RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Changement de contexte pour tâches virtualisées à l'échelle des grappes

Fabien Hermenier, Adrien Lèbre, Jean-Marc Menaud

ASCOLA, EMN-INRIA, LINA UMR 6241
44000 Nantes - France
firstname.lastname@emn.fr

Résumé
De nos jours, la gestion des ressources d'une grappe est effectuée en allouant des tranches de temps aux applications, spécifiées par les utilisateurs et de manière statique. Pour un utilisateur, soit les ressources demandées sont sur-estimées et la grappe est sous-utilisée, soit sous-dimensionnées et ses calculs sont dans la plupart des cas perdus. L'apparition de la virtualisation a apporté une certaine flexibilité quant à la gestion des applications et des ressources des grappes. Cependant, pour optimiser l'utilisation de ces ressource, et libérer les utilisateurs d'estimations hasardeuses, il devient nécessaire d'allouer dynamiquement les ressources en fonction des besoins réels des applications : Être capable de démarrer dynamiquement une application lorsqu'une ressource se libère ou la suspendre lorsque la ressource doit être ré-attribuée. En d'autres termes, être capable de développer un système comparable au changement de contexte sur les ordinateurs standards pour les applications s'exécutant sur une grappe. En s'appuyant sur la virtualisation, développer un tel mécanisme de manière générique devient envisageable. Dans cet article nous proposons une infrastructure offrant la notion de changement de contexte d'applications virtualisées appliquée aux grappes. Cette solution a permis de développer un ordonnanceur exécutant simultanément un maximum d'applications virtualisées. Nous montrons qu'une telle solution augmente le taux d'occupation de notre grappe et réduit le temps de traitement des applications.

Mots-clés : Changement de contexte, Virtualisation, Grappe

1. Introduction

Les grappes sont de nos jours largement utilisées pour un grand nombre de classes d'applications, du calcul scientifique au stockage en passant par les applications à haute disponibilité. En fonction de leurs tailles et objectifs, l'exploitation des grappes peut être spécifique (lorsque l'application à exécuter est dédiée à un calcul) ou générique (lorsqu'il existe une grande variété d'applications). Dans le cadre des grappes génériques, des gestionnaires de ressources (RMS - Resource Management System) sont utilisés. Ces derniers sont en charge de réserver les ressources en fonction des demandes des utilisateurs, demandes fondées sur des estimations de ressources pour l'exécution de leurs applications.
Plusieurs travaux ont été proposés pour apporter plus de flexibilité aux administrateurs et aux utilisateurs (puissance à la demande [1], déploiement d'environnements dédiés [2, 3], concept de bail [4], ...) mais l'ensemble des solutions continue de s'appuyer sur une réservation statique des ressources pour une tâche particulière (le terme de *job* est généralement utilisé).

Un tel mode de réservation ne permet pas de profiter au mieux de la puissance totale offerte par la grappe. Dans le meilleur des cas, le *job* sera terminé avant la fin du temps alloué et les ressources seront inutilisées pour le temps restant de la réservation. Dans le pire cas, la tâche n'aura pas terminé son exécution et le calcul sera interrompu avec potentiellement la perte des résultats. Si de telles réservations statiques de ressources se justifient dans un cadre spécifique tel que la reproductibilité d'expérimentations, la majeure partie des demandes porte sur l'exécution au plus tôt des calculs pendant une durée nécessaire à son exécution. Dans cet article nous proposons d'adresser ce problème en étendant nos précédents travaux sur la consolidation dynamique [5] par un mécanisme de changement de contexte appliqué aux grappes.

Traditionnellement, le changement de contexte est un mécanisme permettant de préempter une ressource, le CPU, affectée à une tâche et de la ré-allouer à une autre tâche. La tâche préemptée peut être en état suspendu ou réaffectée sur un autre processeur (dans le cas de machines SMP). Appliqué aux grappes, ce mécanisme doit permettre de préempter une ressource ou un ensemble de ressources (suspendre la tâche en cours d'exécution) et être capable de reprendre l'exécution d'une tâche précédemment suspendue ou de ré-affecter la tâche préemptée à un autre nœud.

Grâce à la généralisation de l'utilisation de la virtualisation il devient possible de proposer un mécanisme de changement de contexte générique permettant une allocation dynamique et partagée des ressources de la grappe. Plusieurs travaux ont étudié indépendamment ces aspects d'allocation dynamique [4, 6], de ré-allocation à la volée en utilisant la migration à chaud de machines virtuelles (VMs) [7, 8, 9] ou encore de prise d'empreinte mémoire permettant la suspension, la sauvegarde et la reprise d'exécution d'une VM [10]. Cependant, chacune de ces propositions a été réalisée dans un cas particulier. Nous proposons d'aborder dans cet article, le changement de contexte d'une tâche virtualisée (un *vjob*, i.e une tâche encapsulée dans une ou plusieurs VMs) en tant que brique fondamentale d'un système d'ordonnancement à l'échelle des grappes. Pour développer un tel mécanisme, il est primordial :

– De maintenir la cohérence de l'environnement des utilisateurs durant les phases de suspension et de reprise d'application distribuée ;
– De considérer les interdépendances entre les changements de contexte (si certaines opérations peuvent être exécutées en parallèle, d'autres doivent être sérialisées).
– d'évaluer le coût du changement de contexte. Dans notre environnement, migrer une VM peut prendre 26 secondes, reprendre une tâche suspendue sur disque peut prendre 3 minutes.

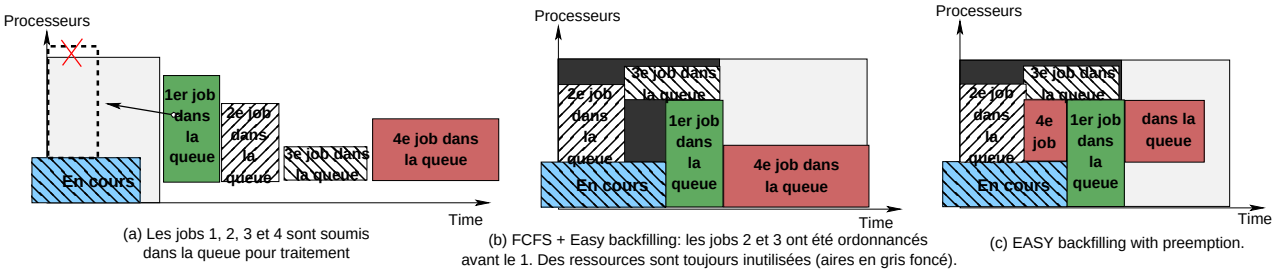
La solution présentée dans cet article répond à chacun de ces points et offre un cadre complet permettant de développer des ordonnanceurs spécifiques aux grappes. A titre d'exemple, nous proposons un algorithme d'ordonnancement permettant d'exécuter le plus grand nombre de *vjobs* tout en préservant la qualité de service des *vjobs* sélectionnés. Les autres sont suspendus.

Une évaluation en simulateur montre que notre solution est viable pour une gestion d'une grappe utilisant jusqu'à 200 nœuds et hébergeant 500 machines virtuelles. Une évaluation pratique sur une grappe de 11 nœuds complète cette évaluation et montre que le temps de complétion, dans le cas de 8 tâches virtualisées composées chacune de 9 machines virtuelles, peut être réduit de 40% et que le temps moyen de changement de contexte est de l'ordre de 70 secondes.

La suite de cet article est organisée comme suit : la section 2 rappelle les limitations des gestionnaires de ressources de type «batch», la section 3 décrit l'architecture de notre solution, la section 4 aborde le mécanisme de changement de contexte, de son principe à son optimisation. Nos expérimentations figurent en section 5. La section 6 présente les travaux apparentés et la section 7 conclue cet article en donnant quelques perspectives.

2. Fondements

Comme mentionné précédemment, la plupart des grappes sont gérées au travers de gestionnaires de ressources (RMS) assignant statiquement un ensemble de ressources à une tâche (un *job*) pour un durée déterminée. Le comportement classique d'un RMS consiste à gérer une liste de *job* sur le modèle du *First Come / First Serve*, en utilisant le mécanisme du "EASY backfilling" (Voir les Figures 1(a) et 1(b)). Les jobs arrivent les uns après les autres et sont ordonnancés en fonction de leur date d'arrivée, du temps d'exécution désiré et des ressources demandées. L'approche "EASY backfilling" limite les problèmes de fragmentation en traitant des tâches nécessitant moins de ressources tout en garantissant que la première tâche en attente sera traitée au plus tôt. Même s'il existe des stratégies plus sophistiquées telle que l'approche dîtes *Conservative*, ce type d'approche ne permet pas de gérer de manière optimale les ressources sans avoir recours à des mécanismes plus élaborés tel que la préemption (Figure 1 (c)).



(a) Les jobs 1, 2, 3 et 4 sont soumis dans la queue pour traitement

(b) FCFS + Easy backfilling: les jobs 2 et 3 ont été ordonnancés avant le 1. Des ressources sont toujours inutilisées (aires en gris foncé).

(c) EASY backfilling with preemption.

FIG. 1 – Limitation des approches par "Backfilling"

Les estimations de temps nécessaire aux réservations faites par les utilisateurs au moment de la soumission des jobs sont très imprécises[11] et seul un système de gestion de ressources dynamique peut permettre une gestion fine et efficace de la grappe. Généralement utilisées comme support pour permettre la tolérance aux fautes des applications, les solutions de «checkpointing» ont été suggérées comme mécanisme de préemption de ressources et permettre une gestion plus fine de ces dernières [12, 13]. Cependant ces méthodes sont fortement dépendantes soit de l'intergiciel soit des systèmes d'exploitations. Pour exemple, les systèmes à image unique tel que openMosix ou Kerrighed [14] ont intégré des stratégies avancées basées sur la préemption des ressources et la migration des calculs. Ces approches apportent une solution concrète pour une version d'un système d'exploitation particulier, mais ont des difficultés à gérer l'évolution de ces derniers. La virtualisation permet d'apporter plus de généricité et des solutions simples pour la migration, la suspension et la reprise des calculs.

2.1. Cycle de vie d'un *vjob*

Dans le cadre de la virtualisation, nous proposons la notion de job virtualisé. Un job virtualisé (*vjob*) peut s'exécuter dans une unique machine virtuelle (VM), ou être une application distribuée s'exécutant sur plusieurs VMs.

Lorsqu'un *vjob* est soumis, il évolue dans différents états décrit dans la figure 2. Il apparaît dans un premier temps dans l'état *Attente*. Lorsque les ressources deviennent disponibles pour l'exécution du *vjob*, le système exécute l'opération *lancement* sur toutes les VMs associées au *vjob* et passe l'état du *vjob* à *Exécution*. Lorsque les ressources viennent à manquer, le système peut exécuter une opération *suspension* sur un ou plusieurs *vjob*. Cette action écrit sur disque l'état mémoire du *vjob*, et passe le *vjob* en état *Endormi*. L'état *Prêt* combine les états *Endormi* et *Attente*. Lorsqu'un *vjob* est considéré terminé par son utilisateur, le *vjob* est arrêté et passe à l'état *Terminé*, entraînant l'application d'une opération *arrêt* sur l'ensemble des VMs composant

le *vjob*. L'opération *migration* permet d'affecter une VM en état `Exécution` à un nouveau nœud sans arrêt de services. Cet article ne traite pas le problème de transformation d'une application distribuée en *vjob*, ni de son instanciation et initialisation. Ces aspects pourront être traités par la suite en s'appuyant sur des travaux proposées par la communauté, par exemple Snowflock[15].

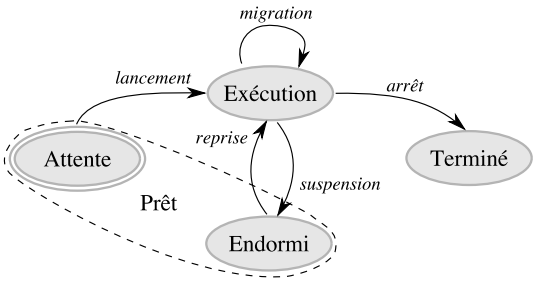
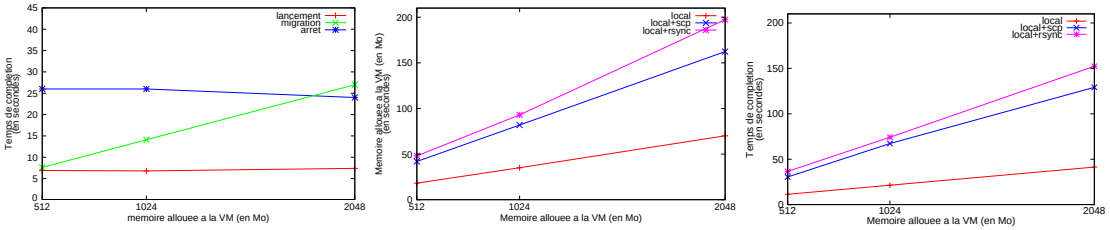


FIG. 2 – Le cycle de vie d'un *vjob*

2.2. Évaluation du changement de contexte pour VM

Cette section se consacre à l'évaluation des coûts liés au changement de contexte (des actions permettant de passer d'un état à l'autre). En effet, et pour exemple, dans la figure 1 (c), démarrer la tache numéro 4 pour un créneau de temps T (avant l'exécution du job 1) peut être inutile si le temps de suspension est égal voir supérieur au temps T. De même, si reprendre la tâche 4, après l'exécution du job 1 prend un temps supérieur au temps T, l'effet de la préemption sera contre-productif puisque la tâche 4 mettra plus de temps à s'exécuter. De plus, les actions de suspension et de reprise consomment du temps CPU du disque et de la bande passante. Nous évaluons également dans cette section l'impact des actions sur les *vjob* en cours d'exécution.



(a) Lancement/Migration/Arrêt (b) Suspension (c) Reprise

FIG. 3 – Temps d'exécution de chaque action selon les besoins en mémoire d'une VM

Nos évaluations ont été effectuées sur une grappe de 11 machines homogènes (2,1 GHz Intel Core 2 Duo : 1 CPU composé de 2 cores, 4 Go de RAM, interconnectées par un réseau giga ethernet). Chacun des nœuds exécute un linux 2.6.26-amd64 avec Xen 3.2 avec 512 Mo de RAM allouée au Domain-0. Trois serveurs NFS fournissent un disque virtuel et partagé à toutes les VMs (des debian lenny). Pour nos évaluations, nous avons désactivé le SMP afin d'évaluer de manière rigoureuse l'impacte de chacune des opérations. Nous avons exécuté 2 VMs sur un nœud de test. La première VM, servant de référence, dispose de 1 Go de mémoire, et réalise un calcul intensif, stressant ainsi le CPU. La seconde, appelée VM cible, sera la VM à suspendre. Elle réalise également un calcul intensif et la mémoire allouée varie de 512 Mo à 2 Go.

Les figures 3(a), 3(b) et 3(c) montrent le temps moyen d'exécution des opérations en fonction de la taille mémoire de la VM cible. Nous constatons que le temps nécessaire aux opérations run et stop n'est pas dépendant de la taille mémoire de la VM : Démarrer une VM requiert approximativement 6 et 25 secondes sont nécessaires pour l'arrêter. Pour les autres opérations,

le temps d'exécution est dépendant de la taille mémoire. De plus, la manière d'effectuer les opérations de suspension/reprise (local ou distant) impacte le temps d'exécution des ces opérations. Une suspension/reprise distante (plus précisément la sauvegarde est faite localement puis recopier sur un serveur distant via `scp` ou `rsync` et réciproquement pour la reprise) est naturellement bien plus coûteuse qu'une suspension/reprise locale.
Enfin, l'impact des opérations effectuées sur la VM cible sur la VM de référence est dépendant du temps nécessaire pour réaliser l'opération[1].
En s'appuyant sur les figures 3(b) et 3(c), nous pouvons observer que le temps nécessaire pour suspendre ou reprendre une VM et proportionnellement plus court quand la taille de celle-ci augmente. Ainsi, l'impact de décélération sur la VM référence est proportionnellement plus faible : il varie ente 1,5 pour une approche par copie distante (`scp` ou `rcp`) et 1.3 lorsque l'opération est réalisée uniquement en local. En d'autres termes, l'impact atteint un maximum de 50% pendant la durée du changement de contexte. L'ensemble des résultats présentés a été intégré dans le modèle de coût exploité par le système Entropy décrit ci-après et visant à proposer une gestion fine et dynamique de l'attribution des ressources.

3. Architecture

Le mécanisme de changement de contexte décrit précédemment à été inséré dans l'architecture d'Entropy [5], un système de consolidation dynamique permettant le placement de machines virtuelles dans une grappe. La notion de *vjob* a été introduite au système. Nous disposons donc de deux niveaux de granularité : la VM et le *vjob* qui est composé de une à plusieurs VMs.
Cette section présente l'architecture d'Entropy avant de s'attarder plus amplement sur un exemple de stratégie d'ordonnancement permettant d'exploiter au mieux les capacités d'une grappe en surcharge.

3.1. Architecture d'Entropy

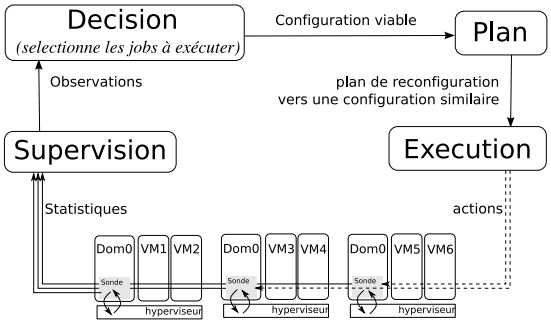


FIG. 4 – La boucle de contrôle d'Entropy

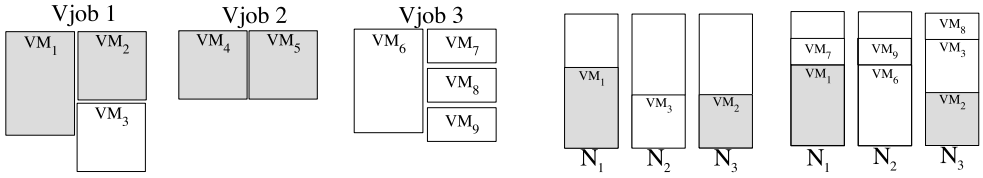
L'objectif d'Entropy consiste à gérer dynamiquement le placement de *vjob* dans une grappe. Plus précisément, Entropy permet de dynamiquement consolider une grappe virtualisée, i.e. d'exécuter un maximum de *vjobs* simultanément en allouant les ressources pour chaque VM à la volée. En cas de surcharge de la grappe, lorsque la demande en ressources est trop importante, le mécanisme de changement de contexte permet de suspendre les *vjobs* les moins prioritaires permettant ainsi d'affecter les ressources libérées aux *vjobs* les plus prioritaires.
Dans Entropy, d'un point de vue architectural, une grappe de machines virtuelles est composée de nœuds de travail chargés d'exécuter les *vjobs*, d'un ensemble de nœuds de stockage héber-

[1] Pour des raisons d'espace, nous n'avons pas fait apparaître les différents graphes présentant cette impact. Ils sont disponibles sur le site http ://entropy.gforge.inria.fr.

geant les images et les disques virtualisés des *vjobs* et des nœuds de service hébergeant les services de gestion tel que le système de supervision ou Entropy lui même.
La Figure 4 présente l'architecture globale. Entropy fonctionne sur le principe d'une boucle infinie qui (i) observe la consommation CPU et mémoire des VMs, (ii) réalise un nouveau placement en fonction de la consommation des ressources observées et de l'algorithme de décision mis en œuvre, (iii) définit un plan d'action pour passer de l'état courant au nouvel état en s'appuyant sur les mécanismes du changement de contexte (migration/suspension/reprise/lancement/arrêt) (iiii) et réalise le plan d'actions. En vue d'augmenter la réactivité du système, Entropy permet de réduire au maximum le temps d'exécution de la phase (iv) en déduisant un placement équivalent proposé en phase (ii) mais limitant le nombre de migrations, et de reprise distante. D'un point de vue technique, Entropy fonctionne avec le système de virtualisation Xen 3.2.1 [16] et le système de supervision Ganglia 3.0.7 [17].

3.2. Un algorithme d'ordonnancement pour la consolidation dynamique

L'algorithme développé dans le module de décision propose une nouvelle configuration viable où chacune des VMs en état *Exécution* dispose de suffisamment de ressources CPU et mémoire. Le problème consistant à trouver une configuration viable est comparable au problème NP-Dur *2-Dimensional Bin Packing* [18] où les deux dimensions sont la mémoire et la capacité du processeur.



(a) Une file de 3 vjobs.

(b) Toute les VMs du vjob 1 peuvent être placer

(c) Vjob3 peuvent être placé avec Vjob 1

FIG. 5 – Construction d'une solution avec 3 vjobs et 3 nœuds uni-processeur. Les VMs en gris requièrent un CPU.

Pour étudier notre mécanisme de changement de contexte, nous avons développé un algorithme d'ordonnancement de type *First Come/First Serve* (FCFS). Cet algorithme est chargé, en fonction des ressources consommées, de choisir et d'exécuter un maximum de *vjob* parmi ceux présent dans une liste (les plus prioritaires étant les premiers de la liste). Nous désignerons cet algorithme le *Running Job Selection Problem*(RJSP). La liste des *vjobs* à exécuter, triée par ordre de priorités, est réévaluée régulièrement. En fonction de la consommation des ressources des VMs, certain *vjobs* doivent être suspendus et d'autres réactivés. Pour déduire la liste de *vjobs* à exécuter, nous utilisons l'heuristique First Fit Decrease (FFD).
La figure 5 présente le déroulement de FFD dans le cas de 3 *vjobs* devant s'exécuter sur 3 nœuds de travail. Dans l'état initial, les *vjobs* 1 et 2 sont en cours d'exécution (avec 2 VMs demandant chacune 100% du CPU pour chacun des *vjobs*), alors que le *vjob* 3 est suspendu. A la première itération, FFD trouve une solution pour exécuter le *vjob* 1 sur la grappe (voir Figure 5(b)). A la seconde itération, l'algorithme ne peut placer le *vjob* 2, entraînant sa mise en suspension. Finalement le *vjob* 3 peut être placé, entraînant alors sa reprise (voir Figure 5(c)).

4. Le changement de contexte

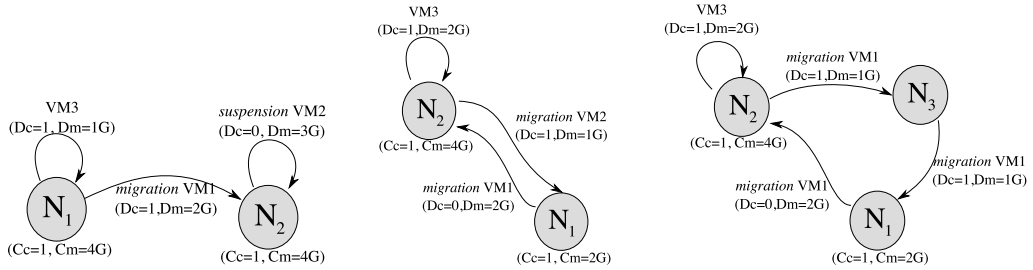
Le système de changement de contexte consiste à passer d'un état courant non-viable, à un nouvel état défini par le module de décision. Cette section se focalise sur la planification et

l'optimisation des étapes liées au changement de contexte.

4.1. Planification du changement de contexte

Passer d'un état courant à un état final ne peut se limiter à une suite sans relation d'actions sur les VMs. En effet, certaines de ces actions nécessitent de vérifier leurs *faisabilités*. Si la suspension et l'arrêt ne nécessitent pas de conditions particulières quant à leurs exécutions, la migration, la reprise ou le démarrage d'une VM ne peut être effectué que si les ressources nécessaires à la VM sont disponibles sur la machine devant héberger la VM.

Pour se faire, nous définissons un multi-graphe orienté de reconfiguration où les arrêtes représentent les actions (avec les demandes de mémoire d_m et de cpu d_p) et les nœuds, les machines et leurs capacités (mémoire c_m et cpu c_p). Pour exemple, deux actions sont à effectuer dans le graphe présenté dans la figure 6(a) : $suspension(\mathtt{VM}_2)$ et $migration(\mathtt{VM}_1)$. La $\mathtt{VM}_3$ quant à elle, continue son exécution sur le noeud 1. Comme constaté sur cet exemple, les actions doivent être séquentialisées. En effet, la migration de la $\mathtt{VM}_1$, ne peut être effectuée qu'après la suspension de la $\mathtt{VM}_2$. En plus du problème lié au séquencement des actions, il existe également des problèmes d'inter-dépendances entre les migrations.



(a) Une séquence d'actions (b) Migrations Inter-dependantes (c) Un pivot brise le cycle

FIG. 6 – Séquentialité et Inter-dépendances des actions

L'inter-dépendance des actions est présentée dans la figure 6(b) lorsqu'un cycle se produit, empêchant l'exécution de toutes actions. Ici, $\mathtt{VM}_1$ ne peut migrer de N_1 à N_2 que lorsque $\mathtt{VM}_2$ aura migré du N_2 au nœud N_1 et inversement. Ce problème est résolu en utilisant un nœud supplémentaire hébergeant temporairement une VM et brisant le cycle de dépendances. La figure 6(c) présente l'utilisation d'un pivot pour le problème présenté en figure 6(b).

Un plan de reconfiguration consiste à trouver une solution à l'ensemble des problèmes de séquentiallité ou d'inter-dépendances pour passer d'un état courant à un état viable. Pour que la reconfiguration de la grappe se fasse aussi rapidement que possible, il est nécessaire de trouver un plan de reconfiguration avec le maximum d'actions parallélisables. L'ensemble des actions qui peuvent être exécutés en parallèle pour l'étape E est regroupé dans un groupe d'actions. Le groupe d'actions de l'étape E+1 est composé des actions pouvant être exécutées en parallèle suite à l'exécution des actions du groupe E. Pour exemple, la figure 7 présente un graphe de reconfiguration de 4 actions. Le plan de reconfiguration associé est composé de 2 groupes d'actions : le premier groupe exécute en parallèle les actions $suspension(\mathtt{VM}_3)$ et $migration(\mathtt{VM}_1)$ puis second groupe exécute les actions $reprise(\mathtt{VM}_5)$ et $lancement(\mathtt{VM}_6)$.

Le dernier point traité par le plan de reconfiguration est celui de la suspension/reprise des VMs inter-dépendantes, i.e. appartenant à un même *vjob*. Plusieurs expérimentations ont montré qu'une application distribuée (*vjob*) peut être suspendue ou reprise, si les actions sont exécutées en même temps et dans le même ordre pendant une courte période [10]. Notre approche consiste à grouper les suspensions et les reprises dans un même groupe. Les différentes reprises

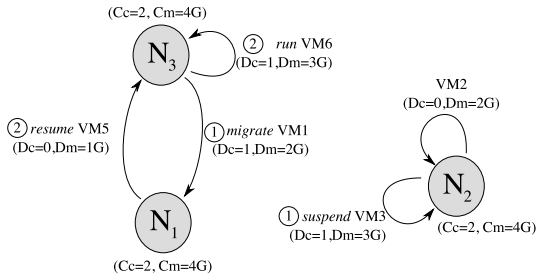


FIG. 7 – Plan de reconfiguration

étant naturellement regroupées dans le premier groupe, il suffit alors de regrouper les actions de reprise dans le dernier groupe concerné puis d'exécuter ces actions en pseudo-parallèle, en démarrant chaque action 1 seconde après la précédente.

4.2. Coût et optimisation du changement de contexte

Action	Coût
migration(vm_j)	$\mathcal{D}_m(v_j)$
lancement(vm_j)	*constant*
arrêt(vm_j)	*constant*
suspension(vm_j)	$\mathcal{D}_m(v_j)$
reprise(vm_j)	$\mathcal{D}_m(v_j)$ si *local*, $2 \times \mathcal{D}_m(v_j)$ sinon

TAB. 1 – Coût d'une action sur v_j. $\mathcal{D}_m(v_j)$ décrit la mémoire demandée par la VM v_j

Nous avons défini une fonction de coût pour nous permettre d'évaluer un plan de reconfiguration. Le coût d'un plan est l'ensemble du coût total de toutes les actions. le coût total d'une action est le coût local plus le coût des groupes précédents. Enfin le coût d'un groupe est le coût de l'action de plus grand coût.
Le coût local de chaque action est décrit dans la table 1, reprenant les évaluation de la section 2.2. Nous avons montré que le coût d'un arrêt ou démarrage d'un *vjob* est dépendant des applications s'exécutant à l'intérieur. Pour cette étude, nous considérerons le temps de ces actions comme une constante, égale à 0. Les autres opérations sont directement liées à la taille mémoire de la VM. Nous considérons que le coût de la reprise est double lorsque l'image mémoire est accèdée de manière distante plutôt que localement.

4.3. Optimisation du coût du changement de contexte

Réduire le coût du changement de contexte consiste à (i) exécuter les actions le plus tôt possible, (ii) maximiser le nombre d'éléments dans les groupes et (iii) limiter les reprises distantes et les migrations. Comme illustré dans la section 3.2, il existe plusieurs configurations viables pour un même état courant. Réduire le coût du changement de contexte, consiste à trouver une solution viable parmi l'ensemble des solutions dont le coût définie précédemment est le minimum. Pour se faire nous utilisons la Programmation Par Contraintes.
L'idée de la Programmation Par Contraintes (PPC) est de proposer des solutions à un problème en spécifiant seulement un ensemble de contraintes (conditions, propriétés) devant être satisfaites par toute solution acceptable pour le problème donné. Un problème de satisfaction de contraintes est alors modélisé sous la forme d'un ensemble de variables (au sens mathématique), d'un ensemble de domaines représentant les différentes valeurs que peuvent prendre les

variables et de contraintes formant un ensemble de relations entre celles-ci. Une solution représente alors une affectation de valeurs à chaque variable satisfaisant simultanément l'ensemble des contraintes. Un solveur de contraintes est un moteur permettant de générer ces solutions. Les utilisateurs n'ont alors qu'à décrire leurs variables, leurs domaines et leurs contraintes. Il s'agit donc d'un paradigme de programmation déclaratif. Entropy utilise la librairie Choco 1.2.04 [19]. Le modèle PPC utilisé dans Entropy est décrit dans [5].

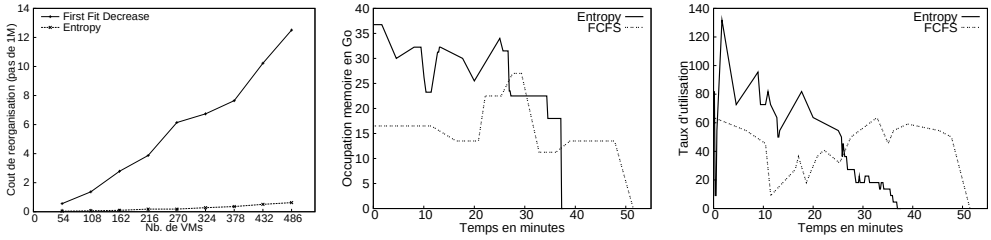
5. Evaluations

Nous détaillons dans cette section une première évaluation sur simulateur démontrant que la solution permet de gérer un grand nombre de nœuds. Une seconde évaluation sur une grappe de 11 nœuds montre l'intérêt du changement de contexte pour la de consolidation dynamique.

5.1. Expérimentation sur simulateur

Cette simulation comprend 200 nœuds de travail, disposant de 2 CPU et 4 GB de mémoire, et d'un nombre variable de VMs. Nous rejouons 81 traces réel de l'exécution du NAS Grid Benchmarks [20]. pour les cas d'étude W,A et B. Chaque *vjob* utilise 9 ou 18 VMs, l'état initial est choisi de manière aléatoire. Chacune des VMs utilise 256 MB, 512 MB, 1024 MB ou 2048 MB de mémoire, 100% du CPU lorsque la VM exécute un calcul, 5% sinon. L'ensemble de la simulation est exécutée sur un Macbook Intel Core Duo 1,83 GHz CPU avec 2 GB de RAM.

La figure 8(a) quantifie les temps de reconfiguration entre Entropy et un algorithme basé sur l'heuristique FFD. Le temps maximum de reconfiguration pour Entropy est de 40 secondes. Nous constatons que le coût de reconfiguration est réduite de 95% par Entropy. Ce résultat s'explique par le fait que dans les solutions traditionnelles basées sur FFD, l'algorithme choisit la première solution viable. Entropy en revanche recherche la meilleure solution viable avec un plan de reconfiguration le moins coûteux dans un temps maximal imparti.



(a) Coût de reconfiguration (b) Besoins mémoire (c) Besoins CPU

FIG. 8 – Évaluations

5.2. Expérimentations sur grappe

L'architecture d'expérimentation est identique à celle présentée dans la section 2.2. L'expérimentation consiste à exécuter 8 *vjobs*, composés chacun de 9 VMs. L'ensemble des *vjobs* est soumis à Entropy au même moment et dans un ordre précis. Chaque *vjob* exécute une instance de NAS Grid Benchmarks. Chaque VM demandant 100% du CPU exécute un calcul, 5% dans le cas contraire, et une quantité de mémoire fixe, mais propre à chaque VM (de 512 MB à 2048 MB). Le *vjob* signale sa fin d'exécution à Entropy, entraînant l'exécution de l'action stop.

Pour un même jeu de test, nous avons comparé Entropy à une allocation statique gérée par un algorithme de type FCFS. Les figures 8(b) et 8(c) présentent l'utilisation des ressources des VMs en fonction de l'approche. Nous remarquons que le taux moyen de l'utilisation des ressources est plus importante dans Entropy jusqu'à la $25^{\text{ième}}$ minute puis le nombre de *vjob* diminuant

décroît. À 2 minutes 10, la grappe devient surchargée : l'ensemble des *vjobs* en cours d'exécution demande 29 CPUs alors que la grappe n'en dispose que de 22. À ce moment là, l'algorithme RJSP demande la suspension des *vjobs* les moins prioritaires (les derniers arrivés) pour revenir à une situation satisfaisante où chaque VM dispose d'une quantité adéquate de ressources.
Pour conclure, nous constatons que l'utilisation d'Entropy permet de réduire significativement le temps de complétion des *vjobs*. Pour l'exécution complète d'un même jeu de test, FCFS nécessite 250 minutes de temps d'exécution lorsque Entropy n'en demande que 150 minutes. Grâce à notre système de consolidation dynamique avec suspension et reprise des *vjob* nous pouvons réduire de près de 40% les temps d'exécution.

6. Travaux apparentés

Les travaux apparentés les plus proches sont ceux de Sotomayor *et al.* [21, 4] qui proposent avec Haizea le concept de bail pour une réservation de ressources. Les utilisateurs demandent des ressources pour une durée prédéterminée et indiquent si le bail est de type "best-effort" ou non. Suivant le type, les VM associées au bail peuvent migrer ou être suspendue par le système. Nous retrouvons dans ces travaux le concept de suspension de *vjobs* mais les ressources associées au bail ne peuvent être ré-allouée dynamiquement, contrairement à notre solution où la notion de plan de reconfiguration n'est pas adressée.
La notion de plan de reconfiguration est en partie adressée dans les travaux de Grit *et al.* [22, 23, 24]. Ces travaux permettent de placer dynamiquement des VM dans une grappe. Ils démontrent la nécessité de dissocier le processus de décision de celui de planification. En revanche, ils ne considèrent pas le mécanisme de suspension et reprise.
Enfin, des travaux se sont focalisés sur l'intérêt de la consolidation dynamique dans les grappes en vue d'améliorer leur taux d'utilisation. Par exemple, Khanna *et al.* [25] et Bobroff *et al.* [8] proposent un algorithme permettant de minimiser les ressources inutilisées mais ils ne traitent pas les problèmes des cycles d'inter-dépendance et de séquentialité dans le plan de reconfiguration. Les travaux de Wood *et al.* [26] portent sur la consolidation de machines virtuelles en maximisant le partage de la mémoire vive entre VM. Ils adressent les problèmes de séquentialité mais pas celui des cycles d'inter-dépendances. Pour l'ensemble de ces derniers travaux, les solutions proposées adressent le cas des grappes sous-chargées sans disposer de mécanisme de suspensions/reprise nécessaire à la gestion des grappes surchargées.
En se focalisant sur un point particulier, les travaux apparentés présentent des solutions pour gérer de manière plus flexible les ressources disponibles sur une grappe mais ne proposent pas de mécanismes génériques pour le traitement dynamique des ressources.

7. Conclusion

En s'appuyant sur les estimations des utilisateurs, un grand nombre de gestionnaire de ressources pour grappe reposent sur une réservation statique des ressources. Plusieurs travaux récents ont montré qu'une gestion dynamique des ressources réellement utilisées permettait d'améliorer la productivité des grappes. Si ces travaux différent par les algorithmes d'allocation utilisés, tous utilisent des mécanismes similaires mais implémentés de manière ad-hoc. Dans cet article, nous avons proposé un changement de contexte dédié aux environnements virtualisés, permettant de développer des stratégies avancées et génériques d'ordonnancement de tâches virtualisées (*vjobs*). L'intégration dans Entropy, nous permet d'offrir un environnement complet de gestion de grappes virtualisées, gérant intégralement le cycle de vie d'un *vjobs*, via les opérations de migration, suspension/reprise et arrêt/démarrage. L'ensemble est optimisé

par l'utilisation d'un solveur de contraintes permettant de réduire la durée du changement de contexte. La mise en œuvre d'un premier ordonnanceur basique dans le module de décision montre que le temps de complétion des *vjobs* a été réduit de 40% comparé à l'approche *First Come/First Serve* couramment utilisée.
Nos prochains travaux porteront sur le développement d'un langage dédié permettant la conception d'ordonnanceurs spécifiques aux grappes et aux *vjobs*. Enfin, nous proposerons un mécanisme de suspension en RAM, pour reduire l'impact de décélération et les temps de reprise.

Bibliographie

1. J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle, "Dynamic virtual clusters in a grid site manager," in *HPDC '03 : Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*. Washington, DC, USA : IEEE Computer Society, 2003, p. 90.
2. G. Vallee, T. Naughton, and S. L. Scott, "System management software for virtual environments," in *CF '07 : Proceedings of the 4th international conference on Computing frontiers*. New York, NY, USA : ACM, 2007, pp. 153–160.
3. R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa, "Grid'5000 : a large scale and highly reconfigurable experimental grid testbed." *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, Nov. 2006.
4. B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *HPDC '08 : Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA : ACM, 2008, pp. 87–96.
5. F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy : a consolidation manager for clusters," in *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA : ACM, 2009, pp. 41–50.
6. N. Fallenbeck, H.-J. Picht, M. Smith, and B. Freisleben, "Xen and the art of cluster scheduling," in *VTDC '06 : Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*. Washington, DC, USA : IEEE Computer Society, 2006, p. 4.
7. P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen, "Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure," *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pp. 5–14, 2006.
8. N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing SLA violations," *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pp. 119–128, May 2007.
9. A. Verma, P. Ahuja, and A. Neogi, "Power-aware dynamic placement of hpc applications," in *ICS '08 : Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA : ACM, 2008, pp. 175–184.
10. W. Emeneker and D. Stanzione, "Increasing reliability through dynamic virtual clustering," in *High Availability and Performance Computing Workshop*, 2006.
11. A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529–543, 2001.
12. D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice : the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

13. P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics : Conference Series*, vol. 46, pp. 494–499, 2006. [Online]. Available : http://stacks.iop.org/1742-6596/46/494
14. R. Lottiaux, P. Gallard, G. Vallée, C. Morin, and B. Boissinot, "Openmosix, openssi and kerrighed : a comparative study," in *CCGRID 05 : Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid05) - Volume 2*. Washington, DC, USA : IEEE Computer Society, 2005, pp. 1016–1023, iSBN 0-7803-9074-1.
15. H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock : rapid virtual machine cloning for cloud computing," in *EuroSys '09 : Proceedings of the fourth ACM european conference on Computer systems*. New York, NY, USA : ACM, 2009, pp. 1–12.
16. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, USA : ACM Press, Oct. 2003, pp. 164–177.
17. M. Massie, "The ganglia distributed monitoring system : design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, July 2004. [Online]. Available : http://dx.doi.org/10.1016/j.parco.2004.04.001
18. P. Shaw, "A constraint for bin packing," in *Principles and Practice of Constraint Programming (CP'04)*, ser. Lecture Notes in Computer Science, vol. 3258. Springer, 2004, pp. 648–662.
19. N. Jussien, G. Rochart, and X. Lorca, "The CHOCO constraint programming solver," in *CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, Paris, France, Jun. 2008. [Online]. Available : http://www.emn.fr/jussien/publications/jussien-OSSICP08.pdf
20. M. Frumkin and R. F. V. der Wijngaart, "NAS grid benchmarks : A tool for grid space exploration," *Cluster Computing*, vol. 5, no. 3, pp. 247–255, 2002.
21. B. Sotomayor, R. M. Montero, I. M. Llorente, and I. Foster, "Capacity leasing in cloud systems using the opennebula engine," in *Cloud Computing and Applications 2008 (CCA08)*, 2008.
22. L. Grit, D. Irwin, A. Yumerefendi, and J. Chase, "Virtual machine hosting for networked clusters : Building the foundations for "autonomic" orchestration," in *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, Nov. 2006, pp. 1–8.
23. L. Grit, D. Irwin, V. Marupadi, and P. Shivam, "Harnessing virtual machine resource control for job management," in *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, Nov. 2007.
24. D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum, "Sharing networked resources with brokered leases," in *ATEC '06 : Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2006, pp. 18–18.
25. G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pp. 373–381, 2006.
26. T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies : exploiting page sharing for smart colocation in virtualized data centers," in *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA : ACM, 2009, pp. 31–40.