Compression d'ordonnancements

Jérémie Dumas
École Normale Supérieure de Lyon

Janvier 2012

Le présent rapport se propose d'exposer la problématique soulevé par le papier de Bozdağ et al. [BOC09], à propos de la compression d'ordonnancements sur un DAG. Nous présenterons donc simplement les deux algorithmes détaillés dans [BOC09], et discuterons du cadre général de travail proposé pour résoudre le problème de l'attribution de processeur à un ordonnancement quelconque correspondant à un DAG.

Table des matières

1 Généralités sur l'ordonnancement de DAG

1.1 Introduction

Une branche importante de l'algorithmique distribuée se concentre sur l'ordonnancement de graphes de tâches, qui constituent un modèle pratique pour répartir un gros calcul sur plusieurs processeurs : le travail est découpé en plusieurs petites tâches, dépendantes les unes des autres, et sont ensuite envoyées sur les ordinateurs qui vont bien. Les tâches ainsi décrites forment un graphe acyclique orienté (*DAG* en anglais), où les sommets représentent les tâches à exécuter et les arêtes les dépendances entre les tâches.

Ordonnancer les tâches d'un DAG consiste donc à les répartir sur différents processeurs, en respectant les contraintes de dépendances, de façon à minimiser une certaine métrique. Généralement on cherche à minimiser le *makespan* (date de fin de la dernière tâche exécutée), mais on peut aussi vouloir minimiser les communications inter-processeurs, ce qui revient à s'intéresser au nombre de processeurs nécessaire à l'ordonnancement. Dans leur article [BOC09], Bozdağ et al. font la remarque que la plupart des algorithmes existant cherchent surtout à minimiser la durée totale d'exécution, au détriment la plupart du temps du nombre de processeurs utilisés. Les trois algorithmes `PLW`, `TCSD` et `CPFD` auxquels ils se réfèrent ont par exemple besoin de plus de 800 processeurs pour effectuer une élimination gaussienne sur des matrices de taille 55 !

On comprend donc l'intérêt de minimiser le nombre de processeurs utilisés, si on veut s'attaquer à des cas réalistes, surtout quand l'algorithme qu'ils proposent arrive à réduire 70% à 90% le nombre de processeurs nécessaires à un ordonnancement proposé par l'un des algorithmes sus-cités.

1.2 Contexte et définitions

Un DAG est un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, orienté et acyclique, où $\mathcal{N}$ représente les tâches à effectuer, et $\mathcal{E}$ les dépendances entre les tâches. On se place dans un système avec processeurs homogène, donc une tâche n_t a un temps d'exécution w_t identique sur tous les processeurs. Une arête $(n_p, n_t) \in \mathcal{E}$ indique que la tâche fille n_t doit recevoir des informations de la tâche parente n_p avant de pouvoir être exécutée. Si deux tâches dépendantes sont exécutées sur le même processeur, il n'y a pas de temps de communication on peut démarrer le calcul de la tâche fille immédiatement. Si la tâche fille n_t doit recevoir des informations d'un parent n_p situé sur un autre processeur, il faudra en revanche payer un coût en temps de $c_{p,t}$ correspondant à l'envoi des données. Ici toutes les unités de calcul sont connectées entre elle, et les communications sont homogènes (on pairera le même $c_{p,t}$ partout). Un processeur peut calculer et communiquer en même temps, et l'ordonnancement est non-préemptif.

La question posée est donc, étant donné un graphe de tâches $\mathcal{G}$, trouver un ordonnancement valide sur plusieurs processeurs qui minimise par exemple le makespan à l'exécution. Le problème dans le cas général est NP-dur, et nécessite donc d'utiliser des heuristiques. On distingue alors deux familles d'algorithmes, selon qu'ils s'autorisent ou non à dupliquer une tâche ou non. L'intérêt de dupliquer une tâche sur différents processeurs et de supprimer le temps de communication en regroupant les fils d'une tâche sur le

même processeur que son parent. Les algorithmes sans duplications sont soit des ordonnancements de listes, soit basés sur des méthodes de clustering, qui tenten de regrouper les tâches communiquant le plus. Parmi les algorithmes à base de duplications, il en est qui offrent plus ou moins de flexibilité quant aux tâches condisérées pour la duplication. Cela mène donc à des compromis entre temps d'exécution de l'algorithme, et efficacité du résultat obtenu, les algorithmes avec duplications étant généralement plus coûteux mais plus performants. L'algorithme `SDS` proposé dans [BOC09] se place dans cette deuxième catégorie.

Toutefois, la partie centrale du papier de Bozdağ et al. repose sur l'élaboration d'un algorithme de *schedule compaction* (noté `SC`) qui s'applique à n'importe quel ordonnancement de DAG valide et permet de réduire le nombre de processeurs utilisé, sans augmenter le makespan initial σ. La présentation détaillée et complète de cet algorithme serait un peu fastidieux, donc nous tâcherons d'en dégager les grandes lignes et d'en exposer les idées essentielles. De même pour l'algorithme d'ordonnancement `SDS` qu'ils proposent. Mais avant cela, il nous faut introduire certaines définitions utiles.

Définitions

Une tâche d'entrée (respectivement de sortie) dans un DAG est une tâche n'ayant pas de parents (respectivement de fils) dans le graphe. Un *chemin critique* est un chemin dans le graphe tel que la somme des temps d'exécutions des tâches et des temps de communications le long du chemin soit maximum. On définit le *bottom-level* b_t d'une tâche n_t comme le poids d'un chemin critique pour le sous-DAG enraciné en n_t, et se calcule par la formule $b_t = w_t + \max_{n_t \to n_c}(c_{t,c} + b_c)$. De plus, si n_t désigne une tâche placée sur un processeur P_l, on notera $st(n_t, P_l)$ et $ft(n_t, P_l)$ les temps de début et de fin d'exécution de la tâche respectivement (*start time* et *finish time* en anglais). On notera $pos(n_t, P_l)$ le numéro de la tâche n_t dans son ordonnancement sur le processeur P_l, la première tâche exécutée ayant pour valeur 1.

Il est à noter que plusieurs copies d'une même tâche peuvent être présentes sur l'ensemble des processeurs, du fait de potentielles duplications. Si $(n_p, n_t) \in \mathcal{E}$, on dira qu'une copie de n_t est un fils local de n_p si une copie de n_p est ordonnancée avant n_t sur le même processeur P_l. Sinon n_t sera appelé un fils hors-processeur de n_p. Cela nous amène à définir le temps de fin au plus tard d'une tâche n_t, noté $lft(n_t)$ pour *latest finish time*, comme la date maximum à laquelle au moins une copie de n_t doit avoir fini de s'exécuter pour satisfaire toutes les dépendances avec ses fils hors-processeur.

Un des paradigmes de l'algorithme `SC` présenté plus loin est de choisir une copie particulière d'une tâche n_t, qui se chargera de satisfaire les dépendances hors-processeur en envoyant ses données à travers le réseau, tandis que les autres copies auront plus de flexibilité pour être déplacée, n'ayant que des dépendances avec leurs fils locaux à satisfaire. La copie de n_t ainsi choisie sera appelée *copie fixe*. On note alors $dat(n_t, n_p)$ (pour *data arrival time*) la date minimum à laquelle la tâche n_t est sûre d'avoir reçu les données de la copie fixe de n_p. Enfin, on définit le temps de début au plus tôt $est(n_t, P_l)$ d'une tâche sur le processeur P_l la date minimum à laquelle n_t aura reçu les données de tous ses parents et sera prêt à être exécuté sur P_l.

2 Les algorithmes de compression SC et SDS

2.1 Réduire le nombre de processeurs, l'algorithme SC

L'idée générale de l'algorithme SC est plutôt simple, mais sa réalisation compliquée. On procède en 3 temps : premièrement, fixer une copie pour les différentes tâches considérées ; éliminer les copies redondantes ; fusionner itérativement les processeurs selon une certaine mesure de similitude. Afin de rendre les fusions plus faciles, les deux premières phases sont là pour préparer le terrain, et déplacer déjà certaines tâches dans l'ordonnancement donné en entrée. Plus précisément, pour permettre la fusion de deux processeurs en un nouvel ordonnancement partiel P_m, on s'attachera à faciliter le remplissage des 4 conditions suivantes :

VC1 Une tâche de P_m doit recevoir les données de ses parents avant sa nouvelle date de début prévue.

VC2 Une tâche avec des fils hors-processeur doit finir suffisamment tôt pour pouvoir leur transmettre leurs données à temps.

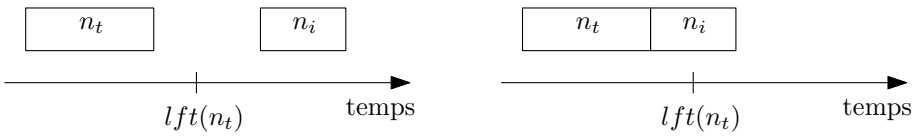
VC3 Une tâche avec des fils locaux doit être exécutée avant tous ses fils.

VC4 La durée total d'exécution des tâches sur P_m ne doit pas dépasser la durée totale de l'ordonnancement σ.

Première phase

Dans cette partie, les tâches du DAG sont parcouru par bottom-level croissant, afin d'examiner une tâche avant ses parents. Pour chaque tâche n_t, on calcule $lft(n_t)$ de la manière suivante. Soit $\mathcal{Q}(n_t)$ l'ensemble des fils hors-processeurs de n_t. si $\mathcal{Q}(n_t)$ est vide, alors n_t doit finir au plus tard à la date σ pour ne pas rallonger l'ordonnancement que l'on veut optimiser. Sinon, $lft(n_t)$ est initialisé en fonction du temps de début de ses fils hors-processeurs.

Deuxièment, il faut fixer une copie de n_t. Pour ce faire on distingue deux cas. Si pour chaque une copie de n_t, la tâche n_i qui suit commence après $lft(n_t)$ (figure 1a), on fixe la copie sur le processeur qui minimise le temps d'inactivité. Sinon (figure 1b), on choisit le processeur sans temps d'inactivité dont la tâche n_t finit au plus tôt.



(a) Minimiser l'inactivité (b) Finir au plus tôt

FIGURE 1: Fixer une copie de n_t : deux cas de figure.

Enfin, les copies des tâches sont décalées pour commencer le plus tard possible (en autorisant les sauts), afin de donner plus de marge aux fils hors-processeurs pour la réception de données.

Deuxième phase

Au cours de cette deuxième partie, il faut calculer $est(n_t, P_l)$ pour chaque copie de chaque tâche n_t, et supprimer les éventuelles copies superflues. est est calculé de manière itérative et mis à jour au cours de cette phase. Ensuite, comme on venait de décaler des tâches vers la fin, on regarde à nouveau les contraintes de données entre les copies des tâches. 3 cas se présentent pour satisfaire une contrainte de dépendance (n_p, n_t) du DAG, où on considère la copie de n_t qui est sur un processeur P_l :

- n_t est un fils hors-processeur de n_p, et aucune copie de n_p n'est ordonnancée avant n_t sur le processeur P_l. Il y a transmission inter-processeur.
- n_p apparaît avant n_t sur l'ordonnancement de P_l et est nécessaire à l'exécution de n_t. Pas de transmission inter-processeurs.
- n_t peut recevoir ses données d'une copie locale de n_p ou de la copie fixe de n_p qui se trouve sur un autre processeur. Dans ce cas, afin de supprimer n_p de P_l, il faut s'assurer que tous les fils locaux de n_p se trouvent dans cette situation là (i.e. n'ont pas besoin de la copie de n_p sur P_l pour s'exécuter au moment prévu).

Pour finir, on re-décale les copie de chaque tâche le plus tôt possible (sans sauts autorisés cette fois), et on met à jour leur lft, afin d'aider leurs fils à vérifier la règle VC1 dans la troisième phase de l'algorithme.

Troisième phase

Dernière partie de l'algorithme et sans doute la plus délicate. Il s'agit de fusionner itérativement des processeurs, et d'y réordonner les tâches communes, jusqu'à ce qu'aucune fusion ne soit plus possible. En pratique, on impose toutefois qu'au moment de fusionner deux processeurs P_l et P_k, l'un au moins doit être le résultat d'une fusion de l'itération précédente. Cette heuristique permet de borner plus aisément la complexité totale de l'algorithme.

Les paires de processeurs que l'on tente de fusionner sont sélectionnées selon la mesure de similarité suivante : soit et_l (resp. et_k) la somme des temps d'exécution des tâches sur P_l (resp. P_k), et $et_{l,k}$ la somme des temps d'exécution des tâches communes à P_l et P_k. Soit aussi σ_l (reps. σ_k) la date de fin de la dernière tâche sur P_l (resp. P_k). On pose alors $sim_{l,k} = et_{l,k} / \min(et_l, et_k)$ si $et_l + et_k - et_{l,k} \leqslant \max(\sigma_l, \sigma_k)$, et -1 sinon (il faut que l'union des tâches des deux processeurs ne prennent pas trop de temps à exécuter, i.e. ne rallonge pas l'ordonnancement déjà en place). On choisit donc d'abord les paires de processeurs ayant la plus grande similitude comme candidats à une fusion.

Le mécanisme de fusion fonctionne alors de la manière suivante. On sélectionne (en partant de la fin) les tâches de P_l ou de P_k selon certains critères, puis elles sont placées sur le nouveau processeur P_m en partant également de la fin. À tout moment, si la fusion est détectée comme étant impossible, on passe à la paire de processeur suivante. Au cours étape, les tâches redondantes sur P_l et P_k sont simplifiées, et les informations importantes (est, lft, etc.) sont mises à jour. Les ensembles $\mathcal{Q}(n_t)$ de fils hors-processeurs sont également affectés, car des communications inter-processeurs peuvent être évitées.

Si un ensemble $\mathcal{Q}(n_t)$ devient vide, on libère également la copie fixe de n_t, ce qui donnera encore plus de flexibilité aux itérations suivantes.

Ce processus est alors répété jusqu'à ce qu'il n'y ait plus de fusion qui réussisse, auquel cas l'algorithme s'arrête et renvoie un nouvel ordonnancement valide du DAG. Bozdağ et al. montrent alors dans [BOC09] que la complexité temporelle de l'algorithme `SC` est $\mathcal{O}(|\mathcal{N}|^3)$, sous les hypothèses raisonnables qu'il y ait moins de processeurs que de tâches dans le DAG, et qu'il y ait au plus une copie de chaque tâche sur chaque processeur.

2.2 Ordonnancer les tâches d'un DAG, l'algorithme `SDS`

Afin de mettre à profit l'algorithme `SC` précédemment exposé, les auteurs de [BOC09] proposent alors un algorithme d'ordonnancement destiné à être utilisé conjointement avec SC, et qui se focalise cette fois sur la diminution du makespan obtenu, laissant à `SC` le soin de réduire le nombre de processeurs utilisés. Le principe général est le suivant : allouer à chaque tâche n_i un processeur P_i, dont l'ordonnancement sera calculé de façon à faire terminer n_i au plus tôt sur ce processeur. En dupliquant progressivement sur P_i les parents des tâches n_t qui bloquent à cause des communications, on s'attend à obtenir un temps de fin raisonnable pour les différentes tâches du graphe. Avant d'aller plus dans le détail, introduisons quelques définitions supplémentaires :

Soit une copie d'une tâche n_j ordonnancée sur un processeur P_t. Le parent de n_j qui n'est pas ordonnancé avant n_j sur P_t et possède la plus grande date d'arrivée de données $dat(n_j, n_p)$ est appelé *parent critique* de n_j. Si la tâche n_j peut démarrer plus tôt lorsque la tâche en position $pos(n_j, P_t) - 1$ commence plus tôt, on dit que n_j est *computation-bounded*. Sinon, on dira que n_j est *communication-bounded*. Enfin, parmi toutes les tâches de P_t, la dernière tâche *communication-bounded* sera appelée le *bottleneck* de P_t.

L'algorithme `SDS` (pour *sub-DAGs scheduling*) fonctionne alors comme suit. On considère une tâche n_i à optimiser sur un certain processeur P_i. On va faire évoluer l'ordonnancement actuel en dupliquant sur P_t les parents critiques n_p des bottlenecks de P_t, jusqu'à ce que cela devienne impossible. On garde alors le meilleur des ordonnancements partiels P_t obtenus. La fonction de duplication cherchera encore à ranger n_p, le parent critique du bottleneck n_b, juste avant ce dernier, en initialisant $st(n_p, P_t)$ sur $dat(n_p, n_r)$, où n_r est le parent critique de n_p. De fait les tâches suivant n_b sur P_t se retrouvent décalées plus tard. Le makespan de l'ordonnancement partiel sur P_t peut donc être amené à augmenter, puis à diminuer au fur et à mesure que des tâches importantes sont dupliquées sur P_t. Il convient donc de répéter l'opération en question un maximum de $|\mathcal{N}| \times \kappa$ fois, où κ est une constante permettant de borner la complexité temporelle de l'algorithme.

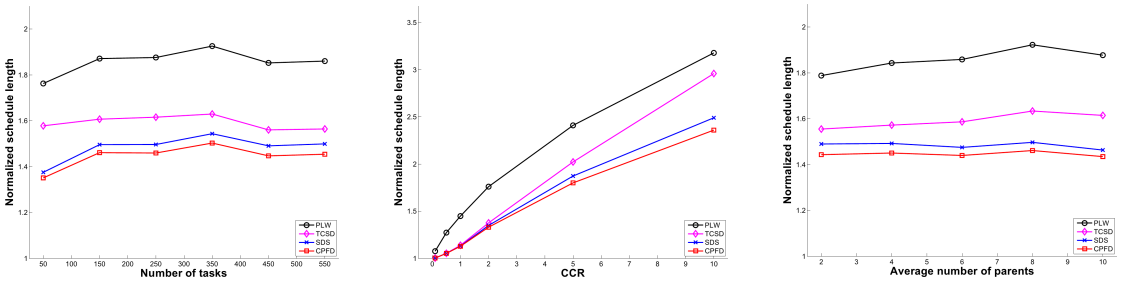
Là encore, l'algorithme n'est pas présenté dans le détail, et il faudra se référer à [BOC09] pour se convaincre que le temps d'exécution de `SDS` est bien $\mathcal{O}(|\mathcal{N}|^3)$.

3 Expériences et résultats

Dans leur papier [BOC09], Bozdağ et al. décrivent des expériences réalisées sur des graphes de tâches générés aléatoirement ainsi que sur des instances de la vie réelle. Ils

comparent leur algorithme `SDS` aux heuristiques de la littérature `PLW`, `TCSD` et `CPFD`, et discutent des effets de la compression `SC` sur ces mêmes méthodes classiques.

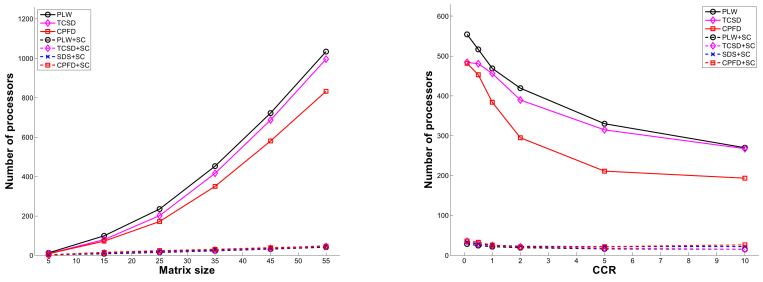
Au niveau des graphes aléatoires, les différents paramètres sur lesquels on joue sont : le nombre moyen de parents pour un nœud, le ratio *calcul sur communication* (noté *CCR*, rapport entre temps de calcul moyen d'une tâche et coût moyen de communication lié à une arête du graphe), et enfin le nombre de tâches (sommets) dans le graphe. Les instances issues de la vie réelle correspondent à des graphes de tâches pour la décomposition LU, l'élimination gaussienne, ou encore les équations de Laplace, où les paramètres à faire varier sont la taille des matrices considérées, et le CCR. En effet, les temps de calcul et de communication dépendent fortement de l'architecture du système sur lequel on veut paralléliser nos calculs. Il convient donc d'étudier l'influence de différents ratios calcul-communication sur ces graphes réels.



(a) (b) (c)

FIGURE 2: Longueur normalisée des ordonnancements sur des DAGs aléatoires.

La conclusion des expériences réalisées est que finalement, `SC` se débrouille plutôt bien, et réduit beaucoup le nombre de processeurs nécessaires à un ordonnancement donné (voir figure 3a quand la taille de la matrice augmente). De plus, sur des graphes aléatoires comme sur des graphes réels, l'algorithme `SDS` développé par les auteurs offre de bonnes performances (voir les figures 2a, 2b et 2c pour l'influence des différents paramètres sur des graphes aléatoires).



(a) (b)

FIGURE 3: Nombre de processeurs requis pour une élimination gaussienne.

En observant le graphe de la figure 2b, on constate que plus le CCR augmente, plus les communications prennent de l'importance et les différences entre les stratégies de duplication des algorithmes étudiés s'accentue. Autre effet du CCR : plus celui-ci est grand, plus l'ordonnancement obtenu aura un makespan σ élevé. Mais cela implique aussi plus de possibilité pour exécuter des tâches sur un nombre réduit de processeur, ce que nous confirme la figure 3b.

Conclusion

Finalement, les deux algorithmes proposés dans [BOC09] sont assez prometteurs. D'une part l'algorithme `SC` a une complexité $\mathcal{O}(|\mathcal{N}|^3)$ et permet d'optimiser n'importe quel ordonnancement valide de DAG en termes de nombre de processeurs. De plus l'algorithme `SDS` présenté a aussi une complexité $\mathcal{O}(|\mathcal{N}|^3)$, soit $\mathcal{O}(|\mathcal{N}|)$ fois moins que `CPFD` qui est présenté comme l'un des algorithmes les plus efficaces en ce qui concerne la longueur de l'ordonnancement obtenu, et pour des performances assez similaires. Notons aussi que les ordonnancements obtenus après `SC` utilisent pratiquement tous le même nombre de processeurs. Il serait donc intéressant de savoir à quel point ils se ressemblent, car `SC` doit modifier intensivement les différents ordonnancements pour arriver à des résultats aussi similaires.

Quoiqu'il en soit, l'approche proposée constitue donc une heuristique assez sophistiquée face à un problème pourtant NP-dur. Pour autant, il faut garder à l'esprit que même si la correction ou la complexité temporelle de ces algorithmes sont facile à déterminer, on ne dispose pas vraiment de borne théorique sur leur efficacité en tant qu'algorithme d'approximation. Les résultats obtenus sont donc avant tout expérimentaux, et il faut donc garder un esprit critique quand on s'attelle à un graphe de tâche issu d'un problème réel.

Enfin, on soulignera également les multiples hypothèses faites sur le modèle : homogénéité des processeurs en termes de calcul ou de communication, communications simultanées, etc. Or sur des systèmes réels, les unités de calculs sont souvent très hétérogènes, les communications créent des interférences, les coûts de communication ne sont pas les mêmes entre deux processeurs, etc. Il serait donc intéressant d'adapter ces méthodes théoriques sur des systèmes plus diversifiés correspondant à de véritables grilles de calcul.

Références

[BOC09] Doruk Bozdağ, Füsun Özgüner, and Umit V. Catalyurek. Compaction of Schedules and a Two-Stage Approach for Duplication-Based DAG Scheduling. *IEEE Transactions on Parallel Distributed Systems*, 20 :857–871, June 2009.