RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

HWS
Vol de travail hiérarchique

Jean-Noël Quintin, et Frédéric Wagner

Équipe projet INRIA MOAIS, Laboratoire CNRS LIG, Universités de Grenoble

---

Résumé
Nous présentons ici un nouvel algorithme d'ordonnancement distribué : *HWS*, destiné aux plate-formes fortement contraintes par le réseau telles les plate-formes de grilles ou les super-calculateurs actuels. Cet ordonnancement est une variante de l'ordonnancement classique par vol de travail et vise à diminuer le nombre de transferts réseau à longue distance réalisés. Contrairement aux travaux existants nous effectuons une analyse théorique de notre algorithme pour le cas restreint où la plate-forme est composée de grappes de puissance identique et où le programme à exécuter est de type "diviser pour régner". Nous montrons une diminution de l'utilisation des liens longue distance tout en conservant un bon équilibrage de charge.

Mots-clés : HWS, hiérarchie, ordonnancement, vol de travail, parallélisme

---

1. Introduction

Au cours des dernières années, l'augmentation de la puissance de calcul des machines parallèles a été réalisée essentiellement grâce à une augmentation massive du nombre de cœurs. La plate-forme *Roadrunner* par exemple, classée première au TOP500 se compose de plus de 100000 cœurs. Même la médiane du nombre de cœurs des plate-formes du TOP500 se situe déjà autour de 3000 cœurs.
Bien entendu, une telle augmentation a nécessité une complexification de l'infrastructure réseau utilisée. Que le réseau soit sous forme de *fat tree* ou d'*hypercube*, l'augmentation du nombre de machine se traduit par une augmentation du diamètre du réseau ainsi que du nombre de nœuds intermédiaires utilisés lors des communications.
Il est intéressant de noter que les outils logiciels utilisés pour la programmation parallèle n'ont pas évolué aussi rapidement que les plate-formes matérielles. A l'heure actuelle, *MPI* reste la bibliothèque la plus utilisée pour ce type d'architectures.
Il existe néanmoins d'autres approches du calcul parallèle où le travail est contenu dans un ensemble de tâches déployées à l'exécution sur l'ensemble des ressources de calcul. Souvent, cet ordonnancement est réalisé à l'aide d'un algorithme distribué de vol de travail comme dans les bibliothèques *CILK* [6], *Satin* [8] ou encore *Kaapi* [7]. Cet algorithme possède de bonnes propriétés dont celle d'être distribué, ce qui en fait un bon candidat pour une exécution sur un nombre élevé de nœuds de calcul.
En revanche, l'impact des communications et donc du réseau sur les performances d'un ordonnancement par vol de travail n'est pas très clair. Nous nous proposons donc ici d'étudier l'ordonnancement par vol de travail sur des plate-formes hiérarchiques et ce, de manière théorique.
Nous commençons, section 2 par présenter les travaux existants sur ce sujet, et notamment les travaux réalisés par Nieuwpoort [9] autour de la bibliothèque SATIN. Ces travaux mettent en évidence la possibilité d'améliorer les performances par d'autres algorithmes d'ordonnancement et présentent différentes heuristiques. Nous présentons section 3 un nouvel algorithme d'ordonnancement : *HWS*, dérivé directement de l'algorithme classique de vol de travail. Cet algorithme a pour vocation d'être extensible sur plusieurs niveaux de hiérarchie mais est ici présenté dans le cas où l'on ne distingue que 2 niveaux. Contrairement aux heuristiques existantes, nous proposons section 4 une analyse théorique de cet algorithme. Nous prouvons notamment que sous certaines hypothèses, l'équilibrage de charge est réalisé de manière quasi-optimale. Nous concluons alors section 5.

2. État de l'art et positionnement

La création de bibliothèques génériques facilitant la programmation est un point clef pour donner un accès facile aux méthodes de vol de travail. Ainsi, plusieurs bibliothèques sont apparues avec des objectifs différents. Les bibliothèques actuelles peuvent être regroupées en deux classes. Les bibliothèques restreintes aux plate-formes à mémoire partagée. Par exemple, la plus connue est CILK [6]. D'autres bibliothèques permettent des exécutions sur plusieurs milliers de machines comme KAAPI [7] et SATIN [8]. Dans la sous-section 2.1, nous rappelons le fonctionnement du vol de travail classique. Puis sous-section 2.2, nous présentons les algorithmes de vol de travail hiérarchiques existants.

2.1. Rappels vol de travail

Le principe d'ordonnancement par vol de travail est assez simple : Chaque processeur possède une pile de travail à réaliser. Lorsqu'un processeur n'a plus de travail prêt dans sa pile il en demande à un autre processeur. Un des points clef de l'algorithme de vol de travail est le choix de la victime lors d'une demande de travail. Initialement, ce choix est réalisé de façon aléatoire entre toutes les machines. Cette politique appelée *random* est largement utilisée dans les différentes bibliothèques. Les atouts principaux de cette politique de vol *random* sont les bornes théoriques du temps d'exécution et du nombre de vols, introduites par Blumofe et Leiserson [4]. Des heuristiques différentes ont été prouvées théoriquement PDF [5] et control-PDF [2] : Leur objectif est de limiter le nombre de défaut de cache sur plate-forme multi-cœurs. Dans la suite, nous nous intéressons au temps d'exécution du vol de travail sur plate-forme hiérarchique.

2.2. Vol de travail hiérarchique

Il existe peu d'études sur le vol de travail hiérarchique, nous présentons ici l'étude réalisée par l'équipe Ibis. Les expériences réalisées par l'équipe Ibis avec la bibliothèque Satin, montrent que sur certaines plate-formes, le temps d'exécution avec la politique *random* n'est pas optimal [1]. Pour améliorer les performances, de nouvelles stratégies ont été implantées. Ces nouvelles stratégies cherchent principalement à minimiser le temps passé à voler.

Les heuristiques proposées dans la bibliothèque SATIN [9] ont l'avantage d'être implantées en pratique. Les trois heuristiques proposées sont :

CHS : La plate-forme est représentée par un arbre. Lors d'un vol, le voleur lance une requête à ses fils. En cas d'échec de la requête au niveau d'un fils, le fils la transmet à ses propres fils. En cas d'échec sur tout le sous-arbre, la requête de vol est envoyé au père. Le but de cette heuristique est de favoriser le travail local et de limiter les vols distants.

CLS : La plate-forme d'exécution est découpée en grappes de machines. Un maître est associé à chaque grappe. Chaque machine de la grappe réalise des vols uniquement au sein de la grappe. Ces machines donnent des informations sur la quantité de travail restant localement au maître. Avec ces informations, le maître prend la décision de réaliser un vol vers une machine quelconque d'une autre grappe. Le but de cette stratégie est de diminuer l'utilisation réseau entre les grappes.

CRS : Les machines organisées en grappes peuvent réaliser deux types de vols : asynchrones et synchrones. Chaque machine peut réaliser un seul vol asynchrone à un instant donné. Les vols asynchrones sont restreint aux vols entre grappes. Au contraire, les vols synchrones sont restreint aux vols intra-grappes. Les vols sont réalisés uniquement dans le cas où il n'y a pas de travail localement. Cette heuristique favorise le travail local et le recouvrement travail, communications et les communications entre elles.

Ces heuristiques ont été utilisées sur de nombreux exemples tels que Fibonacci, Nqueens, ou encore le produit de matrices. Les baisses de performances de la politique *random* par rapport aux autres heuristiques sont principalement présentes sur des plate-formes avec une hiérarchie réseau.

Cette étude met en évidence que les heuristiques prenant en compte la hiérarchie peuvent en tirer partie pour fournir de meilleures performances. Ces heuristiques utilisent les informations de la plate-forme. Pourtant, il nous paraît important d'utiliser aussi les informations liées à l'application ordonnancée.

Dans la suite, nous proposons donc une heuristique différente pour prendre en compte les informations sur la plate-forme et sur l'application.

3. Vol de travail hiérarchique

Dans cette section, nous présentons notre heuristique prenant en compte la hiérarchie des plate-formes. Sur les plate-formes hiérarchiques, les communications longues distances sont plus coûteuses et de plus les liens de communications sont parfois partagés avec d'autres utilisateurs. Une utilisation abusive de ceux-ci entraîne une augmentation du temps d'exécution de notre application. Nous nous intéressons donc à réduire l'utilisation des liens réseaux les plus faibles. Lors de l'exécution d'une application par vol de travail, les transferts de données peuvent être réalisés à différents moments de l'exécution. Les deux principales manières sont :

préemptive : Lors d'un vol, toutes les données nécessaires a son exécution, sont transférées. Cette méthode permet d'avoir directement les données et de ne pas attendre leurs transferts. Par contre, les données peuvent être transférées inutilement.

paresseuse : les données sont transférées uniquement lors d'un accès direct à la donnée. Cette méthode permet de réaliser une découpe récursive en parallèle sans transférer les données. Les données sont transférées uniquement lors de l'exécution du travail proprement dit.

En pratique dans la bibliothèque KAAPI, le transfert est réalisé de manière préemptive. Chaque vol entraîne donc un transfert de données. En réduisant la quantité de vols distants, nous réduirons l'utilisation de ces liens et augmenterons potentiellement les performances.
Malheureusement, cette limitation rend complexe le problème d'équilibrage de la charge entre les groupes de machines reliées par des liens de communications plus faibles. Afin de mettre en évidence les liens de communications les plus faibles, nous définissons ici un groupe de machines comme un ensemble d'unités de calcul homogènes reliées par un réseau homogène. Classiquement, une grappe peut être considérée comme un groupe. Nous pouvons également considérer les coeurs d'un processeur comme un groupe. Ici, nous considérons un seul niveau de hiérarchie mais ce travail peut être généralisé à plusieurs niveaux.

Pour répondre au problème d'équilibrage de charge entre les groupes, nous proposons un algorithme de vol de travail basé sur l'idée suivante : les tâches de taille réduite ne peuvent être transférées d'un groupe à un autre. Les tâches de taille importante sont utilisées pour équilibrer la charge entre les groupes. Ainsi, nous exploiterons la localité des données et du travail. Nous considérons dans notre cas un ensemble de tâches décrites de manière récursive. Les tâches avec une profondeur faible représentent le travail contenu dans toutes ces descendantes. Les tâches de profondeur faible contiennent ainsi potentiellement plus de travail. Pour implanter notre contrainte, nous limitons les vols distants aux tâches au-dessus d'une limite. Nous définissons de par cette limite deux types de tâches :

Les tâches globales au dessus de la limite, qui peuvent être volées entre les groupes.

Les tâches locales en dessous de la limite, qui sont locales à un groupe.

Le nombre de tâches globales est restreint. Par exemple pour une application représentée par un arbre binaire comme Fibonacci, le nombre de tâches globales est égale à 2^4 pour une limite de 4. Lors d'un vol distant, la probabilité de trouver une tâche globale est alors très faible. Nous proposons donc de centraliser les tâches globales sur une machine du groupe (le maître du groupe). Dans notre heuristique, nous avons donc deux niveaux de hiérarchie :

Les maîtres : gèrent la distribution du travail entre eux et fournissent du travail à leur groupe dans le but d'exploiter toute la puissance disponible.

Les machines de chaque groupe : exécutent le travail fourni par leur maître. Ces machines ne réalisent pas de vols distants.

Au sein d'un groupe, le travail doit être équilibré entre les machines du groupe. Dans le but de répartir le travail fourni par le maître, nous utilisons l'algorithme de vol de travail classique en le restreignant au groupe.
Pour réaliser la distribution du travail, les maîtres utilisent deux piles de tâches. Une première pile appelée pile globale sert à conserver les tâches globales. Une seconde pile appelée pile locale est utilisée pour conserver les tâches locales.
Au début de l'exécution, un seul des maîtres a du travail. Il exécute les tâches en profondeur d'abord. À la création des tâches, le maître les insère dans une des piles en fonction de leur profondeur. Si la

profondeur est inférieure à la limite, il ajoute la tâche à la pile globale. Sinon il l'ajoute à la pile locale. Le maître fournit les tâches locales au groupe au fur et à mesure. Le maître a deux possibilités pour fournir à nouveau du travail à son groupe.

1. Le maître a encore du travail dans la pile globale : il exécute le travail pour en fournir une partie à son groupe.
2. Le maître n'as plus de travail dans la pile globale : il envoi une requête de vol auprès d'un autre maître.

Pour choisir l'instant où le maître fournit une tâche locale, nous pouvons utiliser une méthode pour détecter la quantité de travail restant dans le groupe. La détection de la quantité de travail au sein d'un groupe peut être réalisée de plusieurs manières. Satin en propose une dans son algorithme CLS [9]. Nous proposons d'utiliser la fréquence des vols au sein du groupe pour réaliser la détection. Lors de la fin d'exécution d'un travail, la fréquence des vols augmente. En analysant la fréquence des vols, nous pourrons prendre une décision sans perturber le système. Une dernière méthode utilisée dans la preuve est d'attendre la fin complète du travail local.

4. Validation

Nous réalisons dans un premier temps une analyse théorique de l'algorithme proposé. Section 4.1, nous prouvons théoriquement les performances de notre heuristique. La formule ainsi obtenue est alors analysée section 4.2.

4.1. Analyse de l'algorithme

Lors de la présentation de l'heuristique, nous avons distingué plusieurs gestions du groupe par le maître. Lorsque le maître donne du travail à son groupe, une unique tâche crée juste en dessous de la limite est fournie. Cette tâche crée à son exécution d'autres tâches de profondeur plus grande. Toutes les tâches créées par l'intermédiaire de la tâche fournie par le maître font parties du même bloc de tâches. Dans notre analyse théorique, nous nous limitons au cas où le maître attend la fin d'exécution du bloc fourni au groupe.

Cette gestion introduit le plus de temps d'inactivité. Les autres gestions auront donc de meilleurs performances. Avec la version de l'algorithme utilisée pour la preuve, les attentes du maître peuvent provoquer des inter-blocages lors de l'exécution. Pour éviter cette configuration, nous limitons notre démonstration à des applications sans dépendances entre les blocs. Cette restriction permet néanmoins d'exécuter des classes d'applications comme diviser pour régner.

La réalisation de la preuve se déroule en plusieurs parties. Nous commençons par modéliser l'application et la plate-forme d'exécution. Puis dans la preuve théorique, nous bornerons le temps d'exécution d'un bloc exécuté par un groupe. Cette borne, nous permettra de déduire le temps d'exécution de l'application globale.

4.1.1. Modélisation

Nous utilisons des résultats classiques montrés précédemment par Blumofe et Lierson [4]. Pour utiliser ces résultats, nous devons utiliser la représentation habituelle de l'application. Nous définissons ici un DAG G pour représenter l'application. Chaque noeud de G est une tâche unitaire. Les arêtes de G représentent une dépendance entre les tâches, par exemple création d'une tâche, ou précédence avec une tâche.

Cette modélisation permet de décrire une application parallèle dans son intégralité. Afin de décrire le modèle, nous utiliserons les notations classiques du domaine [3]. Nous rappelons ces notations :

W **:** Le nombre de noeuds dans G.

D **:** La longueur du chemin critique.

T_p **:** Temps d'exécution sur p processeurs.

Pour la stratégie *random*, la plate-forme réalisant l'exécution de G est composée de p machines. Le coût d'une communication sur la plate-forme est supposé borné. Sous ces contraintes, le temps d'exécution de G a été borné par :

$$O(\frac{W}{p} + D)$$

L'heuristique proposée nécessite d'autres notations. La limite imposée est définie par rapport à l'arbre de création des tâches. Ainsi, G est composé par les tâches globales et les tâches locales. Nous caractérisons plus précisément G :

W_u : Le nombre de tâches globales.

D_u : La longueur du chemin critique du travail réalisé par les maîtres.

W_d : Le nombre de tâche locales.

D_d : La valeur du chemin critique en-dessous de la limite.

L'ensemble des tâches locales est équivalent à l'ensemble des blocs. L'ensemble des blocs est caractérisé par :

W_i : Le nombre de noeuds présents dans le i^{eme} bloc.

D_i : Le nombre de tâches présentes sur le chemin critique du i^{eme} bloc.

B : Le nombre de blocs.

La figure 1 montre la représentation d'une application avec les différentes notations introduites.

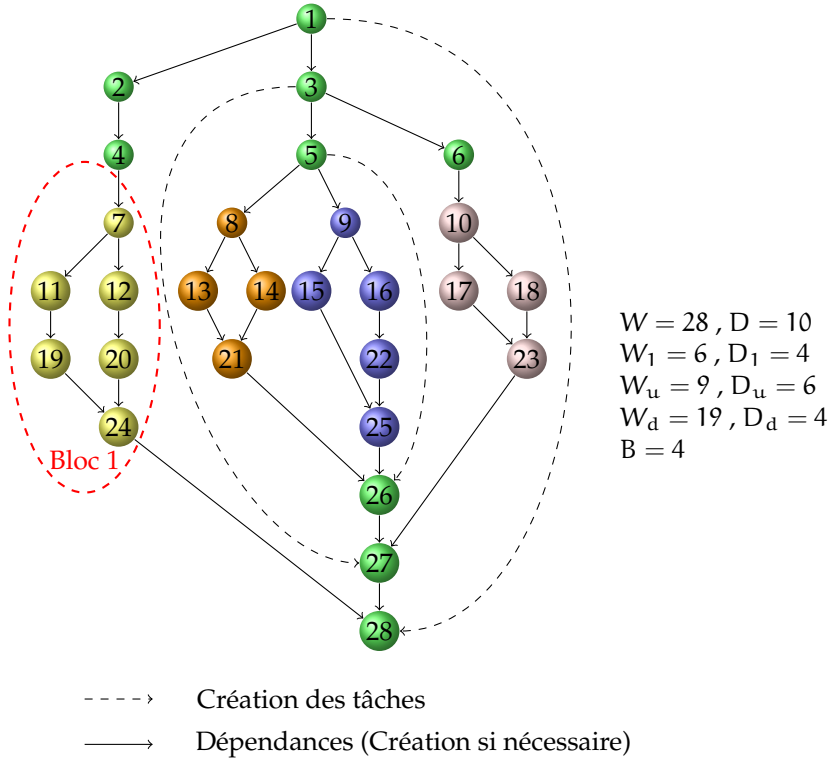


FIGURE 1 – Graphe représentant l'application

Par définition des blocs :

$$W_d = \sum_i W_i \tag{1}$$

$$D_d = \max_i D_i \tag{2}$$

L'application modélisée est exécutée par une plate-forme. La plate-forme d'exécution contient p processeurs (unités de calcul) divisés en g groupes de machines. Chaque groupe k de machines contient p_k machines. Dans chaque groupe, nous nous restreignons à avoir des communications à coût borné. Les communications entre les maîtres sont considérées bornées par une valeur plus importante que pour les groupes. Dans cette démonstration, nous considérons un nombre de machines identique par groupe. Cette limitation, nous permet de réaliser la démonstration. Nous espérons supprimer cette limitation dans de futurs travaux afin de prendre en compte des groupes hétérogènes de machines.

$$\forall k : p_k = cste = p_g \tag{3}$$

$$g = \frac{p}{p_g} \tag{4}$$

4.1.2. Preuve théorique

Théorème 1. *Le temps d'exécution de l'application avec cette heuristique est inférieur à* $O(\frac{W_u}{g} + \frac{W_d}{p} + D\frac{B}{g} + D + \max_i(\frac{W_i}{p_g}))$ *et le nombre de vol réalisé entre les groupes est* $O(g * (D + \max_i(\frac{W_i}{p_g})))$

Démonstration. Dans un premier temps, nous nous intéressons à l'exécution d'un bloc sur un groupe grâce à l'algorithme de vol de travail. Le bloc est exécuté seul sur un groupe. Son temps d'exécution peut être borné grâce à la formule classique. L'exécution d'un bloc peut être représentée pour le maître comme une chaîne de tâches. Dans la modélisation présenté dans l'article [4], une chaîne de tâches ne peut être volée dès que la première tâche de la chaîne est exécutée. Ainsi, nous pouvons modéliser l'attente du maître.

Nous construisons un DAG G' à partir de G. G' représentera l'application perçue par les maîtres. Toutes les tâches de G créées au dessus de la limite sont présentes à l'identique dans G'. G' contient donc toutes les tâches globales. Chaque bloc i de G est transformé en une chaîne de tâches de longueur $O(\frac{W_i}{P_g} + D_i)$ dans G'. La figure 2 montre un exemple de G'.

Le nouveau DAG G' virtuel représente l'application du point de vue des maîtres. Nous analysons l'exécution du nouveau DAG G' par les maîtres. Nous détaillons les caractéristiques de G' :

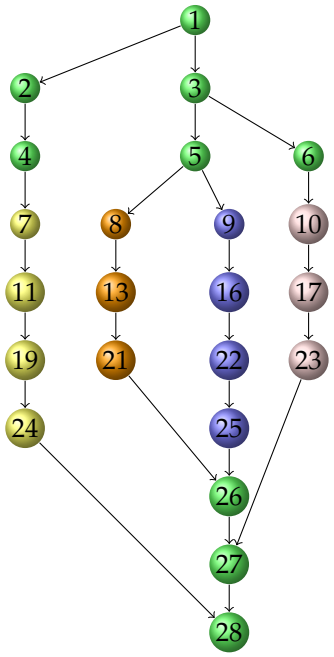
$$D' = D'_u + D'_d = D_u + \max_i O(D_d + \frac{W_i}{p_g})$$

$$=> D' \leq O(D + \max_i \frac{W_i}{p_g}) \tag{5}$$

$$W' = W'_u + W'd = W_u + \sum_i O(\frac{W_i}{p_g} + D_i) \leq W_u + O(\frac{W_d}{p_g}) + O(B * D_d)$$

$$=> W' \leq W_u + O(\frac{W_d}{p_g}) + O(B * D_d) \tag{6}$$

L'exécution de G avec l'heuristique théorique proposée sur la plate-forme hiérarchique est équivalente à l'exécution de G' avec un algorithme de vol de travail par les maîtres. Ainsi, le temps d'exécution de l'application est égale à $O(\frac{W_u}{g} + \frac{W_d}{p} + D\frac{B}{g} + D + \max_i \frac{W_i}{p_g})$. De la même façon, le nombre de vols entre les maîtres est égal à $g * (D + \max_i \frac{W_i}{p_g})$ □



caractéristiques de G'

$$W' = W'_u + W'_d$$
$$= W_u + \sum_i O(\frac{W_i}{p_d} + D_i)$$
$$D' = D_u + \max_i(D'_i)$$
$$= D_u + \max_i(O(\frac{W_i}{p_g} + D_i))$$
$$\leq O(D + \max_i(\frac{W_i}{p_g}))$$

FIGURE 2 – Graphe représentant G' pour une plate-forme où $p_g = 2$

4.2. Analyse des résultats

Nous avons prouvé que le temps d'exécution de l'algorithme théorique est borné par $O(\frac{W_u}{g} + \frac{W_d}{p} + D\frac{B}{g} + D + \max_i(\frac{W_i}{p_g}))$.

Cette formule est relativement complexe et l'on peut se demander si les différents termes qui la composent ont un sens au regard du problème réel ou si ils résultent du jeu de calcul mathématique.

Nous proposons donc d'examiner chaque terme un par un afin de les relier au problème réel.

$\frac{W_u}{g}$ **:** Le travail réservé au dessus de la limite doit être exécuté par les maîtres. Notre algorihtme implique ce terme de par sa conception, les tâches globales étant exécutées uniquement par les maîtres. Nous pouvons dire que ce terme représente un temps relativement faible sous réserve que la limite ne soit pas trop basse.

$\frac{W_d}{p}$ **:** Ce terme montre que le travail local est réalisé de manière optimale sur l'ensemble de la plate-forme.

D **:** Nous retrouvons le terme présent dans la formule classique.

$\max_i \frac{W_i}{p_g}$ **:** Ce terme est lié au point clef de notre algorithme : la limitation d'un bloc de tâches à une seule grappe de calcul. En effectuant cette modification il est clair que le temps de calcul ne peut pas être inférieur à la taille du plus grand bloc divisé par la puissance d'une grappe. Ce terme peut s'avérer important si un des blocs contient beaucoup plus de travail que les autres : seule une grappe sera alors active pour la plupart de l'exécution. Ce cas arrivera rarement en pratique pour des problèmes de type diviser pour régner, car la découpe récursive génerera des blocs relativement homogènes.

$D\frac{B}{g}$ **:** Au niveau de la démonstration, ce membre provient de la transformation des blocs en chaînes de tâches. Il représente la somme des chemins critiques de chaque bloc exécuté par un groupe. Ce terme est donc lié à la version théorique de notre algorithme où chaque grappe ne peut exécuter qu'un seul bloc à la fois. Dans l'algorithme utilisé en pratique, l'exécution des différents blocs se recouvre, et nous espérons donc que l'influence de ce terme se trouvera réduite.

La conclusion de cette analyse s'effectue sur plusieurs points. Tout d'abord, nous notons que la formule obtenue n'est pas un artefact mathématique mais correspond bien aux différentes contraintes posées

par le problème réel. Nous notons également qu'en pratique l'équilibrage de charge sera réalisé de manière quasi optimale. Enfin, la quantité de vols "longues distances" réalisés se trouve fortement réduite, passant de $O(p * D)$ pour l'algorithme classique à $O(g * (D + \max_i(\frac{W_i}{p_g})))$.

5. Conclusion

Nous avons présenté *HWS*, un nouvel algorithme d'ordonnancement distribué, basé sur le vol de travail. Cet algorithme a été conçu afin de pouvoir prendre en compte plusieurs niveaux de hiérarchie de la plate-forme mais est ici présenté et analysé dans le cas le plus simple où seuls deux niveaux sont considérés. Nous avons ainsi pour objectif de limiter l'utilisation des les liens réseaux les plus faibles en restreignant certains transferts de données localement. Bien entendu, il est clair que cet objectif ne doit pas se réaliser au prix d'un mauvais équilibrage de la charge.

Nous avons réalisé une analyse théorique des performances de notre algorithme de vol de travail hiérarchique et montré que sous certaines hypothèses, l'équilibrage de la charge est réalisé de manière aussi efficace que par l'algorithme classique de vol de travail. Cette preuve n'a néanmoins été possible qu'en ajoutant artificiellement un certain nombre de restrictions au problème initial : toutes les grappes ont la même puissance de calcul ; les applications considérées sont de type "diviser pour régner" ; chaque maître atteint l'inactivité de tous les nœuds du groupe qu'il dirige avant de passer au bloc suivant.

D'un point de vue théorique, un certains nombre de travaux sont encore à réaliser. Il nous parait prioritaire d'étendre nos résultats pour le cas de grappes hétérogènes tandis que les deux autres contraintes citées nous paraissent plus délicates à prendre en compte ainsi que de moindre impact.

D'un point de vue pratique, nous espérons être rapidement être en mesure d'effectuer des simulations. Celles-ci nous permettrons alors de guider des expériences réelles au dessus de la bibliothèque *Kaapi*, développée au sein de notre équipe.

Bibliographie

1. D. B. A, R. V. V. NIEUWPOORT, J. MAASSEN, G. WRZESINSKA, T. KIELMANN & H. E. BAL – « Adaptive load-balancing for divide-and-conquer grid applications », *Journal of Supercomputing* (2006).
2. G. E. BLELLOCH, R. A. CHOWDHURY, P. B. GIBBONS, V. RAMACHANDRAN, S. CHEN & M. KOZUCH – « Provably good multicore cache performance for divide-and-conquer algorithms », in *In Proc. 19th ACM-SIAM Sympos. Discrete Algorithms*, 2008, p. 501–510.
3. R. D. BLUMOFE – « Executing multithreaded programs efficiently », Thèse, Cambridge, MA, USA, 1995.
4. R. D. BLUMOFE & C. E. LEISERSON – « Scheduling multithreaded computations by work stealing », in *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science FOCS*, 1994, p. 356–368.
5. G. B. CARNEGIE, G. E. BLELLOCH & P. B. GIBBONS – « Effectively sharing a cache among threads », in *in SPAA '04 : Proceedings of the sixteenth annual ACM symposium on Parallelism in*, 2004, p. 235–244.
6. M. FRIGO, C. E. LEISERSON & K. H. RANDALL – « The implementation of the Cilk-5 multithreaded language », in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada), june 1998, Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998., p. 212–223.
7. T. GAUTIER, X. BESSERON & L. PIGEON – « Kaapi : A thread scheduling runtime system for data flow computations on cluster of multi-processors », in *PASCO '07 : Proceedings of the 2007 international workshop on Parallel symbolic computation* (New York, NY, USA), ACM, 2007, p. 15–23.
8. R. VAN NIEUWPOORT, J. MAASSEN, T. KIELMANN & H. E. BAL – « Satin : Simple and efficient java-based grid programming », *Scientific International Journal for Parallel and Distributed Computing* **6** (2005), no. 3, p. 19–32.
9. R. V. VAN NIEUWPOORT, T. KIELMANN & H. E. BAL – « Efficient load balancing for wide-area divide-and-conquer applications », in *PPoPP '01 : Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming* (New York, NY, USA), ACM, 2001, p. 34–43.