simulation avec SCHIaaS. Le module MSG fournit une API permettant le développement d'un simulateur d'applications distribuées. Sa conception permet à l'utilisateur de se concentrer principalement sur les problèmes métiers qu'il souhaite résoudre. MSG inclut une couche hyperviseur permettant

de simuler des machines virtuelles. Elle offre la possibilité de simuler une infrastructure dynamique. Les opérations classiques sur les machines virtuelles sont disponibles telles que le démarrage, l'arrêt ou la migration.

MSG permet la simulation précise d'une application distribuée ainsi que des communications et synchronisations impliquées classiquement dans ce type de logiciel.

4.2.2 SCHIaaS

SCHIaaS [59] est la couche qui s'occupe de simuler un fournisseur de cloud IaaS. Alors que SimGrid implémente les fonctionnalités de l'hyperviseur, SCHIaaS expose une API implémentant les fonctionnalités du fournisseur d'IaaS. Ses principales fonctionnalités sont les suivantes :

- démarrer, arrêter, suspendre, reprendre une instance ;
- description des ressources disponibles ;
- gestion des types d'instances et d'images ;
- placement automatique des VM sur les machines physiques du cluster ;
- gestion du stockage central de l'IaaS.

SCHIaaS propose une interface améliorée par rapport à la couche hyperviseur de SimGrid. Par défaut, deux moteurs sont intégrés à SCHIaaS : RICE et RISE. Ils gèrent respectivement les VM et le stockage. Ce découpage permet à l'utilisateur d'implémenter n'importe quel type de fournisseur d'IaaS, quel que soit son cloud kit, son modèle économique et son modèle de gestion des ressources. Nous avons utilisé dans nos travaux les deux modèles par défaut.

4.2.3 SimSchlouder

L'objectif de SimSchlouder [62] est de simuler l'exécution de Schlouder. Il a été développé dans l'esprit de mimer le plus précisément possible le fonctionnement interne de Schlouder afin d'obtenir des résultats de simulation les plus proches possible de la réalité.

SimSchlouder permet donc la soumission d'un workload, au même format que dans Schlouder, correspondant à un *bag-of-tasks* ou un workflow. Les sorties de SimSchlouder sont le cout et le *makespan* ainsi qu'un fichier JSON, identique à celui fourni par Schlouder et décrit dans la Section 3.2.3. Ces similitudes nous permettent l'utilisation des mêmes scripts pour l'analyse des résultats produits par SimSchlouder et Schlouder. Nous allons voir dans la section suivante les avantages d'une telle convergence pour l'évaluation du simulateur.

Dans le but d'utiliser ce simulateur comme outil d'aide à la décision, il est important que l'exécution de chaque simulation soit courte. Dans notre cas, le temps de simulation pour chaque stratégie est de l'ordre de la seconde, *sans* avoir activé les coroutines dans la JVM. L'activation de ces coroutines permettrait de réduire grandement ce temps.

4.3 Évaluation

Cette section a pour objectif d'évaluer la précision des simulations en utilisant SimSchlouder en comparaison avec des exécutions réelles en utilisant Schlouder. Pour cela, nous détaillons dans un premier temps les workloads qui ont été simulés dans la Section 4.3.1, puis la démarche que nous avons suivie pour configurer le simulateur dans la Section 4.3.2. Nous introduisons dans la Section 4.3.3 deux métriques permettant de vérifier la précision de SimSchlouder. Enfin, nous comparons les résultats de la simulation avec les exécutions réelles dans les sections 4.3.4 et 4.3.5.

4.3.1 Les workloads

Pour effectuer la comparaison d'exécutions réelles et simulées, nous reprenons les workloads (Montage et OMSSA) et les résultats présentés dans la Section 3.4.2. Nous simulerons donc un total de 206 exécutions de type *bag-of-tasks* et *workflows* de durées variées.

4.3.2 Dispositif expérimental

Pour fournir des résultats précis, ce simulateur nécessite des informations précises sur les capacités de calcul des VM et les caractéristiques des liens réseau les connectant. Dans le contexte d'un IaaS public, ces informations ne sont pas nécessairement disponibles, ou alors de manière parcellaire. Nous nous sommes donc mis à la place d'un client d'un tel cloud et nous avons exécuté un job de monitoring proposé par Schlouder sur un grand nombre de VM sur les clouds icps-os-cloud, BonFIRE-de-hlrs, BonFIRE-fr-inria et BonFIRE-uk-epcc.

Ce job exécute dans un premier temps LINPACK pour déterminer la puissance CPU de chacune des VM en flops. Ce logiciel est reconnu par la communauté scientifique. Il s'agit par exemple du test de performance utilisé pour classer les plus puissants ordinateurs du monde dans le cadre du Top500[2].

Concernant les liens réseau, SimGrid nécessite des informations sur la latence et la bande passante de l'ensemble des liens. Nous avons donc choisi d'exécuter ping entre l'ensemble des VM entre elles (VM <-> VM) et entre les VM et le système de courtage (VM <-> broker) qui est en dehors du réseau du fournisseur d'IaaS pour mesurer la

2. `http://www.top500.org/`

TABLE 4.1 – Caractéristiques des plateformes de cloud IaaS

IaaS	CPU (flops)	Bande passante (octets par seconde)		Latence (seconde)	
		VM <-> VM	VM <-> broker	VM <-> VM	VM <-> broker
icps-os-cloud	9.5×10^9	120×10^6	120×10^6	100×10^{-6}	100×10^{-6}
BonFIRE-fr-inria	8.8×10^8	938×10^6	41×10^6	270×10^{-6}	15×10^{-3}
BonFIRE-uk-epcc	1.2×10^9	942×10^6	19×10^6	267×10^{-6}	96×10^{-3}
BonFIRE-de-hlrs	1.7×10^9	940×10^6	28×10^6	170×10^{-6}	55×10^{-3}

latence. Enfin, nous exécutons iperf entre l'ensemble des VM entre elles et entre les VM et le système de courtage pour connaitre la bande passante disponible.

Nous avons obtenu les résultats présentés dans la Table 4.1. Nous notons dans un premier temps que le cloud icps-os-cloud se distingue des autres par des VM plus puissantes. La bande passante y est en revanche bien inférieure à celle des autres IaaS. Entre les différents sites géographiques de BonFIRE, nous distinguons quelques différences. La puissance des CPU sur BonFIRE-fr-inria est légèrement inférieure que sur les deux autres sites et est équivalente entre ces deux sites. Les capacités du réseau sont similaires sur les trois sites de BonFIRE. Nous exécutons donc nos workloads sur quatre plateformes hétérogènes et représentatives.

Le simulateur nécessite, enfin, de connaitre une prédiction de la durée des jobs. Nous lui fournissons donc la même prédiction que celle fournie à Schlouder au cours des expériences présentées dans la Section 3.3.2.

4.3.3 Métriques

Afin de vérifier la correction des simulations réalisées avec SimSchlouder, nous présentons dans cette section deux nouvelles métriques. Elles ont pour objectif de faciliter la comparaison entre un résultat obtenu par la simulation avec SimSchlouder et un résultat obtenu par une exécution réelle avec Schlouder.

Efficacité des stratégies

Afin de définir l'efficacité e de la stratégie $ASAP$ par rapport à la stratégie $AFAP$, nous définissons d'abord le speedup et le costup. Le speedup est défini dans l'Équation 4.1 comme étant le rapport entre le *makespan* en utilisant la stratégie $AFAP$ et le *makespan* en utilisant la stratégie $ASAP$. Par exemple, un speedup de 2 signifie que l'exécution d'un workload en utilisant $AFAP$ est deux fois plus longue qu'avec la stratégie $ASAP$.

$$\text{speedup} = \frac{\text{Makespan}_{AFAP}}{\text{Makespan}_{ASAP}} \tag{4.1}$$

Le costup est défini dans l'Équation 4.2 comme étant le rapport entre le cout de l'exécution d'un workload en utilisant la stratégie $ASAP$ et le cout en utilisant la stratégie

AFAP. Autrement dit, un costup de 2 signifie que l'exécution d'un workload en utilisant *ASAP* coute deux fois plus cher qu'avec la stratégie *AFAP*.

$$\text{costup} = \frac{\text{cost}_{ASAP}}{\text{cost}_{AFAP}} \tag{4.2}$$

Nous pouvons maintenant définir l'efficacité de la stratégie dans l'Équation 4.3. Elle est définie comme étant le ratio entre le speedup et le costup. Nous définissons cette métrique pour connaitre le bénéfice en terme de temps pour chaque dollar investi dans la location de l'infrastructure.

$$e = \frac{\text{speedup}}{\text{costup}} \tag{4.3}$$

Nous avons donc les valeurs suivantes de e qui signifient :

- $e = 1$: *ASAP* augmente le cout autant qu'il accélère l'exécution. Par exemple, l'utilisateur paye deux fois plus en utilisant *ASAP* dans le but de diviser par deux le temps de calcul.
- $e > 1$: *ASAP* augmente moins le cout qu'il n'accélère l'exécution du calcul. Par exemple, l'utilisateur paye 1,5 fois plus en utilisant *ASAP* dans le but de diviser par deux le temps de calcul.
- $e < 1$: *ASAP* augmente plus le cout qu'il n'accélère l'exécution du calcul. Par exemple, l'utilisateur paye 3 fois plus en utilisant *ASAP* dans le but de diviser par deux le temps de calcul.

Cette métrique nous permet de comparer l'avantage d'utiliser la stratégie *ASAP* par rapport à la stratégie *AFAP*.

Taux d'erreur

Nous définissons ensuite le taux d'erreur comme étant une métrique permettant de comparer les mesures observées lors de l'exécution réelle et celles obtenues par la simulation. Soit une mesure réelle m et la même mesure obtenue par simulation m'. Nous définissons le taux d'erreur comme étant $\text{err} = \frac{|m-m'|}{m}$. m correspondra à une des métriques suivantes :

- efficacité d'une exécution (e) ;
- nombre de BTU consommées (cost) ;
- temps d'exécution total du workload (makespan).

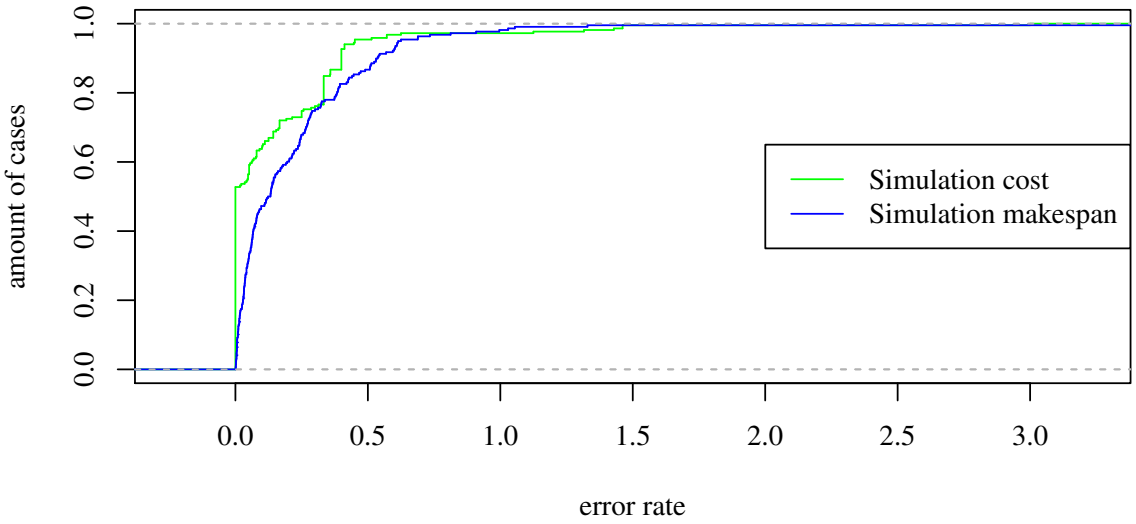


FIGURE 4.2 – Précision de SimSchlouder pour le *makespan* et le cout

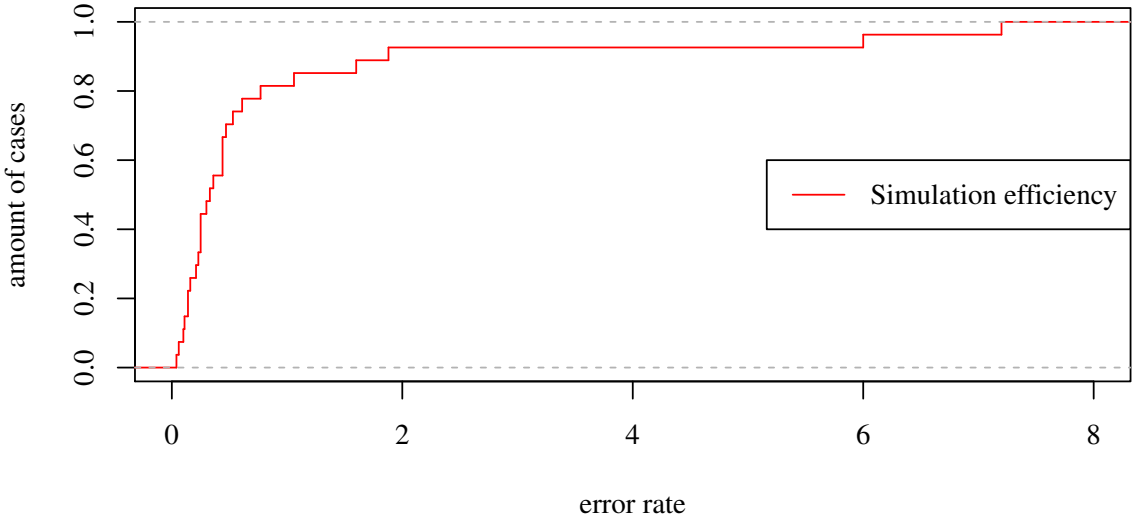
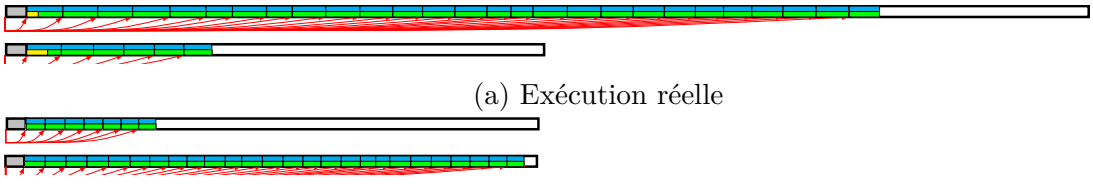


FIGURE 4.3 – Précision de SimSchlouder pour l'efficacité



(a) Exécution réelle

(b) Simulation

FIGURE 4.4 – Exemple de comparaison grâce au script de visualisation des BTU

4.3.4 Précision globale de la simulation

La Figure 4.2 montre la fonction de répartition cumulative du taux d'erreur pour le cout et le temps d'exécution total. Le cout est parfaitement prédit ($err = 0$) pour la moitié des 218 cas. Ce taux d'erreur est inférieur à 0,5 dans 95% des cas. Le taux d'erreur du temps total d'exécution est inférieur à 0,5 dans 87% des cas. Ces deux taux d'erreur n'excèdent jamais 3,0.

Nous montrons sur la Figure 4.4 un exemple de l'exécution de *brt* sur BonFIRE-uk-epcc en utilisant la stratégie *AFAP*. La Figure 4.4a représente l'exécution réelle et la Figure 4.7b l'exécution simulée. Nous constatons bien que la simulation a prédit un cout de 2 BTU alors que l'exécution réelle a couté 3 BTU. De plus, le temps total prédit par l'exécution simulée est quasiment de 3 600 secondes alors qu'il est de 5 775 seconds pour l'exécution réelle.

4.3.5 Précision de la comparaison des stratégies

La Figure 4.3 montre la fonction de répartition cumulative du taux d'erreur de l'efficacité. Elle est parfaitement prédite pour la moitié des 218 cas. Ce taux d'erreur est inférieur à 1,0 dans 81% des cas. Cependant, une petite partie des simulations montre un écart considérable entre l'efficacité prédite et réelle avec un maximum de 7,2.

4.3.6 Les raisons de l'imprécision du simulateur

Nous avons constaté que le simulateur peut, dans certains cas, donner des résultats éloignés de la réalité. Dans le but de comprendre ces imprécisions, nous avons à nouveau simulé ces 218 cas en utilisant le *walltime* prédit pour l'ordonnancement, mais le *walltime* réel pour la simulation des tâches. Nous utilisons maintenant le *walltime* réel de chaque job, tel que nous l'avons mesuré lors de son exécution. Nous souhaitons ainsi supprimer les imprécisions de la simulation due à une mauvaise prédiction du *walltime* par l'utilisateur.

Les résultats sont montrés dans les figures 4.5 et 4.6. L'efficacité est très bien simulée dans tous les cas ($\max_{\text{err}} = 0,30$). Les taux d'erreur du cout et du temps d'exécution total sont maintenant inférieurs à 0,1 pour 96% des cas.

Nous montrons ainsi que la principale source d'imprécision du simulateur est due à la prédiction du *walltime* fourni par l'utilisateur et non à un problème interne à SimSchlouder.

Nous montrons sur la Figure 4.7 l'exemple de l'exécution de *brt* sur BonFIRE-uk-epcc en utilisant la stratégie *AFAP*. La Figure 4.7a représente l'exécution réelle et la Figure 4.7c l'exécution simulée. Nous constatons que la simulation prédit maintenant correctement le cout de 3 BTU de l'exécution réelle. De plus, le temps total de l'exécution simulée est maintenant de 5 536 secondes et de 5 775 seconds pour l'exécution réelle.

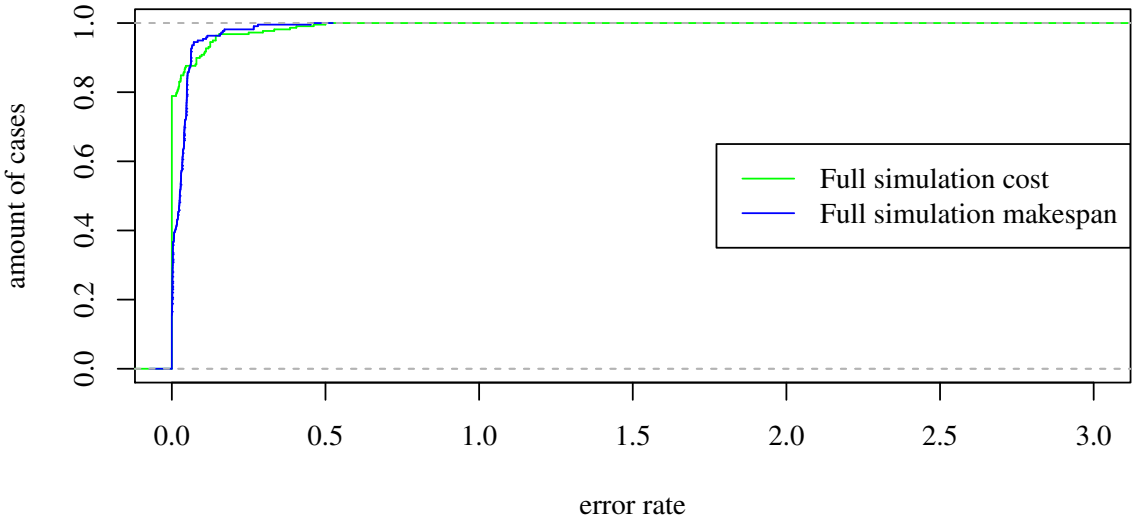


FIGURE 4.5 – Précision de SimSchlouder pour le *makespan* et le cout

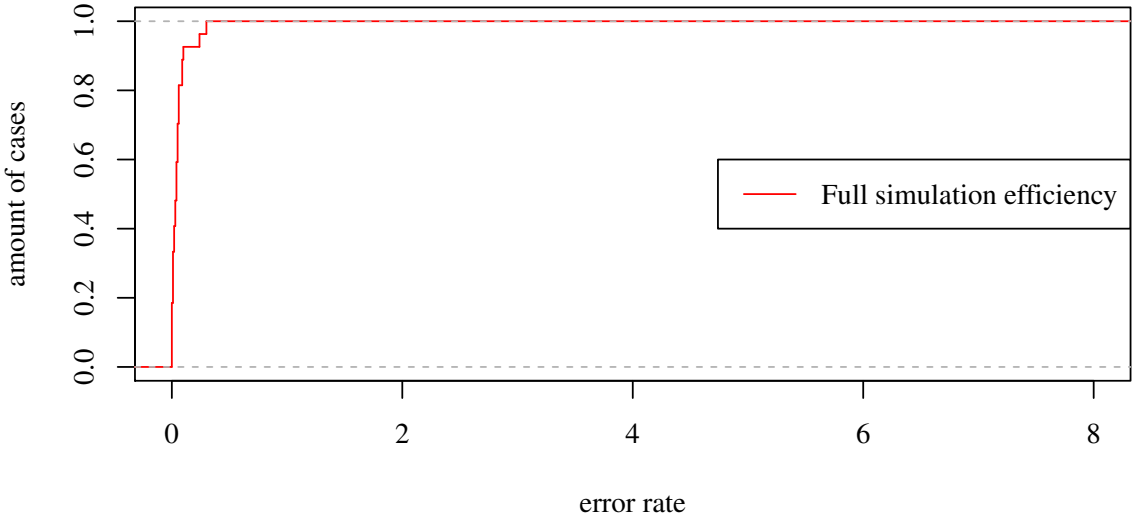
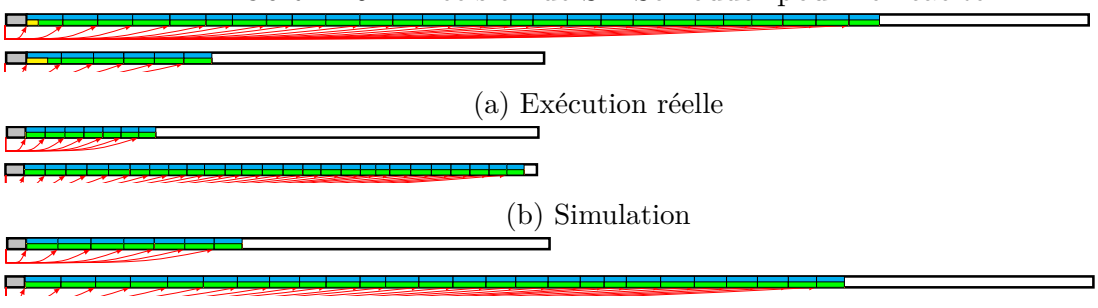


FIGURE 4.6 – Précision de SimSchlouder pour l'efficacité



(a) Exécution réelle

(b) Simulation

(c) Full simulation

FIGURE 4.7 – Exemple de comparaison grâce au script de visualisation des BTU

Les sources des imprécisions restantes (4% des cas) sont doubles. Premièrement, le surcout dû à la gestion des jobs par Schlouder et le *batch scheduler* n'est pas simulé. Il implique de toutes petites erreurs dans le temps d'exécution total qui sont négligeables.

Deuxièmement, ces simulations ont révélé un bug de Schlouder qui cause une mauvaise décision de *provisioning* dans de rares situations. Ce bug est causé par l'exécution simultanée des threads de *provisioning* et de soumission. Lorsque l'action de *provisioning* survient au milieu de la soumission du workload, notre prédiction du temps de démarrage de la VM est erronée. Cet entrelacement des threads rend ce bug difficilement détectable. Nous avons pu l'identifier grâce aux résultats de la simulation que nous avons analysés à l'aide de notre script de visualisation des BTU. Ce bug a pu être corrigé et nous nous sommes assurés que seules ces deux raisons sont à l'origine des imprécisions du simulateur.

Finalement, nous pouvons conclure que SimSchlouder est un simulateur d'IaaS très précis. Lorsque l'utilisateur sait prédire avec justesse les *walltimes* de ses jobs, il peut être utilisé pour prédire le comportement des stratégies, quels que soient le workload et le fournisseur d'IaaS. De plus, SimSchlouder s'est montré très utile dans la vérification de l'implémentation des stratégies de *provisioning*, et peut donc servir au développement de nouvelles stratégies et à leur évaluation.

4.4 Conclusions

Nous avons vu dans ce chapitre que la simulation d'applications distribuées est une tâche difficile. Cependant, elle est facilitée par l'homogénéité des performances de l'IaaS et l'isolation entre les différents utilisateurs grâce à la virtualisation. Schlouder propose un Simulation Engine capable de prédire précisément le *makespan*, le cout et l'ordonnancement des jobs sur les VM avec différentes stratégies de *provisioning*.

Bien que le Simulation Engine nécessite une description détaillée de la plateforme physique — qui n'est souvent pas connue dans le cas d'un IaaS public — il est possible de découvrir la bande passante, la latence et la puissance CPU des ressources à l'aide d'outils de benchmark disponibles et ainsi modéliser la plateforme, au moins du point de vue de l'utilisateur.

Nous avons montré que le simulateur fournit des résultats très précis avec un taux d'erreur inférieur à 0,1 pour la quasi-totalité des cas. Cette précision nous a également permis de débusquer un bug d'entrelacement de threads dans le logiciel Schlouder.

Chapitre 5

Conclusions et travaux futurs

5.1 Conclusions

Dans le contexte du cloud computing, l'IaaS fournit des ressources de calcul virtualisées à la demande suivant un modèle de paiement à l'utilisation. Du point de vue de l'utilisateur, ce nouveau paradigme fournit un stock inépuisable de ressources, qui peuvent être dynamiquement demandées et relâchées. L'IaaS permet l'exécution de calculs scientifiques sur un budget de fonctionnement plutôt que sur un investissement initial important. Le grand nombre de tâches et de ressources à prendre en compte pour réaliser le *provisioning*, mais également le grand nombre de plateformes et de modèles économiques disponibles rendent l'ordonnancement sur une telle plateforme élastique difficile. Nous avons abordé ce problème en concevant un système de courtage côté client capable (1) d'automatiser le *provisioning* en fonction d'une stratégie sélectionnée par l'utilisateur et (2) de simuler l'exécution afin de fournir à l'utilisateur une estimation des couts et temps qu'impliquent les différentes stratégies. Son architecture ouverte permet de s'adapter à un grand nombre de fournisseurs de cloud et de stratégies de *provisioning*. Des expérimentations à grande échelle ont été menées sur plusieurs plateformes de clouds avec des applications de type *bag-of-tasks* et *workflows*. Elles montrent la capacité de nos outils à exécuter différents types de workloads sur des plateformes variées et à simuler avec une grande précision ces exécutions.

Nous avons vu tout au long de cette thèse que ce type de service est amené à être utilisé par des utilisateurs non informaticiens. Nous souhaitons conclure l'exposé de nos travaux de recherche en nous positionnant du point de vue de l'utilisateur.

5.1.1 Quelle taille de plateforme pour quels workloads ?

Alice est chercheuse en protéomique. Pour sa recherche, elle a régulièrement besoin d'exécuter des calculs. Ne disposant pas d'une plateforme, et ses calculs étant ponctuels, Alice souhaite exécuter des calculs sur le cloud IaaS, ce qui correspond mieux à ses budgets qui sont essentiellement du fonctionnement. Dans un premier temps, Alice demande à Bob, l'ingénieur de l'équipe, de l'aider à déployer quelques VM. Ensemble, ils développent un script automatisant les calculs. Cependant, Alice se rend compte qu'elle maitrise mal les couts et les temps d'exécution de ses calculs. De plus, certains jours Alice a besoin

d'obtenir rapidement des résultats pour satisfaire une échéance proche telle qu'une deadline pour la soumission d'un article. À d'autres moments, elle peut se permettre d'utiliser moins de machines et ainsi obtenir ses résultats plus tard.

Du côté de Bob, il doit régulièrement adapter ses scripts en fonction des performances réelles de la plateforme. De plus, l'application d'Alice n'est pas constante et elle en change souvent les paramètres. Et puis de nombreuses situations forcent Alice à reprendre ses calculs depuis le début (VM qui ne bootent pas, qui ne répondent pas, panne réseau, etc.). De plus, Alice et Bob essaieraient volontiers plusieurs plateformes de clouds et plusieurs types de VM, plusieurs stratégies de *provisioning* et plusieurs modèles économiques. Mais leur script d'exécution devient déjà très complexe et est de plus en plus difficile à maintenir.

Bob décide donc de la mise en place d'un système de courtage afin d'améliorer la prise en charge des calculs d'Alice. Ce système de courtage doit permettre à Alice de choisir parmi un grand nombre de stratégies de *provisioning* automatiques.

5.1.2 Automatisation du *provisioning* au travers d'un courtier

Afin de choisir un système de courtage approprié à ses besoins, Bob fait un état de l'art des solutions existantes. Certaines solutions sont de type PaaS et contraignent donc à l'utilisation d'un langage de programmation pour les logiciels à exécuter ce qui ne convient pas à Alice et ses collègues. Les autres solutions sont des systèmes de courtage à utiliser conjointement avec une plateforme de type IaaS. Bob souhaite que la solution choisie soit open source afin de pouvoir s'assurer que le courtier ne réalise pas d'actions malveillantes. Ainsi, parmi les solutions qu'il a trouvées Schlouder est le projet qui correspond le mieux à ses besoins : orienté calcul scientifique, permettant la personnalisation des stratégies de *provisioning*, et open source.

Bob installe donc le système de courtage Schlouder, afin de concrètement mettre en œuvre le *provisioning* de manière automatique pour Alice. Alice parvient effectivement à réaliser ses calculs en choisissant une stratégie de *provisioning* parmi une grande liste de choix. Ce choix dépend des soumissions d'Alice et de sa compréhension de l'action de ces stratégies. Certaines stratégies telles qu'*ASAP* ont un nom qui permettent intuitivement de comprendre son action. D'autres telles qu'*AFAP* nécessitent qu'Alice lise la documentation disponible pour comprendre qu'il s'agit d'une stratégie plus économique.

Bob est satisfait de l'automatisation du processus de *provisioning* qui ne nécessite plus un effort d'adaptation constant. Néanmoins, Alice a du mal à faire le choix de la stratégie de *provisioning*. En effet, malgré l'intention affichée des stratégies, elle remarque que les gains de performances ou de cout varient de façon imprédictible en fonction de ses calculs. Elle utilise donc le Simulation Engine intégré à Schlouder.

5.1.3 Aide à la décision de la stratégie de *provisioning*

Le Simulation Engine intégré à Schlouder permet à Alice d'obtenir facilement le cout, le *makespan* et l'ordonnancement des jobs pour l'exécution de son workload avec l'ensemble des stratégies.

Alice a une échéance afin de soumettre un article pour le lendemain à 16 h. Sa volonté est d'utiliser la solution la plus rapide, quoi qu'il en coute. En utilisant le simulateur, elle constate que l'utilisation d'*AFAP* lui couterait 51€, et finirait l'exécution à 14 h. L'utilisation d'*ASAP* couterait 98€ pour une diminution de 2 h du temps d'exécution. Le résultat des simulations permet à Alice de se rendre compte qu'utiliser *ASAP* implique une grande augmentation du cout pour un gain faible. De plus, les résultats arriveraient pendant sa pause déjeuner. Alice choisit donc finalement la solution économe en exécutant ses calculs avec la stratégie *AFAP*.

Pour une future session de calculs, Alice souhaiterait utiliser une nouvelle stratégie de *provisioning* qui garantit un certain niveau de SLA. Elle demande donc à Bob de l'implémenter au sein de Schlouder. Pour cela, Bob la développe au sein du simulateur et réalise des tests afin de s'assurer de la qualité de son algorithme. Bob peut valider sa stratégie dans le cas général grâce aux nombreux workloads réels fournis avec Schlouder, qui sont représentatifs des applications scientifiques. De plus, ces workloads comprenant les runtimes prédits et réels, Bob peut éprouver sa stratégie face à différents niveaux de précision dans la prédiction des runtimes. Ces tests lui permettent de remarquer des cas particuliers pour certains workloads et ainsi d'affiner son algorithme afin de les prendre en compte.

Alice peut donc maintenant exécuter ses calculs sur une plateforme de cloud IaaS en utilisant son budget de fonctionnement. L'utilisation du courtier lui permet de maitriser les couts et les performances de ses calculs grâce à différentes stratégies de *provisioning*. La présence d'un outil permettant d'obtenir un retour, avant l'exécution, sur les conséquences du choix des stratégies de *provisioning* permet à Alice d'avoir toutes les informations en main pour planifier ses campagnes de calculs.

5.2 Travaux futurs

Les travaux futurs s'articulent autour de deux axes principaux. D'un point de vue technique, nous proposons le développement d'une nouvelle stratégie s'appuyant notamment sur le simulateur ainsi que l'amélioration du simulateur, et la faisabilité d'adapter Schlouder pour l'utilisation de la virtualisation applicative. Ensuite, nos perspectives concernent la découverte des caractéristiques de la plateforme physique ainsi que la création d'une stratégie de *provisioning* prenant en compte les nouvelles façons de stocker les données.

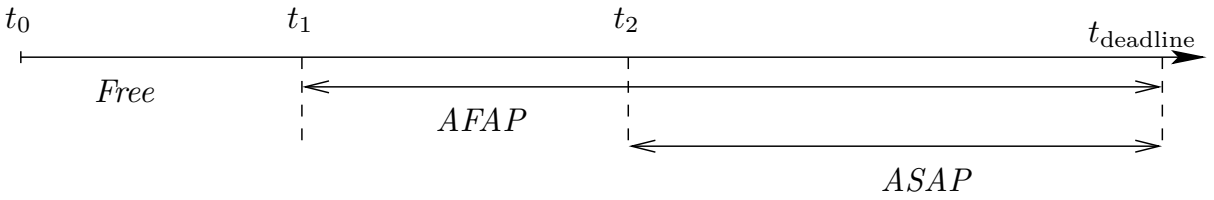


FIGURE 5.1 – Exemple de méta-stratégie

5.2.1 Perspectives techniques

Conception d'une méta-stratégie de *provisioning*

L'architecture ouverte de Schlouder permet l'ajout et le partage de stratégies de *provisioning*. Le nombre des stratégies peut donc devenir grand et poser un problème à l'utilisateur dans le choix de la stratégie qui convient le mieux à ses besoins.

Afin de résoudre ce problème, une solution serait de proposer à l'utilisateur une méta-stratégie qui appliquerait dynamiquement la stratégie de *provisioning* la moins couteuse parmi toutes celles disponibles en fonction d'une échéance définie par l'utilisateur. Nous présentons dans la Figure 5.1 un exemple d'application de cette méta-stratégie. Nous y introduisons une nouvelle stratégie de *provisioning*, *Free*, qui ordonnance un job sur une VM existante si cela permet d'en réduire le temps d'inactivité avant la fin de la BTU. À t_0, l'utilisateur soumet son workload ainsi que sa deadline. Nous simulons alors son exécution en utilisant les stratégies *AFAP* et *ASAP* pour en connaitre la durée. Nous obtenons respectivement les durées $t_1 - t_{\text{deadline}}$ et $t_2 - t_{\text{deadline}}$. Nous exécutons alors le workload en utilisant ces différentes stratégies : de t_0 à t_1 en utilisant *Free*, de t_1 à t_2 en utilisant *AFAP* et enfin de t_2 à t_{deadline} en utilisant *ASAP*.

Les deux avantages principaux de cette méta-stratégie sont qu'elle permet d'intégrer de façon transparente un nombre quelconque de stratégies de *provisioning* et qu'elle nécessite de l'utilisateur une valeur simple et non technique qui est l'échéance du calcul.

La réalisation d'une telle méta-stratégie soulève des questions dans la gestion des cas où l'échéance définie par l'utilisateur est impossible à réaliser. L'utilisateur souhaite-t-il alors effectuer les calculs au plus vite ? Ou alors rater l'échéance doit-il être associé à l'utilisation de la stratégie la moins couteuse ? La création d'un système interactif guidant l'utilisateur pourrait alors faciliter la configuration de cette méta-stratégie.

Ce type de stratégie serait très sensible aux résultats de la simulation qui elle-même dépend des prédictions des temps d'exécution des jobs.

Amélioration des résultats du simulateur

Les résultats du Simulation Engine dépendent essentiellement de la prédiction du temps d'exécution des jobs en fonction de la stratégie simulée. Certaines stratégies, comme *ASAP*, sont dites stables. La décision que prennent ces stratégies ne dépend pas de la

prédiction fournie par l'utilisateur. D'autres, comme *AFAP*, sont moins stables, car le *provisioning* réalisé par ces stratégies dépend fortement de la prédiction fournie par l'utilisateur. Nous ne pouvons cependant pas agir sur cette prédiction qui est dépendante de l'application et est sous la responsabilité de l'utilisateur. Une mauvaise prédiction entrainera une simulation faussée de la durée des jobs.

Afin d'éviter une perte de confiance par l'utilisateur dans le simulateur, une possibilité serait d'ajouter une *mesure de confiance* aux résultats du Simulation Engine. Connaissant le taux d'erreur e de la prédiction du *walltime*, nous pouvons simuler l'exécution du workload de l'utilisateur avec deux valeurs différentes : walltime_prediction $\times (1 + e)$ et walltime_prediction $\times (1 - e)$. Nous obtenons ainsi un intervalle des résultats possibles. Cet intervalle permet d'obtenir une mesure de la stabilité des différentes stratégies. Ceci nous donne donc une mesure de la confiance que l'utilisateur peut accorder aux résultats du Simulation Engine.

Cela complexifie les informations données par le simulateur. L'utilisateur pourra faire un choix plus éclairé au regard de ces besoins et des risques qu'il est prêt à prendre. Cependant, nous n'avons aucune garantie que les résultats du simulateur sont linéaires entre les deux bornes walltime_prediction $\times (1 + e)$ et walltime_prediction $\times (1 - e)$. Cet aspect demandera une campagne de validation sur de nombreuses stratégies et de nombreux workloads dont le protocole expérimental ne sera pas facile à préparer sachant qu'il ne sera pas possible de réaliser des simulations sur l'ensemble des valeurs entre les bornes.

Intégration des conteneurs

Depuis quelques années, nous avons vu l'émergence d'un nouveau type de virtualisation. Souvent appelé conteneur, il s'agit d'une technologie proposée par le noyau du système d'exploitation qui permet l'isolation de l'espace utilisateur. Il est utilisé pour isoler l'exécution de plusieurs environnements du système d'exploitation dans des conteneurs partageant le même noyau. Nous parlons également de virtualisation au niveau du système d'exploitation. Cette technologie présente de nombreux avantages dont la forte réduction (voir la suppression) du surcout qu'implique l'utilisation de machine virtuelle. Docker[1] est un logiciel pour Linux qui a popularisé cette technologie en facilitant le déploiement de conteneurs. À l'image de l'IaaS qui a commencé en étant utilisé dans un contexte web, les conteneurs sont aujourd'hui majoritairement utilisés pour exécuter des services web.

Nous pouvons nous poser deux questions : l'intégration des conteneurs à Schlouder permettrait-elle d'utiliser ce courtier dans un contexte web ? Si oui, quel intérêt cela représente-t-il ? Les stratégies actuellement intégrées ne sont pas pertinentes dans un contexte web, mais l'architecture ouverte de Schlouder permet aux utilisateurs d'y re-

1. `https://www.docker.com/`

médier. De telles stratégies peuvent nécessiter d'accéder à des informations telles que la charge CPU. Schlouder donne-t-il accès aux informations nécessaires ? D'un autre côté, cela permettra d'étudier ce que les conteneurs peuvent apporter pour l'exécution de calculs scientifiques.

5.2.2 Perspectives scientifiques

Partage des données sur les infrastructures

Nous avons vu que la simulation nécessite de connaitre, au moins du point de vue des utilisateurs, les caractéristiques de la plateforme d'exécution. Or, les caractéristiques réelles de l'infrastructure physique dans un contexte IaaS sont très souvent couvertes par le secret industriel. L'acquisition de ces caractéristiques est pourtant très importante pour la précision de la simulation.

Une solution serait d'exploiter les capacités de monitoring de Schlouder permettant d'explorer les capacités réelles des VM. Pour l'heure, l'utilisateur doit se contenter des informations qu'il a lui-même collectées. Ceci l'empêche de savoir comment va se comporter son application sur d'autres plateformes. Nous souhaitons proposer un module pair-à-pair capable de partager les informations collectées par l'ensemble des utilisateurs de Schlouder sur différentes infrastructures IaaS. Cela permettrait de créer une base de connaissances globale permettant à l'utilisateur de choisir à priori la plateforme sur laquelle s'inscrire pour lancer sa campagne d'exécution.

Outre les problèmes techniques inhérents au pair-à-pair — tel que la définition de la DHT, la sécurité et la confidentialité des données — il sera particulièrement difficile de mettre en cohérence des données variables au cours du temps et provenant d'un grand nombre de sources. En effet, les capacités d'une VM dépendent de la machine physique sur laquelle elle s'exécute ainsi que de la charge de cette machine physique. De plus, l'infrastructure physique évolue au cours du temps sans notification à l'utilisateur. Des travaux de recherche devront être menés entre autres sur les questions suivantes : quelles métriques sont les plus pertinentes pour la simulation ? Comment agréger des mesures provenant d'un grand nombre de sources ? Quelle pertinence accorder à des mesures hétérogènes ?

Des stratégies de *provisioning* sur la gestion des données

Nous avons vu qu'une partie des applications telles que Montage sont intensives en communication. Les transferts de données ainsi que le stockage sur un service du fournisseur d'IaaS étant payants, en plus des performances, les couts de l'exécution dépendent également de la façon de gérer les données. La difficulté dans la gestion de ces données vient du fait que de nombreuses solutions sont possibles. Il est donc important d'aider l'utilisateur

à faire un choix éclairé. Nous souhaiterions, dans un premier temps, tester les différentes stratégies de gestion des données proposées par Deelman et al. [14] :

Remote IO Les données en entrées sont récupérées pour l'exécution de la tâche. Les données en sortie sont uploadées sur le serveur de stockage central du cloud. Aucune donnée n'est conservée sur la VM.

Regular Si le cloud propose un espace de stockage à partager entre les différentes VM (p.ex. NFS), cette stratégie ne supprime aucune donnée intermédiaire des VM, jusqu'à la fin de l'exécution du *workflow*.

Dynamic cleanup Cette stratégie a pour objectif d'améliorer la stratégie Regular en supprimant les données intermédiaires lorsque tous les jobs en ayant besoin ont terminé leur exécution.

L'architecture actuelle de Schlouder ne permet pas, pour le moment, d'implémenter ce type de stratégie. En effet, notre courtier n'a aucune connaissance du contenu des jobs à exécuter, et donc des données qui sont utilisées au cours de l'exécution d'un *workflow*. Nous souhaiterions donc intégrer à Schlouder une interface d'accès aux données.

De plus, des expériences préliminaires montrent des difficultés à simuler proprement les communications au sein d'un cloud. En effet, les communications sont très dépendantes de l'activité de l'ensemble des utilisateurs de la plateforme physique. Sur un cloud public, ces activités sont impossibles à connaitre, dynamiques et imprédictibles. Les travaux réalisés sur le monitoring permettraient d'obtenir des informations sur la stabilité actuelle de la plateforme.

5.2.3 Le mot de la fin

Pour conclure, les problèmes à adresser concernant la gestion des ressources sont très différents depuis l'avènement du cloud IaaS. En effet, alors que les efforts principaux sur la grille visent à organiser empiriquement le workload et trouver un bon pattern de soumission afin d'utiliser le plus de ressources possible, l'effort principal sur le cloud est de définir le nombre de ressources qui doivent être provisionnées en fonction des objectifs de l'utilisateur.

De façonner l'application en fonction de la plateforme à façonner la plateforme en fonction de l'application : le cloud IaaS représente un changement majeur dans les problématiques liées au calcul scientifique.

Bibliographie

[1] *Amazon Elastic Beanstalk*, (accessed July 17th 2013), URL : `https://aws.amazon.com/elasticbeanstalk/` (cf. p. 48, 49).

[2] *Amazon Elastic Compute Cloud*, (accessed November 24th 2014), URL : `https://aws.amazon.com/fr/ec2/` (cf. p. 3).

[3] Alain Andrieux et al., "Web services agreement specification (WS-Agreement)", *in* : *Open Grid Forum*, t. 128, 2007 (cf. p. 46, 47).

[4] Marcos de Assunção, Alexandre di Costanzo et Rajkumar Buyya, "A cost-benefit analysis of using cloud computing to extend the capacity of clusters", *in* : *Cluster Computing* 13 (3 2010), p. 335–347, ISSN : 1386-7857 (cf. p. 43, 47).

[5] William H Bell et al., "Optorsim : A grid simulator for studying dynamic data replication strategies", *in* : *International Journal of High Performance Computing Applications* 17.*4* (2003), p. 403–416 (cf. p. 86).

[6] Rodrigo N. Calheiros et al., "CloudSim : a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms", *in* : *Software : Practice and Experience* 41.*1* (2011), p. 23–50 (cf. p. 50, 86).

[7] Pascal Calvat et al., "Two experiments with application-level quality of service on the EGEE grid", *in* : *Proceeding of the 2nd workshop on Grids meets autonomic computing*, juin 2010, DOI : `10.1145/1809029.1809031` (cf. p. 75).

[8] Louis-Claude Canon et Emmanuel Jeannot, "Evaluation and Optimization of the Robustness of DAG Schedules in Heterogeneous Environments", *in* : *IEEE Trans. Parallel Distrib. Syst.* 21.*4* (2010), p. 532–546 (cf. p. 73).

[9] Christine Carapito et al., "MSDA, a proteomics software suite for in-depth Mass Spectrometry Data Analysis using grid computing", *in* : *Proteomics* 14.*9* (2014), p. 1014–1019 (cf. p. 74).

[10] Eddy Caron, Frédéric Desprez et Adrian Muresan, "Pattern matching based forecast of non-periodic repetitive behavior for cloud clients", *in* : *Journal of Grid Computing* 9.*1* (2011), p. 49–64 (cf. p. 43).

[11] Henri Casanova, Arnaud Legrand et Martin Quinson, "SimGrid : a Generic Framework for Large-Scale Distributed Experiments", *in* : *10th IEEE International Conference on Computer Modeling and Simulation*, mar. 2008 (cf. p. 48, 86).

[12] *CloudFoundry*, (accessed July 17th 2013), URL : http://www.cloudfoundry.com/ (cf. p. 48, 49).

[13] Edward G. COFFMAN, M. R. GAREY et David S. JOHNSON, "Approximation algorithms for bin packing : a survey", *in* : Boston, MA, USA : PWS Publishing Co., 1997, p. 46–93, ISBN : 0-534-94968-1, URL : http://www2.research.att.com/~dsj/papers/BPchapter.ps (cf. p. 8, 15).

[14] Ewa DEELMAN et al., "The cost of doing science on the cloud : The Montage example", *in* : *SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, nov. 2008, p. 50, DOI : 10.1109/SC.2008.5217932 (cf. p. 6, 41, 60, 103).

[15] Kefeng DENG et al., "Exploring portfolio scheduling for long-term execution of scientific workloads in IaaS clouds", en, *in* : ACM Press, 2013, p. 1–12, ISBN : 9781450323789, DOI : 10.1145/2503210.2503244 (cf. p. 85).

[16] Brian DOUGHERTY, Jules WHITE et Douglas C SCHMIDT, "Model-driven auto-scaling of green cloud computing infrastructure", *in* : *Future Generation Computer Systems* 28.*2* (2012), p. 371–378 (cf. p. 42).

[17] Ta Nguyen Binh DUONG, Xiaorong LI et Rick Siow Mong GOH, "A Framework for Dynamic Resource Provisioning and Adaptation in IaaS Clouds", *in* : *Cloud Computing Technology and Science, IEEE International Conference on*, Los Alamitos, CA, USA : IEEE Computer Society, 2011, p. 312–319, ISBN : 978-0-7695-4622-3, DOI : http://doi.ieeecomputersociety.org/10.1109/CloudCom.2011.49 (cf. p. 44).

[18] Mattias ELLERT et al., "Advanced Resource Connector middleware for lightweight computational Grids", *in* : *Future Generation computer systems* 23.*2* (2007), p. 219–240 (cf. p. 2).

[19] Hamid Mohammadi FARD, Thomas FAHRINGER et Radu PRODAN, "Budget-Constrained Resource Provisioning for Scientific Applications in Clouds", *in* : IEEE, déc. 2013, p. 315–322, ISBN : 978-0-7695-5095-4, DOI : 10.1109/CloudCom.2013.48, (visité le 22/04/2014) (cf. p. 44).

[20] Ian FOSTER, "Globus toolkit version 4 : Software for service-oriented systems", *in* : *Network and parallel computing*, Springer, 2005, p. 2–13 (cf. p. 2).

[21] Donald K. FRIESEN et Michael A. LANGSTON, "Variable Sized Bin Packing", *in* : *SIAM Journal on Computing* 15.*1* (1986), p. 222–230, DOI : 10.1137/0215016 (cf. p. 8).

[22] L. Y. GEER et al., "Open mass spectrometry search algorithm", *in* : *J Proteome Res.* 3.*5* (2004), p. 958–964 (cf. p. 74).

[23] Stéphane Genaud et Julien Gossa, “Cost-wait Trade-offs in Client-side Resource Provisioning with Elastic Clouds”, *in* : *4th International Conference on Cloud Computing (CLOUD)*, Washington DC, USA : IEEE, juil. 2011 (cf. p. 42, 85).

[24] *Google App Engine*, (accessed July 17th 2013), URL : `https://developers.google.com/appengine/` (cf. p. 45, 46, 49).

[25] Alexandru Iosup, Ozan Sonmez et Dick Epema, “Dgsim : Comparing grid resource management architectures through trace-based simulation”, *in* : *Euro-Par 2008–Parallel Processing*, Springer, 2008, p. 13–25 (cf. p. 85).

[26] Alexandru Iosup et al., “How are Real Grids Used ? The Analysis of Four Grid Traces and Its Implications”, *in* : *GRID*, 2006 (cf. p. 29).

[27] Alexandru Iosup et al., “The Grid Workloads Archive”, *in* : *Future Generation Comp. Syst.* 24.7 (2008), p. 672–686 (cf. p. 6, 18).

[28] IPAC, *Two Micron All Sky Survey*, URL : `http://www.ipac.caltech.edu/2mass/` (cf. p. 62).

[29] Joseph C Jacob et al., “Montage : a grid portal and software toolkit for science-grade astronomical image mosaicking”, *in* : *International Journal of Computational Science and Engineering* 4.2 (2009), p. 73–87 (cf. p. 61).

[30] Konstantinos Kavoussanakis et al., “BonFIRE : the Clouds and Services Testbed”, *in* : *5th IEEE International Conference on Cloud Computing Technology and Science, Cloudcom*, 2013 (cf. p. 64).

[31] Katarzyna Keahey et al., “Virtual workspaces : Achieving quality of service and quality of life in the grid”, *in* : *Scientific Programming* 13.4 (2005), p. 265–275 (cf. p. 43).

[32] Attila Kertesz et al., “Enhancing federated cloud management with an integrated service monitoring approach”, *in* : *Journal of grid computing* 11.4 (2013), p. 699–720 (cf. p. 47, 49).

[33] Ekasit Kijsipongse et Sornthep Vannarat, “Autonomic resource provisioning in rocks clusters using Eucalyptus cloud computing”, *in* : *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, MEDES ’10, New York, NY, USA : ACM, 2010, p. 61–66, ISBN : 978-1-4503-0047-6, DOI : `10.1145/1936254.1936265`, (visité le 14/02/2012) (cf. p. 43).

[34] E Laure et al., *Programming the Grid with gLite*, rapp. tech. EGEE-TR-2006-001, Geneva : CERN, mar. 2006 (cf. p. 2).

[35] Guan Le, Ke Xu et Junde Song, “Dynamic Resource Provisioning and Scheduling with Deadline Constraint in Elastic Cloud”, *in* : IEEE, avr. 2013, p. 113–117, ISBN : 978-1-4673-6258-0, 978-0-7695-4972-9, DOI : 10.1109/ICSS.2013.18, (visité le 20/08/2013) (cf. p. 44).

[36] Philipp Leitner et al., “CloudScale : a novel middleware for building transparently scaling cloud applications”, *in* : *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ACM, 2012, p. 434–440 (cf. p. 45, 46, 49).

[37] Frédéric Magoulès, *Fundamentals of grid computing : theory, algorithms and technologies*, CRC Press, 2010 (cf. p. 78).

[38] Muthucumaru Maheswaran et al., “Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems”, *in* : *8th Heterogeneous Computing Workshop (HCW '99)*, IEEE Computer Society, avr. 1999, p. 30–44, DOI : 10.1109/HCW.1999.765094 (cf. p. 36).

[39] Paul Marshall, Kate Keahey et Timothy Freeman, “Elastic Site : Using Clouds to Elastically Extend Site Resources”, *in* : *10th IEEE/ACM Intnl Conf. on Cluster, Cloud and Grid Computing (CCGrid 2010)*, IEEE, 2010, p. 43–52 (cf. p. 42–44).

[40] Michael Maurer, Ivona Brandic et Rizos Sakellariou, “Adaptive resource configuration for Cloud infrastructure management”, *in* : *Future Generation Computer Systems* 29.*2* (2013), p. 472–487 (cf. p. 42).

[41] David Mendez, Mario Villamiazr et Harold Castro, “e-Clouds : Scientific Computing as a Service”, *in* : *Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on*, IEEE, 2013, p. 481–486 (cf. p. 48, 49).

[42] Étienne Michon, “Allocation dynamique sur plateforme IaaS avec maîtrise du compromis coûts/performance”, Réunion de Réseaux Grand Est (RGE), Université de Strasbourg, oct. 2011, URL : http://rge.u-strasbg.fr/reunions/strasbourg131011/index.html (cf. p. 113).

[43] Étienne Michon, Julien Gossa et Stéphane Genaud, “Free elasticity and free CPU power for scientific workloads on IaaS Clouds”, *in* : *18th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2012, p. 85–92 (cf. p. 85, 113).

[44] Étienne Michon et al., “Porting Grid applications to the Cloud with Schlouder”, *in* : *CloudCom-5th IEEE International Conference on Cloud Computing Technology and Science*, 2013 (cf. p. 113).

[45] Étienne Michon et al., “Schlouder : a broker for IaaS clouds”, *in* : *Future Generation Computer Systems (FGCS)* (2015) (cf. p. 113).

[46] Ruben S. Montero, Rafael Moreno-Vozmediano et Ignacio M. Llorente, "An elasticity model for High Throughput Computing clusters", *in* : *Journal of Parallel and Distributed Computing* 71.*6* (juin 2011), p. 750–757, ISSN : 0743-7315, DOI : 10.1016/j.jpdc.2010.05.005 (cf. p. 6, 47, 49).

[47] Manzur Murshed, Rajkumar Buyya et al., "Using the GridSim toolkit for enabling grid computing education", *in* : *International Conference on Communication Networks and Distributed Systems Modeling and Simulation*, 2002, p. 27–31 (cf. p. 86).

[48] Alberto Núñez et al., "iCanCloud : A Flexible and Scalable Cloud Infrastructure Simulator", *in* : *Journal of Grid Computing* 10.*1* (avr. 2012), p. 185–209, ISSN : 1570-7873, 1572-9184, DOI : 10.1007/s10723-012-9208-5 (cf. p. 50, 86).

[49] *Open Cloud Computing Interface*, URL : http://occi-wg.org/ (cf. p. 64).

[50] *OpenStack cloud software*, URL : http://www.openstack.org/ (cf. p. 64).

[51] Ana Oprescu et Thilo Kielmann, "Bag-of-Tasks Scheduling under Budget Constraints", *in* : *Proc. 2nd IEEE International Conference on cloud Computing Technology and Science (CloudCom 2010)* (nov. 2010) (cf. p. 44).

[52] Simon Ostermann et al., "GroudSim : an event-based simulation framework for computational grids and clouds", *in* : *Euro-Par 2010 Parallel Processing Workshops*, Springer, 2011, p. 305–313 (cf. p. 50, 86).

[53] Przemyslaw Pawluk et al., "Introducing STRATOS : A Cloud Broker Service", *in* : *5th International Conference on Cloud Computing (CLOUD)*, juin 2012, p. 891–898, DOI : 10.1109/CLOUD.2012.24 (cf. p. 47, 49).

[54] Julien Perez et al., "Multi-objective reinforcement learning for responsive grids", *in* : *Journal of Grid Computing* 8.*3* (2010), p. 473–492, DOI : 10.1007/s10723-010-9161-0 (cf. p. 43).

[55] Dana Petcu et al., "Experiences in building a mOSAIC of clouds", *in* : *Journal of Cloud Computing : Advances, Systems and Applications* 2.*1* (2013), p. 12 (cf. p. 45, 46, 49).

[56] Guillaume Pierre et al., "ConPaaS : an integrated runtime environment for elastic cloud applications", *in* : *Proceedings of the Workshop on Posters and Demos Track*, ACM, 2011, p. 5 (cf. p. 45, 46, 49).

[57] Alexander Pucher et al., "Using Trustworthy Simulation to Engineer Cloud Schedulers", *in* : *IEEE International Conference on Cloud Engineering (IC2E)*, mar. 2015 (cf. p. 86).

[58] *RightScale*, (accessed July 17th 2013), URL : http://www.rightscale.com (cf. p. 48, 49).

[59] *SCHIaaS : IaaS simulation upon SimGrid*, (accessed November 24th 2014), URL : `http://schiaas.gforge.inria.fr` (cf. p. 88).

[60] Upendra SHARMA et al., "A Cost-Aware Elasticity Provisioning System for the Cloud", *in* : *31st International Conference on Distributed Computing Systems (ICDCS)*, juin 2011, p. 559–570 (cf. p. 47, 49).

[61] João Nuno SILVA, Luís VEIGA et Paulo FERREIRA, "Heuristic for resources allocation on utility computing infrastructures", *in* : *6th International Workshop on Middleware for Grid Computing (MGC 2008)*, ACM, déc. 2008, DOI : `10.1145/1462704.1462713` (cf. p. 44).

[62] *SimSchlouder : Schlouder simulation upon SCHIaaS*, (accessed November 24th 2014), URL : `http://schiaas.gforge.inria.fr/simschlouder.html` (cf. p. 88).

[63] Hu SONG, Jing LI et Xinchun LIU, "IdleCached : An Idle Resource Cached Dynamic Scheduling Algorithm in Cloud Computing", *in* : *9th UIC and ATC*, sept. 2012, p. 912–917 (cf. p. 6, 48, 49).

[64] Anthony SULISTIO et al., "A toolkit for modelling and simulating data Grids : an extension to GridSim", *in* : *Concurrency and Computation : Practice and Experience* 20.*13* (2008), p. 1591–1609 (cf. p. 50).

[65] Enric TEJEDOR et al., "A cloud-unaware programming model for easy development of composite services", *in* : *Cloud Computing Technology and Science (CloudCom)*, IEEE, 2011, p. 375–382 (cf. p. 45, 49, 59).

[66] *The Amazon EC2 myth demystified*, (accessed March 6th 2015), URL : `https://www.quora.com/How-and-why-did-Amazon-get-into-the-cloud-computing-business` (cf. p. 3).

[67] Michael TIGHE et al., "DCSim : A data centre simulation tool for evaluating dynamic virtualized resource management", *in* : *Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualiztion management (svm)*, IEEE, 2012, p. 385–392 (cf. p. 50, 86).

[68] Christian VECCHIOLA, Xingchen CHU et Rajkumar BUYYA, "Aneka : a software platform for .NET-based cloud computing", *in* : *High Speed and Large Scale Scientific Computing* (2009), p. 267–295 (cf. p. 45, 46, 49, 59).

[69] Christian VECCHIOLA et al., "Deadline-driven provisioning of resources for scientific applications in hybrid clouds with Aneka", *in* : *Future Generation Computer Systems* 28.*1* (jan. 2012), p. 58–65, ISSN : 0167739x (cf. p. 42, 46, 49).

[70] Pedro VELHO et Arnaud LEGRAND, "Accuracy study and improvement of network simulation in the simgrid framework", *in* : *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, ICST (Institute for Computer Sciences, Social-Informatics et Telecommunications Engineering), 2009, p. 13 (cf. p. 86).

[71] Pedro VELHO et al., "On the validity of flow-level tcp network models for grid and cloud simulations", *in* : *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 23.*4* (2013), p. 23 (cf. p. 50, 51, 86).

[72] David VILLEGAS et al., "An Analysis of Provisioning and Allocation Policies for Infrastructure-as-a-Service Clouds", *in* : *CCGrid*, IEEE, 2012, p. 612–619 (cf. p. 6, 44, 48, 49).

[73] *Windows Azure*, (accessed July 17th 2013), URL : `http://www.windowsazure.com` (cf. p. 48, 49).

[74] Cheng-Zhong XU, Jia RAO et Xiangping BU, "URL : A unified reinforcement learning approach for autonomic cloud management", *in* : *Journal of Parallel and Distributed Computing* 72.*2* (2012), p. 95–105 (cf. p. 42).

[75] Sami YANGUI et al., "CompatibleOne : The Open Source Cloud Broker", *in* : *Journal of Grid Computing* (nov. 2013), ISSN : 1570-7873, 1572-9184, DOI : `10.1007/s10723-013-9285-0` (cf. p. 47, 49).

[76] Konstantinos ZACHOS et al., "Towards a framework for describing cloud service characteristics for use by chief information officers", *in* : IEEE, août 2011, p. 16–23, ISBN : 978-1-4577-0939-5, DOI : `10.1109/RESS.2011.6043932` (cf. p. 47).

Annexe A

Publications

Conférences internationales

Étienne Michon, Julien Gossa et Stéphane Genaud, “Free elasticity and free CPU power for scientific workloads on IaaS Clouds”, *in* : *18th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2012, p. 85–92

Étienne Michon et al., “Porting Grid applications to the Cloud with Schlouder”, *in* : *CloudCom-5th IEEE International Conference on Cloud Computing Technology and Science*, 2013

Conférences nationales

Étienne Michon, “Allocation dynamique sur plateforme IaaS avec maîtrise du compromis coûts/performance”, Réunion de Réseaux Grand Est (RGE), Université de Strasbourg, oct. 2011, url : `http://rge.u-strasbg.fr/reunions/strasbourg131011/index.html`

Soumissions en cours d’évaluation

Étienne Michon et al., “Schlouder : a broker for IaaS clouds”, *in* : *Future Generation Computer Systems (FGCS)* (2015)

Annexe B

Les algorithmes des stratégies

Table B.1 – Les stratégies de *provisioning* avec leurs paramètres pour l'Algorithme 1

strategy	*eligible*(v, j) returns *true*	*optimum*(C) returns $v \in C$ such that ...		comment
1VM4All	always	$v = v_0$		Slowest/Cheapest
1VMperJob	never	any		Fastest/Most expensive
1VMperJobPlus		any		
1VMperJobBest	if $q_v = \emptyset$	s_v is maximum		
1VMperJobWorst		s_v is minimum		
FirstFit		any		Regular bin packing strategies
BestFit	if $c(s_v - b_v) = c(s_v - b_v + r_j)$	$i_v - s_v$ is maximum		
WorstFit		$i_v - s_v$ is minimum		
EarliestFit		i_v is minimum		+ wait time optimization
RelaxFirstFit$x$	if $c(s_v - b_v) = c(s_v - b_v + r_j)$	any		Cost optimization
RelaxEarliestFit$x$	and $(i_v - t) < (x \times r_t)$	i_v is minimum		+ max wait time constraint
RelaxLastestFit$x$		i_v is maximum		+ min wait time constraint

Annexe C

Les différentes sorties de Schlouder

C.1 La sortie JSON

```
{
  info: {
    start_date: 1388458540,
    version: "7d57c1288e4de8ca2ebfa7d725d6bc71a0ceac7d",
    description: "hrs execution with ASAP",
  },
  nodes: [
  { // Description of a node
    instance_id: "/locations/uk-epcc/computes/28557",
    boot_time_prediction: 212,
    boot_time: 143,
    previous_boot_attempts: [],
    start_date: 1388458591,
    stop_date: 1388462137,
    instance_type: "m1.small",
    host: "bonfire-vm-3",
    cloud: "BonFIRE-uk-epcc",
    user: "emichon",
    jobs: [ // List of jobs executing on the node
      {
        id: 10127,
        name: "X001825CYC+LM_1.mgf",
        command_line: "./omssa.sh hrs ./omssa/X001825CYC+LM_1.mgf",
        SE_data: {
          runtime_prediction: 188,
          data_in: 20365749,
          data_out: 8960672
        },
        dependencies: [],
        provisioning_strategy: "Schlouder::Provisioning::Asap",
        scheduling_strategy: "Schlouder::Scheduling::Default",
        submission_date: 1388458581,
        start_date: 1388458735,
        scheduled_date: 1388458586,
        walltime_prediction: 174,
```

```
        walltime: 309,
        management_time: 1.0929436430000123,
        input_time: 5.1,
        input_files_size: {
          "X001825CYC+LM_1.mgf": 1462098,
          "mods.xml": 117497,
          "usermods.xml": 213,
          "run.sh": 4871,
          "human_DCpSP_ABU_20110615-blast.tgz": 14417326,
          "hrs_params.xml": 4386
        },
        input_size: 16006391,
        runtime: 293.7,
        output_time: 9.1,
        output_files_size: {
          "X001825CYC+LM_1.omx.gz": 15301832,
          "X001825CYC+LM_1.log": 744
        },
        output_size: 15302576,
      },
      { ... },
    ],
  },
  { ... },
  ],
}
```

Listing C.1 – JSON output format

C.2 La sortie en texte formatté

```
.----------------------------------------------------------------------------------------------.
| Cloud              | ID          | Name        | Boot Date | BootTime  | Jobs End Date | Shutdown |
+--------------------+-------------+-------------+-----------+-----------+---------------+----------+
| openstack-ec2-icps | i-000019be  | open-vm-1   | 02:40:59  | 58 (48)   | 02:48:59      | n/a      |
'--------------------+-------------+-------------+-----------+-----------+---------------+----------'
.----------------------------------------------------------------------------------------------------------------------------.
|                                  openstack-ec2-icps - i-000019be - open-vm-1 (02:40:59) + 58 (48)                          |
+-------+-------+-----+---------------------+------------+---------------------------+------------+---------------------------+
| ID    | State | Pos | Name                | Submission | Walltime (real / in batch) | Start Date | End Date (real / in batch) |
+-------+-------+-----+---------------------+------------+---------------------------+------------+---------------------------+
| 49076 | F     |     | X001825CYC+LM_1.mgf | 02:40:49   | (194 / 174)               | 02:41:58   | (02:45:12 / 02:44:52)      |
| 49106 | F     |     | X001840CYC+LM_2.mgf | 02:40:51   | (129 / 120)               | 02:45:12   | (02:47:21 / 02:47:12)      |
| 49125 | F     |     | X001851CYC+LM_2.mgf | 02:40:51   | (98 / 88)                 | 02:47:21   | (02:48:59 / 02:48:49)      |
'-------+-------+-----+---------------------+------------+---------------------------+------------+---------------------------'
```

Table des matières

UNIVERSITÉ DE STRASBOURG

Étienne Michon
Allocation dynamique sur cloud IaaS

msii ÉCOLE DOCTORALE

Résumé

Dans le contexte du cloud computing, l'IaaS fournit des ressources de calcul virtualisées à la demande suivant un modèle de paiement à l'utilisation. Du point de vue de l'utilisateur, ce nouveau paradigme fournit un stock inépuisable de ressources, qui peuvent être dynamiquement demandées et relâchées. L'IaaS permet l'exécution de calculs scientifiques sur un budget de fonctionnement plutôt que sur un investissement initial important. L'ordonnancement sur une telle plateforme élastique constitue un défi important dans le grand nombre de tâches et de ressources à prendre en compte pour réaliser le *provisioning*, mais également dans le grand nombre de plateformes et de modèles économiques disponibles. Nous avons abordé ce problème en concevant un système de courtage côté client capable (1) d'automatiser le *provisioning* en fonction d'une stratégie sélectionnée par l'utilisateur et (2) de simuler l'exécution afin de fournir à l'utilisateur une estimation des coûts et temps qu'impliquent les différentes stratégies. Son architecture ouverte permet de s'adapter à un grand nombre de fournisseur de cloud et de stratégies de *provisioning*. Des expérimentations à grande échelle ont été menées sur plusieurs plateformes de clouds avec des applications de type *bag-of-tasks* et *workflows*. Elles montrent la capacité de nos outils à exécuter différents types de workloads sur des plateformes variés et à simuler avec une grande précision ces exécutions.

Mots clés : IaaS, Courtier, Schlouder, Provisioning.

Abstract

In the field of cloud computing, IaaS provide virtualized on-demand computing resources on a pay-per-use model. From the user point of view, the cloud provides an inexhaustible supply of resources, which can be dynamically claimed and released. IaaS is especially useful to execute scientific computations using operating budget instead of using a big initial investment. Provisioning the resources depending on the workload is an important challenge, especially regarding the big number of jobs and resoruces to take into account, but also the large amount of available platforms and economic model. We advocate the need for brokers on the client-side with two main capabilities: (1) automate the provisioning depending on the strategy selected by the user and (2) able to simulate an execution in order to provide the user with an estimation of the costs and times of his workload's execution. Many provisioning strategies and cloud providers can be used in this broker thanks to its open architecture. Large scale experiments have been conducted on many cloud platforms and show our tool's ability to execute different kind of workloads on various platforms and to simulate these executions with high accuracy.

Keywords: IaaS, Cloud Broker, Schlouder, Provisioning.