Compas'2018 : Parallélisme/ Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Ordonnancement multi-objectifs de workflows dans un Cloud privé

Emile Cadorel, Hélène Coullon, Jean-Marc Menaud

IMT Atlantique, Inria, LS2N, UBL, F-44307 Nantes, France
emile.cadorel@imt-atlantique.fr, helene.coullon@imt-atlantique.fr,
jean-marc.menaud@imt-atlantique.fr

Résumé
Cet article adresse le problème d'ordonnancement de workflows scientifiques dans un environnement de Cloud privé. L'ordonnancement dans ce type d'environnement est un problème d'optimisation multi-objectifs difficile. Généralement, les travaux menés sur l'ordonnancement de workflows se concentrent sur l'ordonnancement dans un Cloud public, et ne considèrent donc pas les différentes limitations de l'infrastructure. Cet article propose d'une part une modélisation de l'infrastructure et du problème d'ordonnancement posé en prenant en compte le nombre fini de ressources disponibles, et d'autre part une heuristique permettant de résoudre l'ordonnancement de workflows tout en essayant de réduire le nombre de ressources utilisées (e.g. réduction de la consommation énergétique). Les résultats préliminaires obtenus sur cette contribution sont prometteurs et ouvrent la porte à de nombreuses perspectives.

1. Introduction

Le Cloud [11], en offrant une grande souplesse d'utilisation pour un coût relativement faible, connaît un succès indéniable depuis une dizaine d'années. La notion de ressources *à-la-demande* et de paiement à l'utilisation a permis l'émergence de nouveaux services applicatifs. Cependant, son utilisation dans le domaine de l'hébergement de calculs scientifiques, et plus généralement de *workflows* [7], reste un domaine de recherche actif levant un certain nombre de verrous. Dans un contexte IaaS privé, nous proposons dans cet article, un modèle permettant l'ordonnancement des exécutions de workflows. Les workflows que nous considérons dans cet article sont des graphes de tâches interdépendantes nécessitant différents types de ressources (matérielles ou logicielles).

Des solutions pour l'ordonnancement de workflows, dans des infrastructures de type Cloud existent, mais les modèles se positionnent généralement dans une infrastructure Cloud de type public, ce qui rend le problème relativement différent. En effet, les solutions d'ordonnancement dans le Cloud public considèrent les ressources disponibles comme étant illimitées, et se concentrent généralement sur le respect d'un budget défini par l'utilisateur. Dans notre cas, l'ordonnancement se fait sur un Cloud privé avec un nombre de machines physiques limité sans notion de coût pour l'utilisateur final. Par ailleurs, les solutions utilisées dans un environnement de grille ne sont pas non plus utilisables car les workflows que nous souhaitons ordonnancer ici présentent des tâches hautements hétérogènes, s'exécutant généralement sur

des systèmes d'exploitations très différents (Windows, Linux, ...), ce qui rend la mise en place d'un système de virtualisation obligatoire .

Notre contribution porte sur la définition d'un modèle d'ordonnancement de workflows dans un Cloud privé de type IaaS, avec deux objectifs : la maximisation du nombre de workflows ordonnancés, c'est à dire qui respectent leur *deadline* définie par l'utilisateur ; et la minimisation du nombre de machines physiques utilisées pour l'ordonnancement, afin de diminuer le coût de fonctionnement (e.g. consommation électrique) et/ou de libérer des ressources pour de futurs applications. Nous commencerons par présenter un état de l'art sur l'ordonnancement de workflow, après quoi nous détaillerons notre modèlisation. Nous présenterons ensuite une première implémentation d'un algorithme permettant la résolution du problème posé et des objectifs fixés. Enfin, nous présenterons des résultats préliminaires et conclurons sur ce travail.

2. État de l'art

L'ordonnancement de workflows dans le Cloud avec pour objectif la minimisation des coûts a été beaucoup étudié [2], et les algorithmes proposés sont généralement destinés au Cloud de type public, ou hybride. Les auteurs de [13] considèrent l'arrivée de workflows au cours du temps. Les workflows sont ordonnés par priorité, et possèdent une *deadline*. L'objectif de la contribution est de maximiser la somme des priorités des workflows qu'il est possible de placer afin que les workflow les plus prioritaires respectent leur *deadline*, ainsi que le budget défini par l'utilisateur. Le modèle prend en compte le temps nécessaire pour l'allumage des *Machines Virtuelles* (VM) approvisionnées. Néanmoins, ce modèle ne prend pas en compte les temps de transferts des données entre les tâches des workflows. Pourtant, ces temps de transfert ne sont généralement pas négligeables et sont donc une composante essentielle au problème.

Dans [12, 1], les auteurs minimisent le coût de l'ordonnancement tout en respectant les *deadlines* des workflows. Le modèle proposé, à l'inverse du premier, prend en compte les temps de transfert des données entre les tâches. Toutefois, la bande passante allouée entre VM est présumée homogène. La deuxième limitation des modèles proposés dans ces deux articles, est qu'une VM ne peut héberger qu'une seule tâche à la fois. De plus, tous les articles cités précédemment considèrent que le Cloud est en mesure de répondre à la demande quelque soit la charge. Cette hypothèse n'est pas valable dans un Cloud de type privé, où les machines physiques sont limitées en nombre et en capacité.

L'article [3] propose un algorithme pour l'ordonnancement dans un Cloud hybride - où le calcul est partagé entre un Cloud privé et un Cloud public. Le modèle proposé prend en compte les différents éléments liés à l'exécution des tâches : le nombre d'instructions, les capacités des ressources et les communications entre les tâches. En revanche, ce modèle ne s'intéresse pas au placement des ressources virtuelles sur les machines physiques. Les machines physiques sont donc considérées comme disponibles à tout instant. Ceci empêche de contrôler le nombre réel de machines physiques allumées, mais aussi de considérer le temps pris par la mise en service des ressources. Enfin, une unique tâche peut être exécutée par VM. Cette supposition forte rend le modèle peu réaliste, et peut conduire à un placement trop pessimiste.

Les auteurs de [10] proposent un modèle pour l'ordonnancement de workflows de tâches, en utilisant des *leviers verts* pour réduire la consommation énergétique globale. Cette contribution n'est toutefois pas adaptée à un environnement de type Cloud. En effet, les ressources physiques sont directement considérées, sans couche de virtualisation. L'infrastructure considérée se rapproche donc d'un cluster (ou serveur). La virtualisation apporte une grande souplesse

notamment pour gérer des besoins logiciels hétérogènes, pour optimiser le placement et la migration de tâches, mais également pour garantir une isolation de certaines applications.

De nombreuses contributions ont abordé le problème de placement de VM [6, 9, 8] qui est connu sous le nom de *bin-packing*. Le problème modélisé et résolu dans cet article est double. Il s'agit à la fois d'un problème d'ordonnancement de workflows mais également d'un problème d'allocation de VM sur un ensemble de ressources (puisqu'un Cloud privé est considéré). Ces deux problèmes sont connus pour être *NP-difficile* [5, 14].

3. Modélisation

Le problème que nous voulons modéliser est l'ordonnancement de workflows dans un environnement de Cloud privé. Ce problème est un double *bin-packing*, où il faut placer les tâches sur des VM en respectant les contraintes de capacité, et placer les VM sur des machines physiques, mais également un problème d'ordonnancement car il faut respecter les *deadlines* des workflows. Le problème d'ordonnancement de workflows possède également un ensemble de contraintes temporelles, qu'il faut respecter pour être certain que les tâches s'exécutent dans l'ordre et possèdent bien les données dont elles ont besoin. Les VM sont dimensionnées grâce à un ensemble de *templates*

3.1. Environnement d'exécution

On cherche à placer un ensemble de workflows de type *data flow*. On note $\mathcal{J}$, l'ensemble des tâches à traiter, qui composent les workflows. Chaque workflow est représenté par un graphe orienté acyclique de tâches $G = (T, D)$, où $T \subset \mathcal{J}$, et D est l'ensemble des dépendances de données entre les tâches. Chaque $d \in \mathcal{D}$ possède un poids représentant la taille des données à transférer d'une tâche à l'autre. Des travaux récents ont cherché à déterminer le nombre d'instructions composant une tâche [4]. Nous supposons de notre côté que cette information nous est donnée. Les tâches possèdent des contraintes que l'on peut distinguer en deux catégories : les contraintes matérielles qui sont dénombrables, et les contraintes logicielles (OS, langage, etc.).

L'environnement dans lequel nous voulons ordonnancer l'exécution des workflows, est un ensemble de clusters de machines, dans lesquels nous avons la possibilité de déployer différentes VM. On note $\mathcal{V}$ l'ensemble des VM utilisées. Ces VM sont approvisionnées selon les besoins grâce à un ensemble de *templates* prédéfinis. On note $\mathcal{N}$ l'ensemble des nœuds (machines physiques) de calcul. Deux nœuds peuvent être dans des clusters distants. On note $bw^{\mathcal{N}}_{n,m}$ la bande passante entre n et $m \in \mathcal{N}$. On note $speed^{\mathcal{N}}_{n}$ la vitesse du nœud $n \in \mathcal{N}$, en nombre d'instructions par cœur (soit sa vitesse totale divisée par son nombre de cœurs). Chaque tâche est considérée comme séquentielle, chaque tâche va donc occuper une seule unité de calcul. Une VM v possède un nombre de cœurs $core^{\mathcal{V}}_{v}$ connu, qui nous est donné par son *template*. Nous ne faisons aucun sur-approvisionnement, par conséquent un cœur d'un nœud est reservé pour un VCPU. Une VM est donc capable d'exécuter au mieux $core^{\mathcal{V}}_{v}$ tâches en parallèle.

3.2. Modélisation des contraintes

Pour chaque nœud $n \in \mathcal{N}$ on associe, à chaque instant t, le vecteur $H_{tn} =< h_{tnv}, \ldots, h_{tn|\mathcal{V}|} >$, tel que $h_{tnv} = 1$ si et seulement si la VM v est allumée sur n à l'instant t, et $h_{tnv} = 0$ sinon. De même, on associe pour chaque $v \in \mathcal{V}$, à chaque instant t, le vecteur $E_{tv} =< e_{tvj}, \ldots, e_{tv|\mathcal{J}|} >$, tel que $e_{tvj} = 1$ si et seulement si $j \in \mathcal{J}$ s'exécute sur v à l'instant t, et $e_{tvj} = 0$ sinon.

$\mathcal{C}$ est l'ensemble des types de capacité (e.g. mémoire, cpu, etc.), où pour chaque type de capacité k on associe trois vecteurs :

- $\mathcal{C}_k^{\mathcal{N}}$, de taille $|\mathcal{N}|$, qui représente les capacités en ressource k de chacun des nœuds $n \in \mathcal{N}$, et on note $\mathcal{C}_k^{\mathcal{N}}(n)$, la quantité en ressource k disponible sur n.
- $\mathcal{C}_k^{\mathcal{V}}$, de taille $|\mathcal{V}|$, qui représente les besoins en ressource k de chaque VM $v \in \mathcal{V}$, on note $\mathcal{C}_k^{\mathcal{V}}(v)$, la quantité de ressource k sur v.
- $\mathcal{C}_k^{\mathcal{J}}$, de taille $|\mathcal{J}|$, qui représente les besoins en ressource k, pour chaque tâche $j \in \mathcal{J}$.

Pour chaque capacité $k \in \mathcal{C}$, on définit les deux contraintes suivantes :

$$\mathcal{C}_k^{\mathcal{V}} \cdot H_{tn} \leq \mathcal{C}_k^{\mathcal{N}}(n) \ \forall n \in \mathcal{N}, \forall t \in \mathbb{N} \tag{1}$$

$$\mathcal{C}_k^{\mathcal{J}} \cdot E_{tv} \leq \mathcal{C}_k^{\mathcal{V}}(v) \ \forall v \in \mathcal{V}, \forall t \in \mathbb{N} \tag{2}$$

On note $\mathcal{S}$, l'ensemble des besoins *logiciels* de tout type (e.g. OS, bibliothèques, langage, etc.). Pour chaque besoin logiciel $s \in \mathcal{S}$, on associe deux vecteurs :

- $\mathcal{S}_s^{\mathcal{V}}$ qui représente la possession du logiciel s pour chaque $v \in \mathcal{V}$, tel que $\mathcal{S}_s^{\mathcal{N}}(v) = 1$ si et seulement si v possède le logiciel s, et 0 sinon.
- $\mathcal{S}_s^{\mathcal{J}}$ l'ensemble des besoins de la capacité logicielle s pour chaque tâche $j \in \mathcal{J}$, tel que $\mathcal{S}_s^{\mathcal{J}}(j) = 1$, si j a besoin du logiciel s, et 0 sinon.

Pour tout besoin logiciel $s \in \mathcal{S}$, on ajoute la contrainte suivante au modèle :

$$\forall t \in \mathbb{N}, \forall v \in \mathcal{V}, \forall j \in \mathcal{J} \ \ (e_{tvj} = 1) \Rightarrow ((\mathcal{S}_s^{\mathcal{V}}(v) = \mathcal{S}_s^{\mathcal{J}}(j)) \vee (\mathcal{S}_s^{\mathcal{J}}(j) = 0)) \tag{3}$$

Contraintes de dépendances temporelles. On note $speed_{tv}^{\mathcal{V}}$ la vitesse de $v \in \mathcal{V}$ à l'instant t, et $speed_{tv}^{\mathcal{V}} = speed_n^{\mathcal{N}}$ si $h_{tnv} = 1$ (rappelons que $speed_n^{\mathcal{N}}$ est en nombre d'instructions par cœur). Les VM sont approvisionnées à la demande pour répondre aux besoins des différentes tâches. On note $start_v^{\mathcal{V}}$ l'instant où $v \in \mathcal{V}$ initie son allumage. L'allumage d'une VM requiert un temps d'allumage. On note $boot_v^{\mathcal{V}}$ le nombre d'instructions à effectuer avant que $v \in \mathcal{V}$ soit opérationnelle et utilisable. Les VM ne sont utilisables qu'une fois que leurs instructions de démarrage sont terminées. On note $ready_v^{\mathcal{V}} = start_v^{\mathcal{V}} + \frac{boot_v^{\mathcal{V}}}{speed^{\mathcal{V}}_{start_v^{\mathcal{V}} v}}$ l'instant où $v \in \mathcal{V}$ est utilisable. On considère que les VM s'exécutent sans limitation de bande passante sur la machine hôte (pour simplifier le modèle), et par conséquent, on note $\forall t \in \mathbb{N},\ bw_{t,v,u}^{\mathcal{V}} = bw_{n,m}^{\mathcal{N}}$ si $h_{tvn} = 1$ et que $h_{tum} = 1$.

On note W_i le poids d'une tâche $i \in \mathcal{J}$ en nombre d'instructions. On note $start_i^{\mathcal{J}}$ l'instant où la tâche $i \in \mathcal{J}$ démarre. On considère qu'une tâche ne peut pas changer d'emplacement (pas de migration), par conséquent, on note indépendemment de l'instant t l'emplacement de la tâche $i \in \mathcal{J}$ sur la VM v, $loc_i^{\mathcal{J}} = v$, si $e_{start_i^{\mathcal{J}} vi} = 1$. Une tâche ne peut pas être exécutée avant que ses prédécesseurs ne soient terminés, par conséquent, on note $pred_i^{\mathcal{J}} \subset T \subset \mathcal{J}$, l'ensemble des prédécesseurs de la tâche $i \in T$, tel que $\forall j \in pred_i^{\mathcal{J}},\ \exists\ d_{ji} \in D$, où d_{ji} représente la taille des données à transférer de j à i. On note $exec_i^{\mathcal{J}}$ le temps d'exécution de $i \in \mathcal{J}$, tel que :

$$\forall i \in \mathcal{J},\ exec_i^{\mathcal{J}} = \frac{W_i}{speed^{\mathcal{V}}_{loc_i^{\mathcal{J}}}} + \sum_{j \in pred_i^{\mathcal{J}}} \frac{d_{ji}}{bw^{\mathcal{V}}_{loc_j^{\mathcal{J}}, loc_i^{\mathcal{J}}}} \tag{4}$$

On ajoute les contraintes suivantes au modèle :

$$\forall i \in \mathcal{J},\ start_i^{\mathcal{J}} \geq \max_{j \in pred_i^{\mathcal{J}}} \left(start_j^{\mathcal{J}} + exec_j^{\mathcal{J}} \right) \tag{5}$$

$$\forall i \in \mathcal{J},\ start_i^{\mathcal{J}} \geq ready_{loc_i^{\mathcal{J}}}^{\mathcal{V}} \tag{6}$$

Enfin, chaque workflow $G = (T, D)$ possède une *deadline* à respecter par la contrainte suivante :

$$\forall\, j \in T \left(start_j^{\mathcal{J}} + exec_j^{\mathcal{J}} \right) \leq \mathit{deadline} \tag{7}$$

3.3. Objectifs

Le modèle que nous proposons est multi-objectifs, nous cherchons à maximiser le nombre de workflows placés (Equation (8)), et à minimiser le nombre de nœuds utilisés (Equation (9)).

$$maximize(\sum_{j \in \mathcal{J}} x), x = \begin{cases} 1 & \text{si } \exists\, t \in \mathbb{N}, v \in \mathcal{V},\ e_{tvj} = 1 \\ 0 & \text{sinon} \end{cases} \tag{8}$$

$$minimize(\sum_{n \in \mathcal{N}} x), x = \begin{cases} 1 & \text{si } \sum_{t \in \mathbb{N}} (H_{tn}) \neq 0 \\ 0 & \text{sinon} \end{cases} \tag{9}$$

4. Heuristique d'ordonnancement

Cette partie introduit un algorithme d'ordonnancement, pour répondre au problème posé par le modèle vu précédemment. Cet algorithme assigne à chaque tâche de chaque workflow, un emplacement réservé sur une VM. Il s'occupe également de créer l'ensemble des VM nécessaires, et affecte chacune de ces VM aux différents nœuds. On peut découper l'algorithme en deux sous parties présentées ci-dessous.

4.1. Ordonnancement d'un workflow dans l'environnement de Cloud

L'algorithme va chercher à réduire le nombre de nœuds allumés en regroupant les tâches sur les mêmes nœuds. Pour cela, on trie l'ensemble des nœuds disponibles par ordre croissant de vitesse ($speed_n^{\mathcal{N}}$). L'objectif de ce tri, est de placer les tâches en priorité sur les nœuds les plus lents afin de laisser le maximum de ressources disponibles pour les autres workflows. Afin d'être sûr de placer une tâche après ses prédécesseurs, et de respecter les contraintes temporelles, on ordonne le placement de tâches, en fonction de leur rang, (plusieurs tâches peuvent avoir le même rang, et s'exécuter en parallèle). Le calcul du rang de chaque tâche, se fait grâce à un premier parcours de graphe, où :

$$\forall j \in \mathcal{J}\ rank_j = 1 + \max_{k \in pred_j} (rank_k) \tag{10}$$

Le placement sur le nœud le plus lent possible, peut engendrer un mauvais placement pour les successeurs d'une tâche, et les obliger à dépasser la *deadline* du workflow associé. Par conséquent l'algorithme dispose d'un système de *backtrack*, afin de revoir le placement d'une tâche et de la positionner sur un nœud plus rapide. Lorsque l'ordonnancement d'une tâche sur la machine la plus rapide ne permet pas de respecter la contrainte de *deadline*, l'algorithme revient (*backtrack*) sur les predécesseurs de la tâche. Cette heuristique est représentée par l'Algorithme 1.

Il prend en entrée la liste des tâches ordonnées $\mathtt{rank}^{\mathsf{T}}$ du workflow $G = (T, D)$, et le rang actuel r (par récursivité). L'ordonnancement des workflows se fait en parcourant tous les workflows par ordre croissant de *deadline*.

Algorithme 1 Placement d'un workflow dans le Cloud

function ORDRANG($\mathtt{rank}^{\mathsf{T}}$, r, $\mathcal{N}$, *deadline*)
  echec $\leftarrow$ null, $i \leftarrow 0$
  while $i < |\mathtt{rank}^{\mathsf{T}}[r]| \wedge$ echec $=$ null **do**
    echec := SELECTNŒUD($\mathtt{rank}^{\mathsf{T}}[r]_i$, $\mathcal{N}$, *deadline*)
    $i := i + 1$
  end while
  if echec $=$ null $\wedge\ |\mathtt{rank}^{\mathsf{T}}| > r + 1$ **then**
    ORDRANG($\mathtt{rank}^{\mathsf{T}}$, $r + 1$, $\mathcal{N}$, *deadline*)
  else if echec $\neq$ null **then**
    BACKTRACK($\mathtt{pred}^{\mathcal{J}}_{\mathtt{echec}}$)
  end if
end function
function SELECTNŒUD(j, $\mathcal{N}$, *deadline*)
  for all $n \in \mathcal{N}$ **do**
    $\mathtt{start}^{\mathcal{J}}_j \leftarrow$ CALCULERSTART(j, n) ▷ Contrainte (5).
    $\mathtt{exec}^{\mathcal{J}}_j \leftarrow$ CALCULEREXEC(j, n) ▷ Équation (4).
    success $\leftarrow$ ORDNŒUD(j, n, $\mathtt{exec}^{\mathcal{J}}_j$, $\mathtt{start}^{\mathcal{J}}_j$, *deadline*)
    if success $=$ null **then return** j **else return** null
  end for
end function

Algorithme 2 Ordonnancement d'une tâche sur un nœud

function ORDNŒUD(j, n, $\mathtt{exec}^{\mathcal{J}}_j$, $\mathtt{start}^{\mathcal{J}}_j$, *deadline*)
  meilleur $\leftarrow$ (infini, infini, None, None)
  for all $v \in \mathcal{V}[n]$ **do**
    place $\leftarrow$ ORDVM(v, j, $\mathtt{exec}^{\mathcal{J}}_j$, $\mathtt{start}^{\mathcal{J}}_j$, *deadline*)
      ▷ Contraintes (2), (3), (6), (7)
    meilleur := **if** meilleur < place **then** meilleur **else** place
  end for
  $v \leftarrow$ NOUVELLEVM($\mathtt{start}^{\mathcal{J}}_j$, j, n) ▷ Contraintes(1),(2),(3)
  if $v \neq$ None **then**
    place $\leftarrow$ ORDVM(v, j, $\mathtt{exec}^{\mathcal{J}}_j$, $\mathtt{start}^{\mathcal{J}}_j$, *deadline*)
    if place $<$ meilleur **then**
      VALIDERALLUMAGE(v)
      return place
    else
      return meilleur
    end if
  end if
  return meilleur
end function

4.2. Placement d'une tâche sur un Nœud

Dans l'Algorithme 1, la fonction `SelectNœud` parcourt tous les nœuds par ordre croissant de vitesse et appelle la fonction `ORDNŒUD` présentée par l'algorithme 2. Cette fonction cherche une place disponible sur un nœud pour une tâche, et respecte les contraintes du modèle. Le placement d'une tâche i correspond au tuple $(\mathtt{start}^{\mathcal{J}}_i, \mathtt{exec}^{\mathcal{J}}_i, \mathtt{loc}^{\mathcal{J}}_i)$. Cette heuristique ne considère pas de migrations des VM, par conséquent, une VM est associée à un nœud et sa vitesse est constante. Chaque VM possède une liste d'occupation qui correspond aux capacités restantes de la VM à chaque instant. L'ordonnancement d'une tâche sur une VM va correspondre à trouver un emplacement dans cette liste assez grand pour accueillir la tâche, et qui respecte les contraintes de capacité. Cette recherche (`OrdVM` de l'Algorithme 2) se fait de manière gloutonne, et lorsqu'aucun emplacement n'est trouvé, on ajoute la tâche à la fin de la liste d'occupation (en respectant l'Equation (2)) . On considère une relation d'ordre sur deux placements p_i et q_i, tel que $p_i < q_i$ si $p_{\mathtt{exec}^{\mathcal{J}_i}} + p_{\mathtt{start}^{\mathcal{J}}_i} < q_{\mathtt{exec}^{\mathcal{J}}_i} + q_{\mathtt{start}^{\mathcal{J}}_i}$, pour $i \in \mathcal{J}$. L'ajout d'une tâche en fin de liste d'occupation engendre un allongement de la durée de vie de la VM, par conséquent il est nécessaire de vérifier l'Equation (1) pour l'hôte de la VM. Les nœuds possèdent un ensemble de VM allumées, et ont la capacité d'en allumer de nouvelles. L'Algorithme 2 représente la fonction qui va chercher une place pour une tâche j, et allume les VM nécessaires.

5. Résultats préliminaires d'expérimentation

L'experience qui a été menée a consisté en la simulation d'un nombre variable de nœuds pour l'ordonnancement d'un nombre constant de workflows. La figure 1 présente les résultats de l'exécution de l'heuristique pour l'ordonnancement de 10 000 workflows sur un nombre de nœuds variant de 1 à 100.

Les nœuds simulés possèdent une vitesse dépendante de leur identifiant, qui est calculé selon la suite suivante :

$$speed_n = \begin{cases} 1 & \text{si } n = 0 \\ speed_{n-1} + 1 & \text{si } n \bmod 10 = 0 \\ speed_{n-1} & \text{sinon} \end{cases} \quad (11)$$

Chaque nœud est connecté à tous les autres avec une bande passante 10 fois inférieur à leur bande passante *loopback*. Un unique type de workflow est utilisé dans cette experimentation. Il est relativement simple mais malgré tout hétérogène comme on peut le voir dans la figure 2. Dans cette experimentation, nous simulons deux patrons de VM différents, le premier est un patron de VM Windows, et le second celui d'une VM Linux.

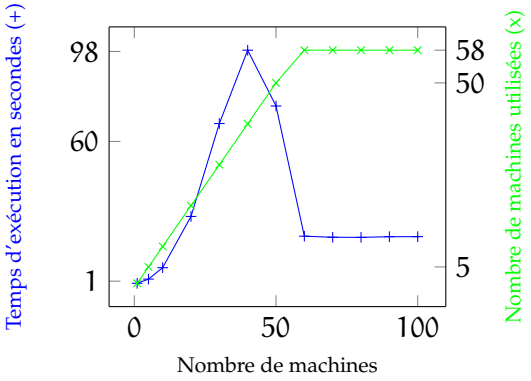

FIGURE 1

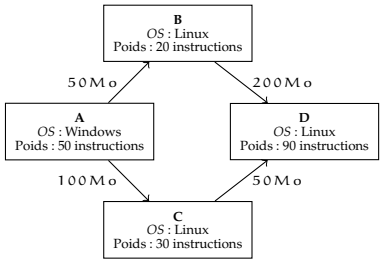

FIGURE 2 – Workflow de 4 tâches

La simulation a été effectuée sur une machine unique dont les caractéristiques téchniques sont les suivantes : Intel Core i5-6260U - 1.80GHz, 32 Go de mémoire vive. On observe un pic du temps d'exécution lorsque le nombre de nœuds est égal à 40, ceci est en corrélation avec le nombre de workflows qui ont pu être placés. Avant d'atteindre ce point, il n'est pas possible d'ordonnancer tous les workflows en respectant leurs deadline car il n'y a pas suffisament de machines. L'ordonnancement sur 40 nœuds nécessite une résolution complexe, pour laquelle un nombre important de *backtrack* est nécessaire, ce qui explique le pic du temps de résolution à 40 nœuds. Lorsque le nombre de nœuds utilisables croît, il n'est plus nécessaire d'effectuer autant de *backtrack*, et le temps d'exécution diminue. Une convergence semble atteinte pour 58 machines utilisées et un temps de résolution de l'ordre de 20 secondes. Lorsque le nombre de machine augmente après ce point, le temps de résolution reste constant.

6. Conclusion

Dans cet article nous avons présenté un modèle pour l'ordonnancement de workflows de type *dataflow*, dans le Cloud privé. Nous avons également développé une première heuristique d'ordonnancement, afin de répondre à une version simplifiée de ce modèle. Dans de futurs travaux, nous chercherons à optimiser cette heuristique et à la rendre plus dynamique (migration), puis nous la comparerons à d'autres algorithmes d'ordonnancement existants. Le modèle sera également complexifié afin de prendre en considération les différentes simplifications qui ont pu être faites dans sa version actuelle (bande passante non limitée, pas de migration de tâches etc.). Il est aussi envisageable d'étendre ce modèle afin par exemple de répondre à un problème d'ordonnancement dans un environnement hybride de type Cloud/HPC.

Bibliographie

1. Abrishami (S.), Naghibzadeh (M.) et Epema (D. H.). – Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, vol. 29, n1, 2013, pp. 158 – 169. – Including Special section : AIRCC-NetCoM 2009 and Special section : Clouds and Service-Oriented Architectures.
2. Alkhanak (E. N.), Lee (S. P.), Rezaei (R.) et Parizi (R. M.). – Cost optimization approaches for scientific workflow scheduling in cloud and grid computing : A review, classifications, and open issues. *Journal of Systems and Software*, vol. 113, 2016, pp. 1 – 26.
3. Bittencourt (L. F.) et Madeira (E. R. M.). – Hcoc : a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications*, vol. 2, n3, Dec 2011, pp. 207–227.
4. Caniou (Y.), Caron (E.), Kong Win Chang (A.) et Robert (Y.). – *Budget-aware scheduling algorithms for scientific workflows on IaaS cloud platforms*. – Research Report nRR-9088, INRIA, août 2017.
5. Coffman (E.), Garey (M.) et Johnson (D.). – Bin packing with divisible item sizes. *Journal of Complexity*, vol. 3, n4, 1987, pp. 406 – 428.
6. Coullon (H.), Le Louët (G.) et Menaud (J.-M.). – Virtual Machine Placement for Hybrid Cloud using Constraint Programming. – In *ICPADS 2017*, Shenzhen, China, décembre 2017.
7. Deelman (E.), Gannon (D.), Shields (M.) et Taylor (I.). – Workflows and e-science : An overview of workflow system features and capabilities. *Future Generation Computer Systems*, vol. 25, n5, 2009, pp. 528 – 540.
8. Gao (Y.), Guan (H.), Qi (Z.), Hou (Y.) et Liu (L.). – A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences*, vol. 79, n8, 2013, pp. 1230 – 1242.
9. Le Louët (G.) et Menaud (J.-M.). – OptiPlace : designing cloud management with flexible power models through constraint programing. – In *International Conference on Utility and Cloud Computing*, Dresden, Germany, décembre 2013.
10. Li (J.), Peng (J.), Lei (Z.) et Zhang (W.). – An energy-efficient scheduling approach based on private clouds. *Journal of Information &computational Science*, vol. 8, n4, 2011, pp. 716–724.
11. Mell (P. M.) et Grance (T.). – *SP 800-145. The NIST Definition of Cloud Computing*. – Rapport technique, Gaithersburg, MD, United States, 2011.
12. Rodriguez (M. A.) et Buyya (R.). – Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds. *IEEE Transactions on Cloud Computing*, vol. 2, n2, April 2014, pp. 222–235.
13. Shi (J.), Luo (J.), Dong (F.) et Zhang (J.). – A budget and deadline aware scientific workflow resource provisioning and scheduling mechanism for cloud. – In *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 672–677, May 2014.
14. Sotskov (Y.) et Shakhlevich (N.). – Np-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*, vol. 59, n3, 1995, pp. 237 – 266.