Ordonnancement de jobs par approximations linéaires successives d'un modèle creux

Lamiel Toch, Stéphane Chrétien, Laurent Philippe, Jean-Marc Nicod, Veronika Rehn-Sonigo

Université de Franche-Comté,
Département Informatique des Systèmes Complexes - Institut Femto-St, Besançon, FRANCE
lamiel.toch@femto-st.fr

Résumé
Dans cet article nous traitons le problème de l'ordonnancement d'une collection de jobs indépendants dans une machine parallèle de type cluster. À la fois les jobs rigides et les jobs moldables sont traités. Le but est de minimiser le makespan de l'ordonnancement. Ce problème est connu pour être NP-Difficile et différentes heuristiques ont déjà été proposées. Nous introduisons une nouvelle approche qui repose sur des approximations linéaires successives. L'idée est de relaxer un programme linéaire en nombres entiers et d'utiliser la linéarisation de la norme ℓ_p dans la fonction objectif pour forcer les valeurs de la solution à tendre vers des nombres entiers. Nous avons conçu différents algorithmes reposant sur les approximations linéaires successives et nous les comparons avec l'algorithme de liste classique LTF (*Largest Task First*) pour montrer que cette approche donne de bons résultats. La contribution de ces travaux tient en la conception de ces algorithmes et l'intégration de la méthode mathématique dans le monde de l'ordonnancement.

Mots-clés : ordonnancement, programme linéaire, approximation successive

1. Introduction

De nos jours les clusters sont très répandus dans le monde scientifique, dans les universités, les centres de recherche qui utulisent le calcul haute performance (HPC). Ils sont utilisés pour effectuer des simulations de tout genre : simulations physiques (aérodynamique, nucléaire, et autres), prévisions météorologiques (superordinateur K de Fujitsu), application à la médecine ou encore à la génétique (Blue Gene). Dans un centre de calcul, à cause du coût important d'un cluster, celui-ci est généralement partagé entre plusieurs utilisateurs. Ainsi plusieurs applications parallèles ou séquentielles, appelées *jobs*, s'exécutent en même temps sur le même cluster. Ordonnancer un ensemble de jobs parallèles sur un cluster en connaissant leurs caractéristiques est un problème qualifié d'*off-line* qui a été très étudié pour les jobs séquentiels [9] et parallèles [8].
Le problème d'ordonnancement *off-line* que nous considérons ici dépend des caractéristiques des jobs. Dans la littérature, les chercheurs distinguent trois types de jobs parallèles. Les jobs *rigides* [11] s'exécutent avec exactement le nombre de processeurs prévu à la soumission. Les jobs *moldables* dont le modèle a été introduit par Turek et al. dans [14], peuvent s'exécuter avec un nombre différent de processeurs que celui prévu à la soumission. Celui-ci ne peut pas changer en cours d'exécution. Les jobs *malléables* [12] peuvent modifier le nombre de processeurs qui leur sont alloués au cours de leur exécution. Le modèle des jobs rigides peut facilement être utilisé dans la plupart des cas pour ordonnancer des jobs parallèles. La virtualisation des processeurs peut être une solution transparente afin de rendre moldable n'importe quelle application parallèle, comme nous le montrons dans [13]. Appliquer la virtualisation avec les jobs *malléables* est probablement plus difficile car cela demande des migrations de machines virtuelles. C'est pourquoi, nous nous concentrons sur l'étude du problème d'ordonnancement de jobs rigides et moldables.
Minimiser le *makespan* d'un ordonnancement de jobs rigides ou moldables sur des ressources de calcul homogènes est un problème NP-Difficile. Du et Leung l'ont démontré pour les jobs rigides dans [5] et Dutot et al. l'ont démontré pour les jobs moldables dans [6].

Plusieurs travaux de recherche ont été menés sur la base d'heuristiques donnant des solutions efficaces mais sous optimales. Dans [2] Amoura et al. étudient le problème d'ordonnancement de jobs parallèles rigides *off-line* afin de minimiser le *makespan*. In [7], Dutot et al. considèrent le problème d'ordonnancement de jobs moldables avec comme objectif de minimiser le *makespan*. Les auteurs présentent des heuristiques aux performances garanties et ils montrent que l'heuristique *Largest Task First* (LTF) est la meilleure pour minimiser le *makespan*.
La contribution de notre travail tient en la proposition d'une nouvelle approche pour ordonnancer une collection de jobs rigides ou moldables en utilisant des approximations linéaires successives fondées sur la norme ℓ_p. À notre connaissance, il n'existe pas de travaux utilisant cette approche prometteuse inspirée des outils pour résoudre le problème de récupération creuse du monde des statistiques et du traitement du signal.
Le contenu de ce papier est structuré de la manière suivante. Le paragraphe 2 décrit le problème et le modèle des jobs moldables. Nous présentons au paragraphe 3 le principe de la méthode exploitant le caractère creux de notre problème. Nous montrons au paragraphe 4 comment nous avons adapté la méthode à notre problème. Au paragraphe 6 nous comparons notre technique avec l'heuristique LTF développé par Dutot et al. [7] dans trois séries d'expériences afin d'évaluer les performances de notre approche. Enfin nous concluons et donnons des perspectives pour les travaux futurs au paragraphe 7.

2. Le problème

Dans ce paragraphe nous définissons le problème de manière formelle. Nous considérons le problème d'ordonnancement d'une collection de n jobs parallèles indépendants. Nous nous intéressons aux cas des jobs rigides et moldables.
Dans un cluster homogène chaque nœud est constitué de processeurs identiques qui sont eux-même constitués de cœurs identiques. Les politiques d'ordonnancement utilisées dans la plupart des clusters homogènes ne tiennent pas compte de la répartition géographique des cœurs alloués à un job parallèle. Pour cette raison, par la suite, nous ne considérons que le nombre de cœurs alloués qui sont assimilables à des unités de calcul que nous notons *PEs* (*Processing Elements*) [8].
Ainsi les jobs considérés sont exécutés dans un cluster homogène de nœuds distribués ou sur un ordinateur à mémoire partagée ou distribuée. Nous notons m le nombre de PEs disponibles dans la plate-forme sur laquelle les jobs parallèles s'exécutent.
Les jobs rigides se caractérisent par un temps d'exécution fixé ainsi qu'un nombre donné de PEs requis, c'est-à-dire que le job ne s'exécute avec ni plus ni moins de PEs que prévu au départ. Chaque job rigide i est défini par son nombre de PEs requis $\mathtt{reqproc}_i$ et sa durée d'exécution $\mathtt{reqtime}_i$.
Un job moldable peut être exécuté sur différents nombres de PEs, mais ce nombre est fixé une fois pour toute dès l'exécution du job et ne peut être changé durant celle-ci. Les jobs moldables considérés respectent le modèle défini dans [7]. Soit $\mathtt{reqtime}_i$ la durée du job i qui demande au plus $\mathtt{reqproc}_i$ PEs. Soit $t_i(n)$ la durée du job i si n PEs sont alloués au job i. La relation entre la durée d'un job i et son nombre de PEs alloués est : $\forall i,\ \forall n \leq \mathtt{reqproc}_i,\ t_i(n) = \left\lceil \frac{\mathtt{reqproc}_i}{n} \right\rceil \mathtt{reqtime}_i$. Notre objectif est de minimiser le temps de complétion du dernier job (*makespan*) de l'ordonnancement. Selon la classification pour les problèmes d'ordonnancement $\alpha|\beta|\gamma$ (plate-forme | application | critère à optimiser) donné par Graham dans [10], ce problème se note P|jobs parallèles|C_{max}.

3. Approche creuse pour promouvoir les pénalisations avec des linéarisations successives

La méthode d'optimisation du makespan d'un ordonnancement d'une collection de jobs parallèles que nous proposons repose sur deux étapes principales. D'abord nous formulons le problème sous forme d'un programme linéaire en nombres entiers. Étant donné sa taille ce programme ne peut pas être résolu sous cette forme par les solveurs existants. Nous le relaxons donc et nous mettons en œuvre une approche creuse promouvant la pénalisation, afin de se rapprocher de solutions à valeurs presque entières. Nous désignons par le terme *sparsité* le caractère creux du problème. Dans ce problème le vecteur d'inconnues est très creux. Un vecteur est creux s'il contient "beaucoup" de zéros. Comme cette sparsité implique la minimisation d'une fonction objectif non linéaire nous utilisons des approximations par linéarisations successives afin de linéariser la fonction objectif. Dans ce paragraphe nous détaillons les

principales étapes de la méthode.

3.1. Sparsité pour promouvoir les pénalisations

Des travaux récents sur le problème de récupération creuse (an anglais *sparse recovery problem*) en statistiques et dans le traitement du signal ont attiré l'attention générale sur le fait qu'utiliser des pénalisations non différentiables telles que la norme ℓ_p avec $0 < p < 1$ peut être un outil pour résoudre des problèmes combinatoires. Ainsi Chartrand [3] propose la procédure suivante, avec x comme solution d'un problème combinatoire creux : $\min_x \|x\|_p$.
La norme ℓ_p n'est cependant pas linéaire et le problème ne peut donc être résolu par un solveur linéaire. Il est cependant possible d'utiliser des programmes linéaires successifs pour linéariser le problème. Cette approche repose sur l'injection de solutions trouvées à chaque itération dans la fonction objectif linéarisée du programme linéaire à l'itération suivante. Cette linéarisation est réalisée au moyen de l'opérateur de gradient.

3.2. Approximation linéaire

En physique et mathématiques on parle de développement limité d'ordre 1 ou d'approximation linéaire d'une fonction f définie sur $\mathbb{R}^n$, quand on souhaite approximer f au voisinage d'un point x_0 par une fonction linéaire. Si f est différentiable en un point x_0 une approximation linéaire de f s'exprime de la manière suivante : $f(x_0 + h) = f(x_0) + \langle \nabla f(x_0), h \rangle + o(h)$ où les chevrons représentent le produit scalaire et ∇ est le gradient de f.

4. Application de la méthode sur le problème d'ordonnancement

Dans ce paragraphe nous appliquons la méthode sur le problème d'ordonnancement d'une collection de jobs parallèles. D'abord cela implique de définir une représentation creuse du problème. Nous appliquons ensuite les deux étapes de la méthode d'approximation linéaire sur le modèle creux ainsi défini.

4.1. Formulation du problème comme un programme linéaire en nombres entiers

Comme il était indiqué dans la formulation du problème d'ordonnancement, pour les jobs rigides, une des solutions du programme linéaire doit fournir uniquement le début d'exécution de chaque job car ceux-ci ont un nombre de PEs et des durées constants. Pour les jobs moldables, les durées d'exécutions dépendent du nombre de PEs qui leurs sont alloués. Donc l'ordonnancement des jobs moldables est complètement déterminé si nous connaissons leur date de début d'exécution et le nombre de PEs qui leur sont alloués. Nous appelons *configuration* d'un job le nombre de PEs alloués à ce job. Nous appelons *position* d'un job, sa position donnée directement par sa date de début d'exécution dans une échelle temporelle discrétisée. Enfin, nous appelons *slot* le couple (*configuration*, *position*).
Créons une liste de slots (configurations, positions) pour chaque job. L'idée consiste à créer un vecteur x_i pour chaque job i. Chaque composante $x_{i,s}$ du vecteur x_i est une variable binaire qui indique si le slot s du job i est choisi ou non. Ensuite nous fixons un horizon de temps T et nous laissons un programme linéaire trouver une solution. Nous réduisons progressivement l'horizon de temps T jusqu'à ce que le programme linéaire ne trouve plus de solution. Nous définissons le vecteur x comme la concaténation du vecteur x_i de chaque job i.

5. Formulation du problème en un programme linéaire en nombres entiers

Pour formuler le problème nous utilisons les notations suivantes : n est le nombre de jobs à ordonnancer, m est le nombre de PEs disponibles, T est l'horizon du temps, $\mathrm{proc}_{i,j}$ est le nombre de PEs pour la configuration j du job i, nconf_i est le nombre de configurations possibles pour le job i, nslot_i est le nombre de slots pour le job i, $C_{i,s}$ est la configuration du job i donnée par le slot s, $\mathrm{run}_{i,s,t}$ indique si le job i avec le slot s tourne à l'instant t, $\mathrm{reqprocs}_i$ esy le nombre de PEs demandés au départ par le job i et $\mathrm{reqtime}_i$ est la durée d'exécution requise par le job i. Ces valeurs sont des constantes du problème.
La formulation du problème ne contient qu'une variable indicée, $x_{i,s}$, qui est une variable binaire : elle indique si le slot s du job i est utilisé. Cette formulation exprime le problème comme un programme linéaire mais sans fonction objectif car il suffit de trouver une solution. Comme nous l'avons dit précédemment, nous travaillons par itérations successives de résolutions du programme linéaire en réduisant

progessivement l'horizon T, jusqu'à ce que celui-ci ne trouve plus de solution.
Les contraintes qui caractérisent le problème sont les suivantes :

$$\forall 1 \leq i \leq n, \quad \sum_{s=1}^{s=nslot_i} x_{i,s} = 1 \tag{1}$$

$$\forall 1 \leq t \leq T, \sum_{i=1}^{i=n} \sum_{s=1}^{s=nslot_i} x_{i,s} \times run_{i,s,t} \times proc_{i,C_{i,s}} \leq m \tag{2}$$

La contrainte (1) impose qu'un seul slot s soit choisi pour un job i. La contrainte (2) impose qu'à chaque instant t l'ensemble des jobs qui tournent ne consomment pas plus que ce qui est disponible.

5.1. Relaxation du programme linéaire

Ce programme linéaire en nombres entiers est exponentiel en temps d'exécution. C'est pourquoi nous effectuons une relaxation. Nous transformons la variable binaire $x_{i,s}$ en une variable rationnelle avec $0 \leq x_{i,s} \leq 1$. Prenons maintenant une représentation matricielle et vectorielle des données et des variables. Chaque job i a un vecteur x_i dont chaque composante indique si le slot s correspondant est choisi. En concaténant chaque vecteur x_i, on obtient un vecteur x. Nous souhaitons que chaque vecteur x_i soit très creux. Idéalement chaque vecteur x_i doit contenir exactement un "1". Ainsi nous souhaitons minimiser $\sum_i \|x_i\|_p$.

5.2. Linéarisation

Nous définissons la fonction f de la manière suivante : $\forall i, \; f(x_i) = \|x_i\|_p = \left(\sum_{j=1}^{j=nslot_i} x_{i,j}^p\right)^{\frac{1}{p}}$. En utilisant le résultat $x_i^{(iter)}$ de l'itération précédente nous linéarisons la norme ℓ_p en posant $h = x_i - x_i^{(iter)}$:

$$f(x_i) = f(x_i^{(iter)}) + \langle \nabla f(x_i^{(iter)}), x_i - x_i^{(iter)} \rangle + o(x_i - x_i^{(iter)})$$

Comme f n'est pas différentiable en 0, posons $g \in \partial f$ un des sous-gradients possibles de f et nous utilisons ce sous-gradient g à la place de ∇f.

$$g_{i,j} = \begin{cases} x_{i,j}^{p-1} \times f(x_i)^{1-p} & \text{si } x_{i,j} \neq 0 \\ 0 & \text{sinon.} \end{cases} \tag{3}$$

Notre fonction objectif dont le but est de forcer la solution à contenir le maximum de 0 et le bon nombre de 1 est $\sum_i f(x_i)$ sous les contraintes (1) et (2). La linéarisation de cette fonction permet alors une résolution par un solveur de programme linéaire.

5.3. Algorithme

La méthode de résolution est implémentée dans l'algorithme 1 où x est initialisé au vecteur nul. À la ligne 1, nous calculons une borne inférieure pour le makespan, égale au maximum entre la durée du job le plus long et $\frac{\sum_i reqproc_i \times reqtime_i}{m}$. L'horizon de temps T est initialisé au makespan de l'algorithme LTF. Si le programme linéaire $\mathcal{LP}$ trouve une solution satisfaisante (ligne 8), l'algorithme réduit l'horizon de temps (ligne 10) jusqu'à ce qu'il le ne puisse plus (line 20) avant $maxIter$ itérations. S'il ne trouve pas de solution satisfaisante avec $T = listMakespan$ avant $maxIter$ itérations (ligne 14), il augmente l'horizon de temps T (ligne 18). Pour un horizon de temps T donné il met à jour la fonction objectif du programme linéaire à chaque itération (ligne 6) selon la formule de linéarisation.
Durant la phase d'expérimentation un problème est apparu dans la résolution du programme linéaire. Des solutions satisfaisantes pour l'algorithme 1 sont uniquement détectées (à la ligne 8) si tous les jobs i ont leur vecteur x_i avec exactement un "1", car l'algorithme a été conçu pour obtenir des solutions exactes. En fait, pour un horizon de temps T donné, les approximations par les programmes linéaires successifs arrivent à trouver un ordonnancement pour la plupart des jobs de la collection mais ils laissent quelques jobs j de la collection dans un état d'ordonnancement flou. C'est-à-dire que les vecteurs x_i contiennent exactement un "1" alors que les vecteurs x_j contiennent des valeurs rationnelles entre 0 et

Algorithme 1: Un algorithme d'approximation par linéarisations successives

1 $\mathrm{lb} \leftarrow$ borne inférieure du makespan
2 $\mathrm{sched} \leftarrow$ calculer un ordonnancement avec l'algorithme LTF ; $\mathrm{listMakespan} \leftarrow \mathrm{makespan}(\mathrm{sched})$; $T \leftarrow \mathrm{listMakespan}$; $\mathrm{end} \leftarrow \mathrm{faux}$; $\mathrm{incT} \leftarrow \mathrm{faux}$
3 **tant que** $T > \mathrm{lb}$ *et non* end **faire**
4     $\mathrm{proc} \leftarrow$ calculer les configurations des jobs $\mathcal{J}$; $\mathrm{run} \leftarrow$ calculer tous les slots possibles avec $(\mathcal{J}, m, T)$; $\mathrm{iter} \leftarrow 1$; $\mathrm{found} \leftarrow \mathrm{faux}$; $\forall i, k, x_{i,k}^{(\mathrm{iter})} \leftarrow 0$
5   **tant que** $\mathrm{iter} < \mathrm{maxIter}$ *et non* found **faire**
6   mettre la fonction objectif de $\mathcal{LP}(\mathcal{J}, m, T, \mathrm{proc}, \mathrm{run})$ à $\sum_i^{|\mathcal{J}|} f(x_i^{(\mathrm{iter})}) + \langle \nabla f(x_i^{(\mathrm{iter})}), x_i - x_i^{(\mathrm{iter})} \rangle$
7         $x \leftarrow$ exécuter $\mathcal{LP}(\mathcal{J}, m, T, \mathrm{proc}, \mathrm{run})$
8   **si** $\forall i,\ x_i$ *contient exactement un "1"* **alors**
9             $\mathrm{sched} \leftarrow$ convertir en un ordonnancement $(x, \mathrm{proc}, \mathrm{run})$
10             $T \leftarrow \mathrm{makespan}(\mathrm{sched})$; $T \leftarrow T - 1$; $\mathrm{found} \leftarrow \mathrm{vrai}$
11   **si** $\mathrm{incT} = \mathrm{vrai}$ **alors**
12                 $\mathrm{end} \leftarrow \mathrm{vrai}$
13         $\forall i, k, x_{i,k}^{(\mathrm{iter})} \leftarrow x_{i,k}$; $\mathrm{iter} \leftarrow \mathrm{iter} + 1$
14   **si** *non* found **alors**
15   **si** $T = \mathrm{listMakespan}$ **alors**
16             $\mathrm{incT} \leftarrow \mathrm{vrai}$
17   **si** *incT = vrai* **alors**
18             $T \leftarrow T + 1$
19   **sinon**
20             $\mathrm{end} \leftarrow \mathrm{vrai}$
21 **retourner** sched

1. Dans ce cas, l'algorithme continue, même si une composante de x_j est proche de 1, jusqu'à ce que $\mathrm{maxIter}$ itérations soient effectuées. En procédant ainsi, les temps de calcul étaient très longs et des solutions trouvées étaient inefficaces.

Par conséquent, nous avons modifié l'algorithme 1 et son critère de détection de solution de la ligne 8 comme suit : quand le programme linéaire trouve un ordonnancement exact pour les jobs i, dont les x_i contiennent exactement un "1", on garde ces jobs i dans l'ordonnancement et on ordonnance le reste des jobs pour lesquels le programme linéaire a trouvé un ordonnancement flou, grâce à l'algorithme LTF. Si une solution plus courte que l'horizon de temps T est trouvée, alors la variable found est mise à vrai et l'algorithme continue à itérer. Dans la suite nous utilisons cette variante pour l'évaluation de notre approche.

6. Évaluation de performances

La solution proposée étant heuristique, elle doit être évaluée. Pour réaliser cette évaluation, nous utilisons des collections de jobs, appelées *workload*. Ainsi l'algorithme prend en entrée un workload et il rend un ordonnancement, donc un makespan. Nous utilisons les bibliothèques de Gurobi [1] pour résoudre les programmes linéaires. Les calculs ont été réalisés sur les ressources de calcul du Mésocentre de Calcul de France-Comté.

Afin d'observer le comportement de l'algorithme proposé et d'évaluer ses performances, nous mettons en place une suite d'expériences avec différentes lois de probabilité générant aléatoirement le nombre de PEs requis pour l'ensemble des jobs. Ces lois sont une loi gaussienne, une loi mélangeant deux gaussiennes et une loi de Cauchy. Nous définissons la granularité d'un workload comme étant le rapport entre la durée du job le plus long sur la durée du job le plus court.

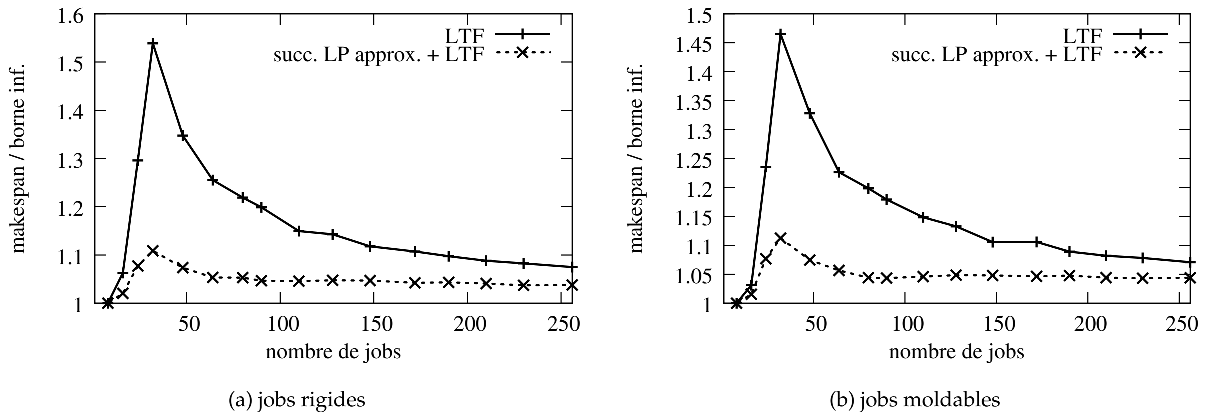
Nous présentons dans ce paragraphe les résultats obtenus avec l'algorithme sur des machines simulées de 128 PEs. Les paramètres des expériences sont les suivants. La granularité est de 10, $p = 0.1$ et $\mathrm{maxIter} = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 40 avec les différentes lois de probabilité. Nous comparons ici les performances avec l'algorithme LTF. Dans les figures qui suivent les algorithmes sont notés

succ. LP approx + LTF pour notre algorithme et *LTF* pour l'algorithme LTF. Nous évaluons les performances pour le cas à la fois rigide et moldable. Pour d'autres expériences, nous invitons le lecteur à se référer au travail que nous avons présenté dans [4].

6.1. Loi gaussienne

Pour les expériences décrites dans ce paragraphe, nous utilisons une loi gaussienne pour générer le nombre requis de PEs par les jobs.

La loi gaussienne a une fonction de densité égale à : $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$ où le paramètre μ est la moyenne et σ est l'écart-type. L'avantage d'une loi gaussienne est qu'elle favorise le mélange de jobs dont le niveau de parallélisme peut être contrôlé. Les jobs requièrent beaucoup de PEs en imposant que la moyenne μ soit importante ou bien peu de PEs en imposant que μ soit petite. L'inconvénient d'une loi gaussienne est que les valeurs de PEs générées sont centrées autour de cette moyenne μ, ce qui n'est pas toujours assez réaliste.
Le paramètre σ (écart-type) de la gaussienne est de 8. Le paramètre μ (moyenne) de la gaussienne est de 4. Donc il y a davantage de jobs qui requièrent un petit nombre de PEs que de jobs qui demandent beaucoup de PEs. Les figures 1a et 1b montrent le ratio du makespan à sa borne inférieure en fonction du nombre de jobs rigides et moldables pour cette distribution. Nous remarquons que les performances de notre algorithme sont bien meilleures que celles de LTF, pour n'importe quelle taille de collection.



(a) jobs rigides

(b) jobs moldables

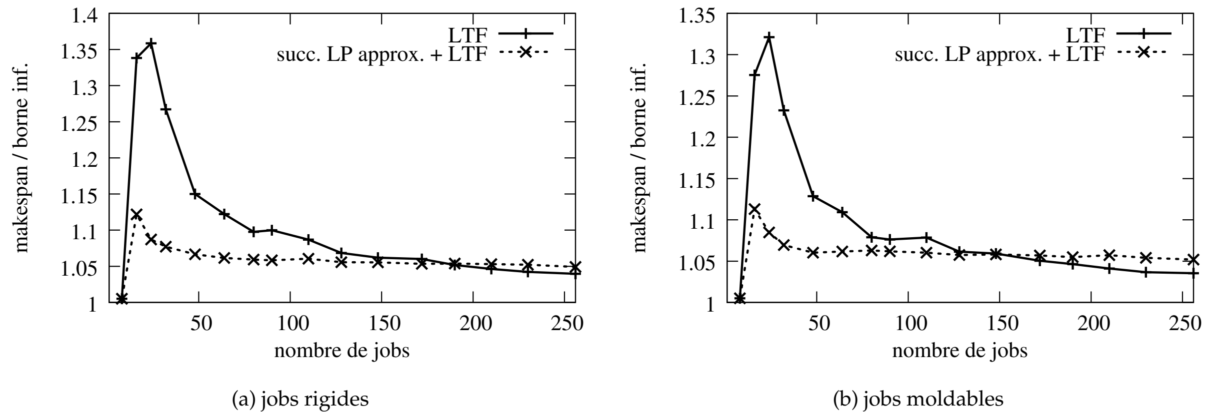
FIGURE 1 – Performances de l'algorithme avec 128 PEs, loi gaussienne : $\mu = 4$ et $\sigma = 8$

6.2. Mélange de lois de probabilité

Dans une autre série d'expériences nous souhaitons observer le comportement des algorithmes testés en mélangeant les lois de probabilité du nombres de PEs requis par les jobs.

La fonction de mélange de K densités de probabilités est définie comme suit : $f(x) = \sum_{k=1}^{K} \pi_k \Phi_k(x)$ où $\Phi_k(x)$ est la fonction de densité de la loi k, π_k est le poids de la loi k et $\sum_{k=1}^{K} \pi_k = 1$.
L'avantage d'utiliser un mélange de deux gaussiennes est que l'on peut générer des valeurs qui se répartissent autour de deux moyennes. Ainsi nous pouvons mieux contrôler les valeurs générées de PEs afin de simuler une distribution de jobs qui requièrent un petit nombre de PEs et d'autres qui requièrent un grand nombre de PEs. L'inconvénient d'un mélange de deux gaussiennes est que les valeurs se répartissent autour de deux moyennes, ce qui produit des valeurs de PEs parfois peu réalistes.
Afin de simuler une distribution de jobs qui requièrent un petit nombre de PEs et d'autres qui requièrent un grand nombre de PEs, le paramètre μ_1 de la gaussienne 1 est de 4 et le paramètre μ_2 de la gaussienne 2 est de 20. Pour favoriser la diversité des valeurs, le paramètre σ_1 de la gaussienne 1 est de 8 et le paramètre σ_2 de la gaussienne 2 est de 8.
Les figures 2a et 2b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. À partir de 150 jobs notre algorithme se fait rattraper par l'algorithme

LTF. En fait notre algorithme ne donne pas de moins bons ordonnancements, mais c'est l'algorithme LTF qui profite du nombre de jobs importants pour combler les trous.



(a) jobs rigides

(b) jobs moldables

FIGURE 2 – Performances de l'algorithme avec 128 PEs, mélanges de deux gaussiennes : $\pi_1 = \pi_2 = 0.5$ $\mu_1 = 4$, $\mu_2 = 20$ et $\sigma_1 = \sigma_2 = 8$

Il faut noter que, quand le nombre de jobs à ordonnancer devient grand, l'algorithme LTF obtient des ordonnancements pour lesquels le ratio du makespan à sa borne inférieure diminue. À partir d'un nombre de jobs à ordonnancer au dessus de la centaine, ce ratio devient quasiment constant, et d'après les courbes, il avoisine environ 1.05. Le ratio entre le makespan et la borne inférieure étant à ce moment très petit la marge de progression devient alors quasi nulle.

6.3. Loi de Cauchy

Nous utilisons une loi de Cauchy pour générer aléatoirement le nombre de PEs requis pour l'ensemble des jobs du workload.

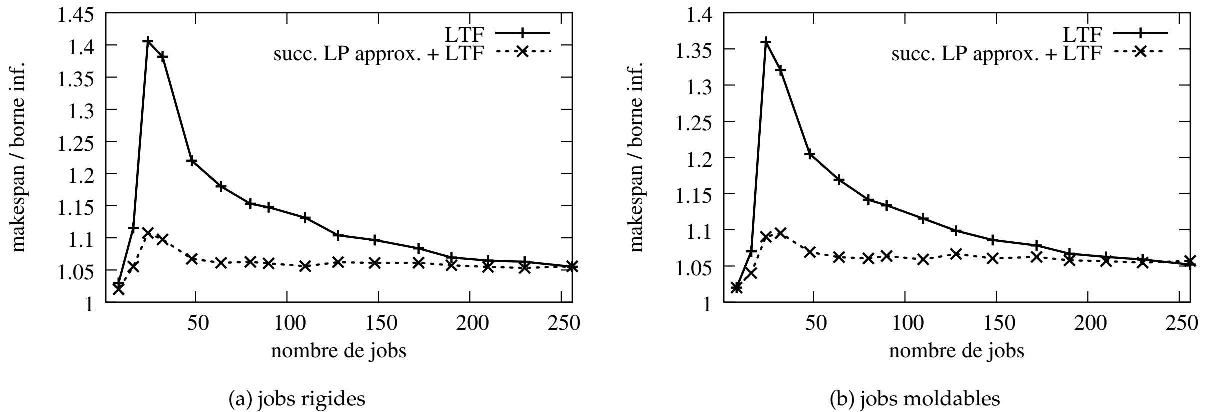
La loi de Cauchy a une fonction de densité égale à : $f(x) = \frac{1}{\pi}\left(\frac{\gamma}{\gamma^2+(x-m)^2}\right)$ où le paramètre m est la médiane et γ un facteur d'échelle. La loi de Cauchy a la propriété de ne possèder ni espérance ni variance. Du fait de cette singularité, nous souhaitons connaître le comportement de notre algorithme avec cette loi.

Le premier avantage d'une loi de Cauchy pour générer des valeurs de PEs requis pour les jobs est que les valeurs ne sont pas regroupées autour d'une moyenne contrairement à une loi gaussienne. Le second avantage est que nous pouvons tout de même contrôler la répartition du niveau de parallélisme des jobs grâce à la médiane. Ainsi les valeurs de PEs requis pour les jobs sont plus réalistes.

Le paramètre γ (facteur d'échelle) de la loi est de 8. Le paramètre m (médiane) de la loi est de 4. Les figures 3a et 3b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. On remarque que l'algorithme donne de meilleurs résultats que l'algorithme LTF, néanmoins dès que le nombre de jobs à ordonnancer est important l'algorithme LTF rejoint notre algorithme.

7. Conclusion

Dans cet article nous avons proposé un algorithme reposant sur des approximations linéaires successives d'un modèle creux. Quelque soit le type de distribution, les performances de l'algorithme proposé sont très bonnes pour un nombre de jobs, et un nombre de processeurs, en dessous de la centaine. Nous avons montré que ces performances ne dépendent pas du type de la distribution utilisée pour générer aléatoirement le nombre de PEs requis pour les jobs. Pour des travaux futurs, nous planifions d'appliquer la méthode sur le problème d'ordonnancement de jobs *online*, sur la base de fenêtres d'ordonnancement dont nous chercherons à optimiser le makespan. Ce problème correspont en effet bien au domaine sur lequel notre algorithme permet un gain significatif de performances.



(a) jobs rigides

(b) jobs moldables

FIGURE 3 – Performances de l'algorithme avec 128 PEs, loi de Cauchy : $m = 4$ et $\gamma = 8$

Bibliographie

1. Gurobi. `http://www.gurobi.com`.
2. Amoura (A. K.), Bampis (E.), Kenyon (C.) et Manoussakis (Y.). – Scheduling independent multiprocessor tasks. *Algorithmica*, vol. 32, n2, 2002, pp. 247–261.
3. Chartrand (R.) et Yin (W.). – Iteratively reweighted algorithms for compressive sensing. *In : 33rd International Conference on Acoustics, Speech, and Signal Processing (ICASSP).*
4. Chrétien (S.), Nicod (J.-M.), Philippe (L.), Rehn-Sonigo (V.) et Toch (L.). – Job scheduling using successive linear programming approximations of a sparse model. *In : Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*. pp. 116–127. – Springer.
5. Du (J.) et Leung (J. Y.-T.). – Complexity of scheduling parallel task systems. *SIAM J. Discret. Math.*, vol. 2, n4, novembre 1989, pp. 473–487.
6. Dutot (P.-F.), Eyraud (L.), Mounié (G.) et Trystram (D.). – Bi-criteria algorithm for scheduling jobs on cluster platforms. *In : Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 125–132. – New York, NY, USA, 2004.
7. Dutot (P.-F.), Goldman (A.), Kon (F.) et Netto (M.). – Scheduling moldable BSP tasks. *In : 11th JSSPP*, pp. 157–172. – Cambridge, MA, USA, 2005.
8. Feitelson (D. G.). – Job scheduling in multiprogrammed parallel systems, 1997.
9. Garey (M. R.) et Johnson (D. S.). – *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. – New York, NY, USA, W. H. Freeman & Co., 1990.
10. Graham (R.) et al. – Optimization and approximation in deterministic sequencing and scheduling : a survey. *Ann. Discrete Math.*, 1979, pp. 287–326.
11. Lublin (U.) et Feitelson (D. G.). – The workload on parallel supercomputers : Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, vol. 63, 2003, pp. 1105–1122.
12. Mounie (G.), Rapine (C.) et Trystram (D.). – Efficient approximation algorithms for scheduling malleable tasks. *In : Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*. pp. 23–32. – ACM.
13. Nicod (J.-M.), Philippe (L.), Rehn-Sonigo (V.) et Toch (L.). – Using virtualization and job folding for batch scheduling. *In : ISPDC'2011, 10th Int. Symposium on Parallel and Distributed Computing*. pp. 39–41. – Cluj-Napoca, Romania, juillet 2011.
14. Turek (J.), Wolf (J. L.) et Yu (P. S.). – Approximate algorithms scheduling parallelizable tasks. *In : Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. pp. 323–332. – New York, NY, USA, 1992.