A GRASP approach for the machine reassignment problem

Michaël Gabay, Sofia Zaourar

Challenge ROADEF-EURO 2012 : Machine Reassignment
Équipe J19

Michaël Gabay*, Sofia Zaourar†

8 juin 2012

Nous présentons ici les méthodes que nous avons implémentées pour proposer des solutions au problème du challenge ROADEF-EURO 2012 : la réaffectation de processus sur des machines. Pour les qualifications, nous avions exploré de nombreuses pistes pour résoudre ce problème. Pour la phase finale, nous avons exploré d'autres piste mais les instances étant significativement plus grandes, nous avons finalement mis l'accent sur la recherche locale. Nous présentons ici les approches implémentées dans la seconde phase.

Mots clés Recherche locale, Méta-heuristiques, Machine reassignment, challenge ROADEF-EURO

Le problème

Le problème est le suivant : étant donné un ensemble de machines, hébergeant chacune différents processus, on cherche à réaffecter les processus pour minimiser le coût. Les processus sont regroupés en services et les machines sont regroupées par emplacements (locations) et voisinages. Afin d'être exécuté, chaque processus à besoin d'une certaine quantité de ressources.

Les contraintes sont les suivantes :

1. Chaque machine dispose d'une capacité maximale pour chaque ressource (capacity constraints) et pour certaines ressources, lorsqu'un processus est déplacé, l'espace n'est pas immédiatement libéré sur la machine initiale (transient usage constraints).
2. Deux processus d'un même service ne peuvent pas être hébergés sur une même machine (conflict constraints)
3. Chaque service doit être réparti sur un nombre minimum d'emplacements (spread constraints).
4. Certains service sont dépendants entre eux. C'est à dire que si le service s_1 dépend du service s_2, chacun des processus de s_1 doit avoir dans le voisinage de sa machine, une machine hébergeant un processus de s_2.

À nos yeux, les contraintes de dépendance 4 sont les plus difficiles pour ce problème. Celles-ci lient a priori toutes les affectations et empêchent les traitement séquentiels ou la décomposition directe du problème.

Nous avons identifié deux approches permettant de s'affranchir de ces contraintes de dépendances. La première approche est d'optimiser l'affectation des processus en se restreignant au voisinage de leurs machines initiales (ou plus généralement d'une solution réalisable). On peut alors utiliser des méthodes à base de programmation linéaire en nombres entiers (PLNE). La seconde technique est d'optimiser au niveau des processus en appliquant des mouvements locaux de type déplacement ou échange. Il devient alors facile de vérifier

*Laboratoire G-SCOP 46, avenue Félix Viallet - 38031 Grenoble Cedex 1, France
†Laboratoire Jean Kuntzmann - Inria Grenoble, 655 Avenue de l'Europe Montbonnot 38334, St Ismier, France

les contraintes de dépendance. Nous avons testé ces deux approches, la première en décomposant le problème selon les voisinages puis en résolvant un PLNE sur chacun ; la seconde en appliquant une recherche locale [1] guidée.

Le problème du challenge est un problème d'optimisation ; divers coûts rentrent donc également en jeu. Ces coûts sont de trois types différents :

1. Des coûts de charge des machines : chaque machine a un seuil (safety capacity) pour chaque ressource. Au-dessus de ce seuil, la charge supplémentaire a un coût.
2. Des coûts d'équilibrage des ressources : pour certains couples de ressources, trop consommer d'une seule des deux est coûteux. Ce coût permet d'équilibrer la charge des machines pour les ressources concernées.
3. Des coûts liés au déplacement des processus.

Les coûts de déplacement des processus peuvent être facilement pris en compte lors d'un mouvement. Ceux d'équilibrage en revanche sont plus complexes : afin de pouvoir les optimiser, il faut pouvoir considérer le problème à une échelle globale (des groupes de machines et processus). Seule la configuration finale compte, les mouvements intermédiaires n'ont pas vraiment de sens pour ces coûts. Les coûts de charge sont plus facilement maîtrisables localement : un mouvement impacte directement ceux-ci. Enfin, sur les instances fournies, nous avons pu observer que l'essentiel du coût était dû au coût de charge et d'équilibre entre ressources. D'où, le coût de charge étant plus facilement maîtrisable, nous utilisons une stratégie particulière pour tenter de le réduire.

Pour chaque instance du problème, une solution initiale était fournie. Pour les heuristiques, il peut être intéressant d'avoir plusieurs points de départ, nous avons donc également implémenté une méthode permettant d'obtenir d'autres solutions initiales. Cette méthode modélise le problème comme un problème de vector bin-packing multidimensionnel et est détaillée en section 1.

1 D'autres solutions initiales

Si l'on cherche des solutions réalisables pour les contraintes de capacité (y compris les transientes) seulement, le problème "ressemble" à un problème de Vector Bin Packing (VBP). Dans le VBP, on se donne n objets dont les tailles sont des vecteurs de d dimensions $o_1, \ldots, o_n$ et un nombre infini de boîtes de même taille fixée b (vecteur de dimension d). Le but est de placer tous les objets dans un nombre minimal de boîtes, de façon à ce que la somme des tailles des objets affectés à une boîte ne dépasse pas la taille de la boîte (i.e. pour chaque boîte et pour chaque dimension j, $\sum_{o_i \text{ dans la boîte}} o_i^j \leq b^j$).

Le problème de VBP est NP-complet, même en dimension $d = 1$. Les heuristiques qui existent dans la littératures pour $d \geq 1$ sont inspirées de celles de la dimension 1. Ce sont des heuristiques gloutonnes. Nous avons appliqué l'heuristique de Bin-Centering [4]. Pour cela, on définit une fonction de poids (ou volume) qui associe à chaque objet o_i un entier (ou réel) v_i. Plusieurs choix sont possibles pour définir le volume d'un objet ; on détaille le notre plus bas. Ensuite, pour chaque machine, on parcours la liste des objets pas encore placés dans l'ordre décroissant de leur volume et on place les processus qui "rentrent". Lorsque plus aucun objet ne rentre dans la boîte, on la "ferme" et on passe à la suivante, etc. Dans notre cas, les boîtes sont la machines, les objets sont les processus, et les dimensions sont les ressources. Les principales différences par rapport au problème de VBP sont que d'une part, on dispose d'un nombre fini de machines et d'autre part les machines ont des capacités différentes. Nous avons définis les volumes suivants, pour toutes machine m, et tout processus p :

$$v(m) = \sum_{r \in R} \frac{C(m,r)}{C(r)}, \quad v(p) = \sum_{r \in R} \frac{R(p,r)}{R(r)}, \tag{1}$$

où C(r) (resp.R(r)) désigne la capacité totale des machines (resp. le besoin total des processus) en la ressource r.

Par ailleurs, vu la forme du coût de charge, l'idéal est de remplir les machines au maximum sans dépasser la *safety capacity*. Cela nous a donné l'idée d'une autre heuristique, qu'on appelle *Bin Balancing* où l'on cherche à équilibrer la charge des différentes machines plutôt que d'utiliser le moins de machines possibles. Concrètement, on trie les machines dans l'ordre croissant des volumes, les processus dans l'ordre décroissant. On affecte le premier processus p_1 à la première machine possible m_i. Pour le processus suivant, on commence par tester les machines à partir de m_{i+1}, de manière circulaire.

La contrainte de conflits est gérée simplement : un processus n'est affecté à une machine que si son service n'est pas déjà présent. Pour la contrainte de dépendance, on se restreint à affecter les processus dans le voisinage de leur machine initiale. En plus de satisfaire automatiquement les contraintes de dépendances, cette restriction nous permet de décomposer le problème en problèmes plus petits (un problème par voisinage) et indépendants si on néglige les contrainte de répartition (spread constraints, 3)

Avec ces heuristiques, nous avons obtenu des solutions réalisables (sans prendre en compte les contraintes de répartition) pour 6 des 10 instances B, en quelques secondes. Ensuite, la réparation des contraintes de répartition risquait de prendre beaucoup de temps et il n'était pas garantie qu'on réussisse à les satisfaire. Comme la contrainte principale pour cette heuristique qui ne cherche qu'une solution réalisable était la rapidité, nous avons dû l'abandonner, à cause du coût des réparations.

2 Décomposition par voisinage et PLNE

Comme expliqué en introduction, lorsqu'on fixe un voisinage et qu'on s'intéresse aux machines et processus de celui-ci dans la solution initiale, toute affectation vérifie les contraintes de dépendances. Nous avons donc décomposé le problème selon les voisinages, puis, pour chacun d'eux, considéré des sous-ensembles de machines (le temps de résolution peut facilement exploser sinon) sur lesquels nous trouvons une affectation optimale des processus. Ces problèmes sont résolus en utilisant `CLP` (solver pour la programmation linéaire) et `CBC` (branch and cut), qui font partie du projet open-source `COIN-OR`[1].

Sur les plus grosses instances, il est toutefois difficile de résoudre ces problèmes à l'optimal. Nous avions donc essayé dans un premier temps de réduire le nombre de machines considérées mais les résultats n'étaient alors pas à la hauteur de la recherche locale. Puis dans un second temps, d'arrêter le branch and cut après un nombre fini de nœuds mais il était alors fréquent de ne pas obtenir de solutions réalisables. L'ajout d'heuristiques telles que le feasibility pump [3] permet parfois d'obtenir des solutions réalisables mais rarement autres que la solution initiale.

L'approche par décomposition et PLNE n'a finalement pas porté ses fruits mais plus d'effort sur l'ajout de coupes et le paramétrage du solver aurait probablement pu permettre d'améliorer sensiblement ces résultats.

3 Recherche locale

Nous disposons de deux types de mouvements : le déplacement (un processus est déplacé sur une autre machine) et l'échange (deux processus sont réassignés, l'un à la machine de l'autre). Nous imposons que ces déplacements ne soient réalisés que lorsque la nouvelle solution est réalisable et de coût inférieur à la précédente. Afin que l'algorithme soit performant, nous calculons seulement les variations du coût d'une solution à l'autre. De plus, pour déterminer si la solution voisine est réalisable, nous exploitons les invariants du mouvement. On navigue ainsi d'une solution réalisable à une autre, pouvant à tout moment interrompre le programme pour récupérer la meilleure.

Lors des qualifications, nous avions utilisé ces mouvements dans le cadre d'une recherche locale de type GRASP [2]. On faisait alors de très nombreux départs depuis la solution initiale que nous optimisions jusqu'à arriver à un puit pour l'un ou l'autre de nos mouvements avant

1. http ://www.coin-or.org/

de recommencer. Les mouvements étaient choisis aléatoirement afin d'effectuer le plus de départs possibles et de varier les solutions obtenues.

Toutefois, cette approche n'était plus la meilleure pour les nouvelles instances. En effet, leurs tailles sont telles que nous ne pouvons effectuer que peu de recherches complètes. Il est donc essentiel de guider la recherche afin que celle-ci nous permette d'obtenir les meilleurs résultats possibles en un seul départ. Comme expliqué en introduction, le coût le plus durablement optimisable pour la recherche locale est le coût de charge. Nous pouvons donc guider la recherche selon ce critère afin de le réduire en priorité.

Dans un premier temps, on trie les machines par ordre décroissants des valeurs de leurs coûts de charge signés (négatif si le seuil de charge n'est pas atteint). Puis à chaque itération, on essaye de déplacer un des processus des machines dont le coût de charge est élevé vers des machines dont le coût est négatif (ou au moins plus petit). Afin de ne pas trop augmenter les autres coûts, on impose tout de même qu'un mouvement ne soit effectué que s'il fait diminuer le coût total de la solution. Cette première étape permet de réduire significativement les coûts en moins de 30s. Cependant, comme on empêche les déplacements d'une machine de moindre coût vers une machine de coût plus élevé, la solution est encore optimisable par déplacements et échanges aléatoires. Ce que l'on effectue dans le second temps.

4 Le programme

Le programme final est open source et n'utilise que des composants open-source. Il sera prochainement publié, probablement sur GitHub [2]. Les bibliothèques utilisées pour le rendu final sont `Boost` [3] et la `STL`. Le programme rendu n'utilise que des mouvements de recherche locale bien que les autres mouvements soient implémentés et fonctionnels.

Le programme fourni répond aux spécifications requises. Il peut être compilé sous linux à l'aide de la commande `make` et l'exécutable se trouve dans le répertoire `bin`.

5 Perspectives

Les heuristiques implémentées sont sous-optimales mais présentent l'avantage de passer à l'échelle. Nous avions beaucoup d'espoir pour la décomposition en sous-problème et l'utilisation de la PLNE. Notre approche pourrait être améliorée par l'ajout de coupes au solver et un affinage des paramètres en fonction de la taille de l'instance considérée. La programmation par contrainte pourrait peut être également remplacer avantageusement la PLNE et permettre d'obtenir plus facilement des solutions réalisables sur les instances des sous problèmes.

On pourrait également implémenter d'autres mouvements pour la recherche locale ou permettre de dégrader la solution lorsqu'on est proche d'être bloqués.

Références

[1] E.H.L. Aarts and J.K. Lenstra. *Local search in combinatorial optimization*. Princeton Univ Pr, 2003.

[2] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2) :109–133, 1995.

[3] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming*, 104(1) :91–104, 2005.

[4] R Panigrahy, K Talwar, L Uyeda, and U Wieder. Heuristics for vecor bin packing. *Technical report*, 2011.

2. https ://github.com/
3. http ://www.boost.org/