Generic deployment of applications on heterogeneous distributed platforms

Benjamin Depardon

Laboratoire de l'Informatique du Parallélisme

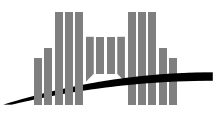
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Generic deployment of applications on heterogeneous distributed platforms

Benjamin Depardon February 2008

Research Report N° RR2008-06

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Generic deployment of applications on heterogeneous distributed platforms

Benjamin Depardon

February 2008

Abstract

From user's point of view Grid computing requires an easy, automatic and efficient resource selection as well as application installation and execution. This problem refers to the deployment process. A major lack of today's deployment tools lies in their inefficient planning algorithms to select and allocate resources for the applications. This paper presents several planning heuristics which aims at minimizing the communication cost between the applications, balancing the load among the processors, and maximizing the number of applications instances on the platform, for sequential and hierarchical applications. These heuristics have been implemented into the deployment tool ADAGE at the price of extending its generic model to support hierarchical applications. Finally, the paper presents some simulations and experimental results.

Keywords: Grid computing, deployment, mapping, middleware, heterogeneous platform.

Résumé

Les actuelles plates-formes de calcul que sont les grilles confrontent les utilisateurs à une complexité croissante d'utilisation de l'environnement. La nécessité d'avoir une méthode simple, automatique et efficace de sélection des ressources, ainsi que d'installation et d'exécution des applications devient primordiale. Les outils de déploiement actuels manquent de maturité concernant leurs algorithmes de planification pour sélectionner et allouer des ressources aux applications. C'est pourquoi il est nécessaire d'en proposer de nouveaux qui soient efficaces et qui supportent aussi bien des applications séquentielles, parallèles que hiérarchiques. Cet article présente différentes heuristiques qui visent à répartir la charge entre les différents nœuds de calcul, à maximiser le nombre d'instances déployées sur la plate-forme, et à minimiser les coûts de communication entre les applications, pour des applications séquentielles et hiérarchiques. Ces heuristiques ont été implémentées dans le logiciel de déploiement ADAGE. Finalement, cet article présente des simulations et des résultats expérimentaux.

Mots-clés: Grilles de calcul, déploiement, planification, intergiciel, plate-forme hétérogène.

1 Introduction

Le calcul haute performance consiste à résoudre des problèmes nécessitant de grandes capacités de calcul. Ce sont par exemple les simulations cosmologiques, les prédictions météorologiques, la détermination de formes de protéines, *etc.* Ces problèmes peuvent être soit des tâches totalement indépendantes, soit un ensemble de tâches couplées entre elles et nécessitant des communications (comme par exemple des workflows). Historiquement ces calculs ont été réalisés sur des ordinateurs massivement parallèles, puis l'émergence des réseaux hautes performances a permis l'agrégation de ressources de calcul homogènes en grappes, puis de ressources hétérogènes fortement distribuées en grilles. Ces dernières permettent aux utilisateurs d'accéder à une grande puissance de calcul, mais n'offrent pas le même environnement homogène présent sur les superordinateurs. De nouvelles difficultés apparaissent alors: comment choisir les ressources sur lesquelles exécuter les applications? Entre deux placements les performances obtenues peuvent varier (variation de la puissance de calcul allouée, du coût de communication, *etc.*). Deux problématiques traitent ces problèmes: l'ordonnancement et le déploiement. L'ordonnancement considère le placement et l'ordre d'exécution de chaque tâche dans le temps (temps de début et de fin des tâches), alors que le déploiement s'intéresse au choix des nœuds pour beaucoup d'instances d'un ou plusieurs processus sans avoir de contraintes temporelles. Les grilles étant des environnements dynamiques très complexes, il est nécessaire d'apporter des outils simples et efficaces aidant les utilisateurs à exécuter leurs applications

Le problème que nous considérons est le suivant. Nous nous intéressons à un ensemble de processus à déployer sur une grille. Ces applications peuvent être séquentielles ou hiérarchiques. Chacune d'elles possède un nombre maximum d'instances à déployer (représentant le nombre de fois que le service peut être répliqué, ou bien un nombre de licences disponibles, ...). Chaque application peut communiquer avec une ou plusieurs autres, mais nous ne considérons pas de contraintes de précédence: toutes les applications doivent être présentes au même moment sur la plate-forme, même si des schémas de communication peuvent exister entre elles. La plate-forme est constituée de nœuds regroupés en grappes de calcul. Nous supposons que chaque nœud peut communiquer avec tous les autres, avec des coûts différents (nous avons donc un graphe complet). Notre but est le suivant: trouver un déploiement qui permette d'avoir **un maximum d'instances**, tout en **minimisant le coût des communications** et en **répartissant la charge** entre les nœuds, afin de répondre à un maximum d'utilisateurs tout en ne surchargeant pas la plate-forme.

Le **déploiement** d'applications consiste à **sélectionner**, puis à **allouer des ressources de calcul** à ces applications. Actuellement, les applications scientifiques sont encore principalement déployées à la main: du choix et allocation des ressources, aux phases d'installation et de lancement. Des outils pour simplifier ce processus existent, mais restent incomplets: certains sont spécifiques à un type d'application [1, 4], d'autres plus génériques traitent de graphes de tâches acycliques représentant les dépendances entre les tâches [6]. Une dernière catégorie s'occupe d'un ensemble quelconque de processus [3, 9, 10, 11, 13]. Ces outils ne couvrent pas tous le même spectre du processus de déploiement, et peu proposent des méthodes automatiques pour créer le plan de déploiement (un état de l'art plus précis peut être trouvé dans [7]). C'est pourquoi nous proposons des heuristiques pour traiter ce problème.

Pour commencer nous présentons en section 2 l'intergiciel de grille DIET, et deux outils de déploiement associés. Afin de créer des plans de déploiement, nous proposons en section 3 plusieurs heuristiques: une pour créer une hiérarchie, et deux types d'heuristiques

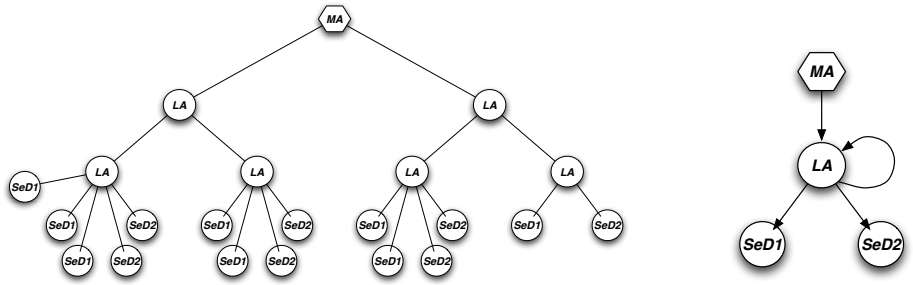
pour déployer plusieurs processus. Ces heuristiques ont été implémentées dans ADAGE. Nous présentons des résultats de simulations et d'expérimentations en section 4. Finalement, nous dressons la conclusion de nos travaux et discutons des travaux futurs.

2 Diet et outils de déploiement associés

Ces travaux ont été initiés par le souci de déployer efficacement l'intergiciel de grille DIET. Nous présenterons donc rapidement dans cette section DIET, puis les outils de déploiement associés. Cependant, notons que ces travaux sont intrinsèquement valables pour tout déploiement d'applications hiérarchiques. DIET n'est pas la seule application hiérarchique, d'autres intergiciels (tel que WebCom-G [12]) ou applications (de type maître/esclaves) suivent ce modèle, et peuvent bénéficier de méthodes automatiques de déploiement.

2.1 Application hiérarchique: Diet

DIET est un intergiciel de grille permettant à un client de soumettre des requêtes à un ensemble de serveurs pour résoudre un problème. Les requêtes sont envoyées des clients vers la racine de la hiérarchie: le *Master Agent* (MA). Ce dernier transmet les requêtes à la hiérarchie sous-jacente, elles sont propagées jusqu'aux feuilles (les SED, *Server Daemons*) qui répondent en précisant leur capacité à les traiter ou non. Un choix est effectué parmi les SED valides, et une réponse est retournée au client. La figure 1a présente un exemple de hiérarchie DIET composée d'une racine (le MA), d'un ensemble de nœuds intermédiaires (les LA, *Local Agents*, qui ont pour tâche de transmettre les requêtes et de réaliser l'ordonnancement des requêtes sur leurs sous-arbres), et enfin les SED qui ont la charge de traiter les requêtes. DIET peut également se présenter sous la forme d'une multi-hiérarchie: plusieurs hiérarchies DIET reliées par la racine de façon statique ou dynamique en pair à pair.



(a) Exemple d'une hiérarchie DIET (b) Modèle

Figure 1: Exemple d'une hiérarchie DIET et sa représentation compacte

Étant donné que la forme de la hiérarchie dépend des ressources disponibles, il n'est pas possible de décrire *une* hiérarchie qui offre un bon débit de requêtes traitées sur n'importe quelle plate-forme. C'est pourquoi il est préférable de laisser le logiciel de déploiement choisir de façon automatique la forme de la hiérarchie. On peut remarquer que ce type d'application a une partie *récursive*: les LA ont pour enfants des LA, ou des SED. Nous pouvons donc représenter la hiérarchie de façon compacte comme sur la figure 1b. Cette représentation indique que le MA a pour fils des LA, et que les LA ont pour fils soit des LA, soit des SeD

de deux types différents. La partie de la hiérarchie à partir du nœud récursif peut donc être répliquée pour construire une hiérarchie.

2.2 Outils de déploiement

Nous disposons de deux outils pour déployer DIET. Le premier, **GoDiet** [4], a été spécialement conçu dans ce but. Il prend en entrée un fichier XML contenant à la fois la description des ressources, l'emplacement des binaires et bibliothèques sur ces ressources, ainsi que la description en extension de la hiérarchie DIET à déployer. Il ne réalise pas le transfert des binaires et bibliothèques, mais configure et lance de façon ordonnée la hiérarchie.
Le second est **Adage** [11] (<< ***A****utomatic* ***D****eployment of* ***A****pplications in a* ***G****rid* ***E****nvironment* >>), c'est un outil générique de déploiement d'applications. Il prend en entrée une description de l'application à déployer décrite en XML, suivant un schéma XML défini par l'utilisateur. Elle est ensuite traduite en description générique (GADe: ***G****eneric* **A**pplication **De**scription). À partir de cette dernière et d'un autre fichier décrivant les ressources, ADAGE génère un plan de déploiement à l'aide d'un plugin de planification, puis réalise le déploiement en transférant les fichiers requis, configure les applications et les lance. Cependant, ADAGE ne propose que deux plugins simplistes de planification: tourniquet (round-robin) et aléatoire. Une des contributions de cet article est la mise au point d'heuristiques pour déployer un ensemble de processus, ainsi qu'une pour créer une hiérarchie. Elles ont ensuite été implémentées dans ADAGE, avec pour seul surcoût celui d'étendre GADe pour qu'il accepte la description de hiérarchies de façon compacte comme celle de la figure 1b, avec un *nœud récursif*.

3 Heuristiques de déploiement

Nous présentons dans cette section le modèle considéré, et nos heuristiques: une pour déployer une hiérarchie, puis nous généraliserons pour déployer un ensemble de processus [7].

3.1 Modèle

Notre but est de trouver un placement pour un groupe de processus sur un ensemble de ressources. Nous étudions notre problème en *régime permanent* [2] (nous voulons pouvoir répondre à un maximum de requêtes de la part de clients par unité de temps). Notre approche se focalise sur le déploiement initial de l'application, c'est une approche statique. Nous ne considérons pas encore du redéploiement dynamique: tous les processus doivent être présents sur la plate-forme au même moment, sans connaître l'ordre d'exécution.
Nous considérons une plate-forme constituée d'un ensemble de m nœuds p_i regroupés en grappes (nœuds homogènes), chaque p_i peut effectuer une certaine quantité de travail en une unité de temps. Tous les nœuds peuvent communiquer entre eux avec un certain coût de communication pour envoyer une unité de données à un autre nœud $p_{i'}$. Nous voulons déployer un ensemble de n processus a_j. Chaque a_j a un nombre maximum d'instances nb_j^{max} qui peuvent être déployées (sur n'importe quel nœud), et une quantité d'opérations à effectuer en une unité de temps. Un a_j peut communiquer avec un ou plusieurs autres $a_{j'}$ avec une quantité de données à transférer par unité de temps. Chaque a_j a une fréquence d'utilisation f_j représentant la façon dont il est utilisé par les utilisateurs. Nous définissons également une variable booléenne $\delta_i^{j,k}$ qui vaut 1 si l'instance k de a_j est placée sur p_i. Notre objectif est de trouver un déploiement qui répartit la charge sur les nœuds, minimise les coûts de

communication, et maximise le nombre d'instances déployées. Pour mesurer la répartition de charge, nous utilisons la variance sur la charge des nœuds ($Var(charge_{noeud})$). Le coût de communication est la somme de tous les coûts de communications engendrés par les différents processus. Nous définissons ainsi une fonction objective que nous souhaitons minimiser:

$$LB = \frac{Var(charge_{noeud}) + coût^2_{communication}}{Nb\ instances\ déployées}$$

3.2 Déploiement d'une hiérarchie

Dans [5] les auteurs prouvent que le déploiement optimal (au sens du débit de requêtes traitées) pour un intergiciel hiérarchique sur une plate-forme homogène, est un *CSD tree* (***C**omplete **S**panning **d**-ary tree*) un arbre couvrant complet d-aire: un arbre pour lequel tous les nœuds internes, à l'exception peut être d'un, ont exactement d enfants.
Nous voulons déployer des hiérarchies sur une plate-forme hétérogène en maximisant le nombre de feuilles (de serveurs). Nous proposons une heuristique qui crée une hiérarchie proche d'un CSD tree par grappe si la hiérarchie est la seule application à déployer sur ces ressources, sinon le degré de chaque nœud est adapté en fonction de la charge des ressources. Nous nous rapprochons ainsi d'un optimum local (par grappe), et espérons tendre vers un optimum global.
Tout d'abord, le nœud le plus puissant est pris pour la racine pour qu'il supporte une grosse charge. Puis, l'heuristique traite les grappes séparément: les premiers nœuds de la hiérarchie sont placés sur des nœuds physiques jusqu'à atteindre le *nœud récursif*. Puis, des enfants de ce nœud récursif sont placés tant que le nœud récursif peut supporter la charge. Quand le nœud physique sur lequel est placé le nœud récursif est complètement chargé, alors un nouveau nœud récursif est ajouté au dessus, et la hiérarchie est reconstruite avec des nœuds récursifs jusqu'à atteindre la profondeur précédente (un nœud physique est choisi pour chacun d'eux), et le processus d'ajout des enfants recommence. Ceci est répété tant qu'il reste des ressources. La figure 2 présente le processus de construction sur une grappe, la description de la hiérarchie contient un nœud récursif (`NR2`) et un enfant (`NR3`). Les nœuds gris sont entièrement chargés.

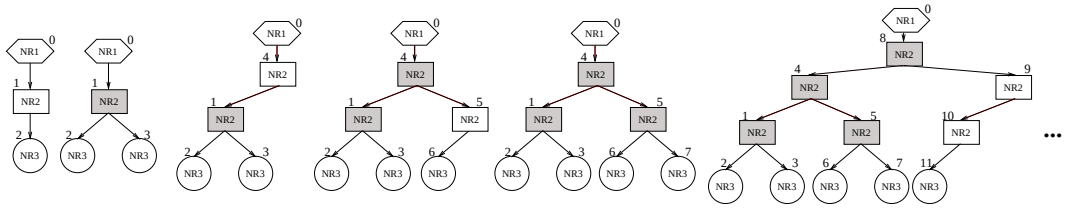


Figure 2: Heuristique de déploiement d'une hiérarchie.

3.3 Heuristiques

Heuristiques: distribution des nœuds, et remplissage de la plate-forme

L'algorithme 1 montre les principales étapes de ces heuristiques. Pour chaque a_j, nous déterminons tout d'abord un sous-ensemble de nœuds p_i sur lesquels nous pourrons le déployer à

l'aide d'une première heuristique. Puis, un nombre maximum d'instances est placé sur ces p_i en choisissant les p_i dans le sous-ensemble avec une seconde heuristique. Si à la fin de cette première étape certains a_j ont des instances non placées, on essaye alors d'ajouter de nouvelles instances, à l'aide d'un tourniquet sur une liste contenant les a_j répliqués en fonction de leur f_j (si nous avons deux processus a_1 et a_2, avec $f_1 = 2 \times f_2$, alors la liste du tourniquet est $\{a_1, a_1, a_2\}$, ceci afin d'obtenir une solution proche des fréquences désirées), cette étape est répétée tant que la fonction objective LB peut être améliorée.
Chaque heuristique distribue les p_i aux a_j en leur allouant une puissance de calcul proportionnelle à leur fréquence, seule la façon de sélectionner les p_i diffère (algorithme 1, ligne 2):

- ***Aléatoire* (Aléa)**: p_i choisis de façon aléatoire, avec priorité pour les a_j avec le f_j le plus haut.
- ***Puissance de calcul* (Puis)**: plus puissants p_i donnés en priorité aux a_j ayant le plus gros w_j.
- ***Clustering* (Clus)**: distribution réalisée en deux étapes. Détection des sous-graphes indépendants à l'aide d'un parcours en profondeur du graphe général contenant tous les processus: obtention de groupes de a_j avec des communications à l'intérieur du groupe, mais pas entre les groupes. Tri des sous-graphes par fréquence décroissante. Tri des grappes de calcul par charge décroissante et puissance de calcul décroissante. Allocation des p_i les moins chargés de la grappe la plus puissante au premier sous-graphe. Seconde étape: tri des a_j du sous-graphe par besoins en communication décroissants, et distribution des p_i aux a_j. Itération pour chaque sous-graphe.

Étant donné un ensemble de nœuds, il est nécessaire de choisir ceux sur lesquels placer chacune des instances (algorithme 1, lignes 3 et 9):

- ***Aléatoire* (Aléa)**: p_i choisi aléatoirement, une instance est placée dessus.
- ***Liste de charge* (List)**: tri des p_i par charge croissante. Placement des instances en suivant cet ordre, la liste est retriée lorsqu'on atteint la fin.
- ***Minimise communications* (Comm)**: choix de la grappe permettant de minimiser les coûts de communication, puis choix du p_i le moins chargé de cette grappe.
- ***Glouton* (Glou)**: choix du p_i minimisant LB.

Heuristique: Affinité

Cette heuristique est inspirée d'une heuristique d'ordonnancement sur plate-forme hétérogène pour des tâches indépendantes nécessitant des fichiers partagés en entrée [8]. L'Algorithme 2 présente l'heuristique: tout d'abord les sous-graphes indépendants sont détectés, et triés par fréquence décroissante (puis besoins en communications et calcul décroissants). Pour chaque sous-graphe, les grappes de calcul sont triées par charge croissante, et puissance de calcul décroissante, puis une instance de chacun de ses a_j est placée sur la première grappe. On construit ensuite un liste d'affinité pour chaque p_i: pour chaque p_i et chaque a_j, on calcule l'*affinité* entre eux, c'est-à-dire la valeur $affinité(a_j, p_i) = Var(charge_{noeud}) + coût^2_{communication}$ si une instance de a_j est placée sur p_i. *affinité* prend en compte deux des métriques que nous

Algorithme 1 : Heuristique, Distribution des nœuds, et remplissage de la plate-forme.

pour tout a_j **faire**
  $list\_nodes \leftarrow$ nœuds choisis pour a_j
  Placer un nombre maximum d'instance de a_j sur $list\_nodes$
si $\exists a_j | \sum_i \sum_j \sum_k \delta_i^{j,k} < nb_j^{max}$ **alors**
  $A \leftarrow \{a_j | \sum_i \sum_j \sum_k \delta_i^{j,k} < nb_j^{max}\}$
  $A_f \leftarrow$ processus de A *répliqués* en fonction de leur fréquence f_j
  tant que LB décroît et $A_f \neq \emptyset$ **faire**
    pour tout $a_j \in A_f$ **faire**
      si une nouvelle instance de a_j améliore LB **alors**
        Placer cette instance

voulons optimiser: la distribution de la charge sur les processeurs, et les coûts de communication. Les listes d'affinité sont triées par valeur croissante. Pour réaliser le placement final, tant qu'il nous reste des instances à déployer, et que les p_i ne sont pas totalement chargés: pour chaque itération, le couple (a_j, p_i) minimisant LB est choisi (pour chaque p_i seul le premier a_j de la liste d'affinité est considéré), et a_j est placé sur p_i.

Algorithme 2 : Heuristique, Affinité

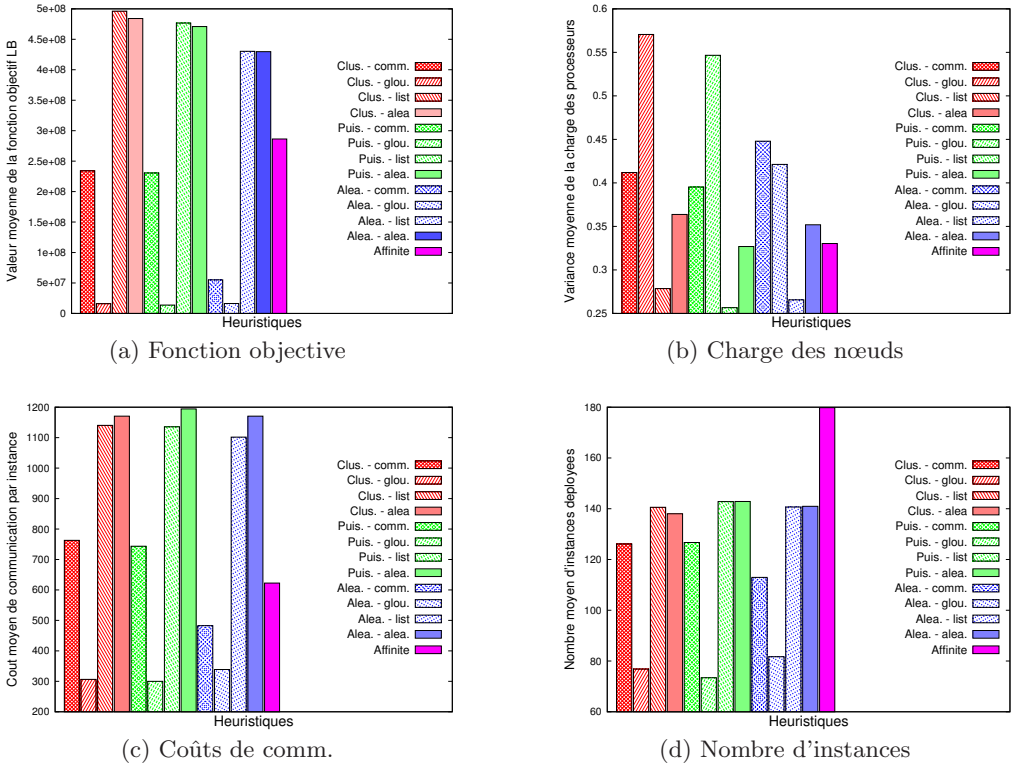
Détection des sous-graphes, et tri par fréquence, besoins communication et travail décroissants
pour tout sous-graphe **faire**
  Choisir la grappe la moins chargée et la plus puissante
  pour tout $a_j \in$ sous-graphe **faire**
    placer une instance de a_j sur le nœud p_i le moins chargé
pour tout p_i **faire**
  pour tout a_j **faire**
    Calculer $affinité(a_j, p_i) = Var(charge_{noeud}) + coût^2_{comm}$
  Construire et trier les listes d'affinité $L(p_i)$
tant que $\exists a_j | \sum_{i=1}^{m} \sum_{k=1}^{nb_j^{max}} \delta_i^{j,k} < nb_j^{max}$ et $\exists p_i | p_i$ non entièrement chargé **faire**
  pour tout p_i non entièrement chargé **faire**
    Soit a_j premier processus de $L(p_i)$ qui peut être placé sur p_i
    Calculer LB si a_j est placé sur p_i
  Choisir le couple (a_j, p_i) qui minimise LB, et placer a_j sur p_i

4 Simulations et expériences

4.1 Simulations de planification

Nous comparons les heuristiques à l'aide d'un simulateur. Les résultats suivants ne sont valables que pour des applications séquentielles (pas d'applications hiérarchiques). Des graphes d'applications ont été générés aléatoirement. Le nombre de processus, leur cardinalité maximum, leurs besoins en communication et calcul ont été choisis aléatoirement. Nous avons généré ainsi 1800 graphes et réalisé les simulations sur une plate-forme composées de quatre

grappes de calcul contenant 10, 20, 20 et 30 nœuds de puissances variables.



(a) Fonction objective (b) Charge des nœuds

(c) Coûts de comm. (d) Nombre d'instances

Figure 3: Résultats des simulations pour les différentes heuristiques.

La figure 3a présente la valeur moyenne de la fonction objective LB. En regardant ce graphe, les heuristiques qui semblent les plus intéressantes sont *les gloutonnes*, à l'opposé des heuristiques par *liste de charge* et *aléatoire*. Si l'on regarde en détail les trois valeurs à optimiser (figures 3b, 3c et 3d), toutes les heuristiques ne se comportent pas de la même façon. *affinité* semble la plus intéressante: elle déploie un maximum d'instances, tout en ayant un coût de communication assez bas et une des meilleures répartition de charge, alors que les heuristiques *gloutonnes* ont le meilleur coût de communication, mais peu d'instances déployées et une mauvaise répartition de la charge. Les heuristiques avec *listes de charge* ont la meilleure répartition de charge. Il faut donc choisir l'heuristique à utiliser en fonction de ce que l'on souhaite privilégier au sein de notre fonction objective: quelle combinaison de *répartition de charge*, *coût de communication* et *nombre d'instances déployées*? L'heuristique qui satisfait le plus les trois valeurs est *affinité*.

4.2 Déploiement sur Grid'5000

Afin de valider l'heuristique pour les hiérarchies, nous déployons DIET avec trois autres services: un de nommage et deux de surveillance. La description de la hiérarchie est la même que celle donnée sur la figure 1b (deux services différents sous le LA). Nous souhaitons avoir le plus de services possible (les SED), il n'y a donc pas de contrainte sur le nombre d'éléments répliqués dans la hiérarchie DIET. Nous réalisons nos expériences sur la plate-forme expérimen-

tale française Grid'5000[1], et réservons entre 25 et 305 nœuds pour réaliser les déploiements. Afin de déployer la même hiérarchie aussi bien avec ADAGE qu'avec GODIET, ADAGE crée la hiérarchie avec les heuristiques, puis l'écrit dans un fichier XML au format GODIET.
La figure 4 présente le temps moyen de déploiement en utilisant ADAGE et GODIET. Comme GODIET ne réalise pas le transfert des fichiers durant son processus de déploiement, nous transférons les fichiers nécessaires à l'aide d'un script qui lance des commandes `scp` en arrière plan sur chaque nœud afin de transférer les mêmes fichiers qu'ADAGE. Le temps de transfert est ensuite ajouté au temps de déploiement de la hiérarchie par GODIET. On peut remarquer que même si ADAGE doit générer le plan de déploiement (incluant la forme de la hiérarchie) son temps de déploiement est bien plus faible que celui de GODIET (il lance les processus séquentiellement, alors qu'ADAGE parallélise leur lancement et optimise le transfert des fichiers).
La forme de la hiérarchie DIET dépend du nombre de grappes et du nombre de nœuds disponibles, ainsi que de leur puissance de calcul. Le tableau 1 présente le nombre de LA et de SED déployés par heuristique. Nous visons le plus grand nombre de SED afin de répondre à un maximum de requêtes. La hiérarchie a au moins un LA par grappe, parfois plus s'il y a assez de nœuds, et s'ils sont suffisamment puissants. Comme le nombre de SED par nœud dépend de la fréquence des CPU, le nombre de ressources peut être plus petit que le nombre d'éléments DIET déployés. Vu que nous n'avons ici que trois autres processus à déployer en même temps que la hiérarchie DIET, la différence entre les hiérarchies est assez faible.

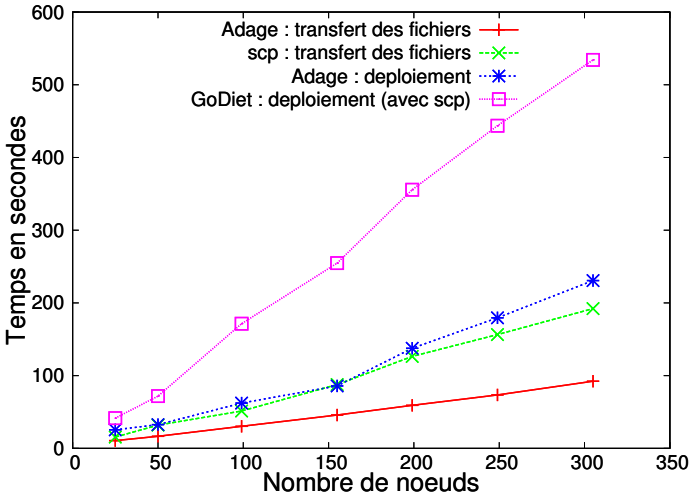


Figure 4: Temps de déploiement pour ADAGE et GODIET.

Nous avons déployé des hiérarchies DIET avec trois autres processus utiles pour son utilisation. La forme des hiérarchies est adaptée au nombre de ressources: le nombre de processus sur les nœuds, ainsi que la profondeur de la hiérarchie dépend des puissances de calcul des nœuds et de leur nombre. Concernant le temps de déploiement, ADAGE surpasse GODIET. GODIET a été conçu comme un ensemble de scripts non optimisés. L'utilisation d'un outil générique de déploiement est alors une meilleure alternative. En effet, la généricité permet d'avoir des algorithmes communs de génération de plan de déploiement, ainsi qu'un processus de

[1] http://www.grid5000.fr

Nb nœuds	Nb grappes	Clus-Comm	Clus-Glou	Clus-List	Clus-Aléa	Aléa-Comm	Aléa-Glou
50	1	1/47	1/48	1/48	1/48	1/48	1/48
155	6	6/223	8/222	8/222	8/223	8/223	8/222
305	8	14/490	17/492	14/492	16/493	14/492	14/491

Nb nœuds	Nb grappes	Aléa-List	Aléa-Aléa	Puis-Comm	Puis-Glou	Puis-List	Puis-Aléa
50	1	1/48	1/48	1/47	1/47	1/47	1/47
155	6	8/222	8/223	10/221	10/221	10/221	10/221
305	8	14/491	14/491	14/492	16/493	14/492	14/492

Table 1: Nombre de LA et de SED obtenus pour les différents déploiements (LA/SED).

déploiement entièrement optimisé de bout en bout.

5 Conclusion

L'utilisation de ressources hétérogènes réparties nécessite d'avoir des outils simples et efficaces pour déployer des applications. Cependant, les outils actuels manquent encore de maturité concernant leurs méthodes de sélection des ressources. Dans ce papier, nous présentons une heuristique pour déterminer la forme d'une application hiérarchique sur une plate-forme donnée, ainsi que deux types d'heuristiques: une basée sur deux sous-heuristiques (une pour définir un sous-ensemble de nœuds, et une pour choisir parmi les nœuds), et une seconde basée sur des listes d'affinité entre les nœuds et les processus. Nous tentons de satisfaire trois critères: minimiser les coûts de communication, répartir la charge entre les nœuds, et maximiser le nombre d'instances déployées. Nos simulations montrent qu'il n'y a pas *une* meilleure heuristique, même si la plus intéressante est *affinité*. Cependant en fonction du ou des critères de la fonction objective que l'on souhaite privilégier nous savons quelle heuristique mettre en œuvre. Nous avons également déployé des centaines d'éléments DIET à l'aide d'ADAGE, et montré que cet outil générique était bien plus efficace que GODIET, l'actuel outil de déploiement.

Le sujet est loin d'être clos. Les travaux futurs concerneront les applications parallèles, qui représentent une grande part des applications utilisées sur une grille. Un autre point important qui n'a pas été pris en compte est la compatibilité entre les applications et les ressources: une application peut ne pas être exécutable sur l'ensemble de la plate-forme à cause de contraintes sur la mémoire, ou encore des bibliothèques disponibles. Enfin d'autres contraintes peuvent être prises en compte: contention sur les liens de communication, redéploiement dynamique, déploiement de multi-hiérarchies, *etc.*

References

[1] G. Antoniu, L. Bougé, M. Jan, and S. Monnet. Large-scale Deployment in P2P Experiments Using the JXTA Distributed Framework. In *Euro-Par 2004: Parallel Processing*, pages 1038–1047, 2004.

[2] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters: why and how? In *6th Workshop on APDCM*, 2004.

[3] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. *IEEE International Conference on Cluster Computing*, pages 1–11, September 25th-28th 2006.

[4] E. Caron, P. K. Chouhan, and H. Dail. GoDIET: A deployment tool for distributed middleware on Grid'5000. In IEEE, editor, *EXPGRID workshop. In conjunction with HPDC-15.*, pages 1–8, 2006.

[5] P. K. Chouhan, H. Dail, E. Caron, and F. Vivien. Automatic middleware deployment planning on clusters. *International Journal of High Performance Computing Applications*, 20(4):517–530, 2006.

[6] E. Deelman, G. Singh, M. H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, and K. Vahi. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Sci. Prog. Jour.*, 13(3):219–237, 2005.

[7] B. Depardon. Déploiement générique d'applications sur plates-formes hétérogènes réparties. Rapport de Master, Juin 2007. http://www.ens-lyon.fr/LIP/Pub/Rapports/Master/Master2007/Master2007-02.pdf.

[8] A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. *J. Syst. Archit.*, 52(2):88–104, 2006.

[9] W. Goscinski and D. Abramson. Application deployment over heterogeneous grids using distributed ant. In *E-SCIENCE '05*, pages 361–368. IEEE Computer Society, 2005.

[10] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide-area networks using AI planning techniques. In *IPDPS '03*, page 3.1, 2003.

[11] S. Lacour, C. Pérez, and T. Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM Inter. Workshop on Grid Computing (Grid2005)*, 2005.

[12] J. P. Morrison, B. Clayton, D. A. Power, and A. Patil. Webcom-G: grid enabled metacomputing. *Neural, Parallel Sci. Comput.*, 12(3):419–438, 2004.

[13] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating Experimentation on Distributed Testbeds. In *Proceedings of the 20th IEEE/ACM Inter. Conf. on Automated Software Engineering (ASE 2005)*, 2005.