Equilibrage de charge dynamique sur plates-formes hiérarchiques

Jean-Noël Quintin

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Jean-Noël Quintin

Thèse dirigée par **Denis Trystram**
et codirigée par **Fédéric Wagner**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Équilibrage de charge dynamique sur plates-formes hiérarchiques

Thèse soutenue publiquement le **8 Décembre 2011**,
devant le jury composé de :

Mme Brigitte Plateau
Professeur Grenoble INP, Président
Mr Frédéric Suter
Chargé de recherche IN2P3 Lyon, Rapporteur
Mr Sid-Ahmed-Ali Touati
Professeur Université de Nice Sophia Antipolis, Rapporteur
Mr Jesper Larsson Träff
Professeur Université de Vienne, Examinateur
Mr Denis Trystram
Professeur Grenoble INP, Directeur de thèse
Mr Frédéric Wagner
Maître de Conférences Grenoble INP, Co-Directeur de thèse

Remerciements

Je voudrais remercier en premier l'ensemble des membres du jury pour avoir accepter de participer à ma soutenance, plus particulièrement mes rapporteurs Frédéric Suter et Sid Touati qui ont accepté la lourde tâche de relire dans le détail ce manuscrit. Mes remerciements s'adressent bien entendu aussi à la présidente du jury Brigitte Plateau et à Jesper Träff pour avoir participé activement à ma soutenance.

Je suis arrivé dans le monde de la recherche, il y a cinq ans lors d'un stage de deuxième année d'école d'ingénieur. Je ne connaissais pas le monde de la recherche. Je remercie Frédéric (Wagner) pour m'avoir permis de découvrir ce milieu et aussi pour m'avoir encadrer durant ces stages et la thèse. J'ai eu beaucoup de plaisir à travailler avec toi. De plus, je remercie Frédéric (Wagner) et Denis de m'avoir donner ma chance et de m'avoir encadrer durant cette thèse. Denis, merci d'avoir été présent dans les moments difficiles.

Ces travaux de thèse ont largement bénéficié de l'environnement de travail. Je tiens à remercier toutes les personnes présentes dans les deux équipes MOAIS et MESCAL qui ont contribé à leur façon à la qualité et au bon déroulement de cette thèse. Notamment, je souhaite remercier les personnes qui ont participé aux nombreuses discutions du matin comme Jean-Louis, Bruno, Marc, Florence. . ., et les acharnés du travail qui se réunissent le midi pour analyser finement les ensembles à 32 éléments : PF, Jean-Marc, Nicolas, Marie, Pierre (les deux N), Guillaume, Vincent, Fabrice, Mathias. . .. Ces personnes sont devenues plus que des collègues, ce sont maintenant de véritables amis. Merci aussi à Hélène et tous ceux qui ont participé à la relecture de ce document.

En particulier, j'ai eu à surmonter les différentes étapes de la thèse en même temps que Swann. Je te remercie pour tous les instants passés ensemble au laboratoire et en dehors. Comme le disent certains, tu es devenu un excellent «compère». Amuse toi bien au Japon.

Merci à mes parents, à ma sœur et à toute ma famille qui m'ont entouré et soutenu durant toutes ces années. Sans vous rien de ceci n'aurait été possible. Enfin, merci Aurélie de m'avoir supporté et de m'avoir épousé.

Sommaire

Introduction

Contexte

Avec la banalisation de l'accès à l'informatique, nous assistons depuis une trentaine d'années à un développement rapide des architectures matérielles, souvent symbolisé par la loi de Moore qui indique un accroissement exponentiel du nombre de transistors contenus dans un microprocesseur. Les développements récents dans ce domaine ont vu les constructeurs se tourner fortement vers les architectures parallèles comme une manière efficace de contourner les problèmes liés à une miniaturisation qui approche de plus en plus les limites physiques.

À terme, il est tout à fait envisageable de voir apparaître pour le grand public des architectures hétérogènes composées d'un ensemble de cœurs reliés par un réseau sur puce. D'une certaine manière, les plates-formes distribuées actuelles préfigurent les plates-formes embarquées de demain.

Cette évolution implique néanmoins de reconsidérer fortement les méthodes et techniques de développement logiciel. Tirer pleinement parti d'architectures parallèles nécessite un effort important de parallélisation des applications.

Or, à l'heure actuelle, cette parallélisation demande en général une compétence, ainsi que des ressources de développement importantes.

Les approches classiques de parallélisation reposent dans la plupart des cas sur l'utilisation de threads POSIX [13] dans le cas de plates-formes à mémoire partagée et de la bibliothèque *Message Passing Interface (MPI)* [56] dans le cas de plates-formes à mémoire distribuée. Il est tout à fait envisageable d'obtenir des performances importantes à l'aide de ces outils, mais les temps de développement requis sont généralement élevés. Le programmeur doit posséder une bonne connaissance de la plate-forme cible, être capable de réaliser la parallélisation de l'application tout en équilibrant lui-même la charge de travail sur l'ensemble des ressources de calcul.

On peut noter que la bibliothèque *MPI* vise déjà à simplifier une part du travail du programmeur en gérant les communications de manière automatique. Le programmeur écrit le type de communication qu'il désire, et la bibliothèque dispose de différents algorithmes qu'elle choisit lors de l'exécution, de manière transparente. Il s'agit là d'un premier changement dans la manière d'écrire les applications : au lieu de gérer directement les communications, le programmeur peut, par l'introduction d'opérations ayant une sémantique claire déléguer une partie de son travail à l'intergiciel. Il décrit ainsi «ce qu'il veut» plutôt que «comment y parvenir».

Plus récemment, cette tendance à l'automatisation s'est encore accrue, notamment par l'introduction d'intergiciels comme *Cilk* [31], *OpenMP* [19] ou encore *Kaapi* [36]. Ces bibliothèques permettent au programmeur d'automatiser l'équilibrage de charge

de son application par l'introduction d'une nouvelle sémantique. Quelle que soit la bibliothèque utilisée, le programmeur décrit non plus directement où exécuter différentes fonctions mais explicite les travaux indépendants et laisse à l'intergiciel la tâche de les ordonnancer sur les différentes ressources.

Cilk par exemple, propose une interface de programmation basée sur les mots clefs «Spawn» (création d'une tâche) et «Sync» (synchronisation des tâches créées). L'intergiciel se charge de répartir les tâches sur les processeurs à l'aide d'un algorithme d'ordonnancement de plus en plus répandu : le vol de travail [9]. Ce type d'algorithmes permet d'obtenir des performances élevées pour de nombreuses applications, et ce, sur de nombreuses architectures. Grâce à cette gestion automatique, le programmeur n'a plus besoin, dans une certaine mesure, de concevoir son application pour une plate-forme spécifique. L'algorithme d'ordonnancement nous permet ainsi d'augmenter de manière significative la portabilité de l'application.

Néanmoins, l'état actuel de la science pose des limites à la variété de plates-formes pour lesquelles les algorithmes d'ordonnancement existants permettent l'obtention de performances satisfaisantes. En particulier, il n'existe, à l'heure actuelle, pas réellement d'algorithme utilisable pour un ordonnancement en-ligne dans le cas où les communications sont susceptibles d'impacter le temps d'exécution.

Dans cette thèse, nous nous intéressons au développement d'algorithmes d'ordonnancement pour ce type de configurations. En particulier, nous espérons étendre le champ d'application du vol de travail en réduisant l'impact des communications. Notre démarche participe ainsi aux développements d'une nouvelle génération d'intergiciels, réduisant par l'automatisation les coûts et la complexité de la programmation parallèle.

Contexte scientifique et industriel de la thèse

Cette thèse s'inscrit dans le cadre du contrat CILOE du pôle de compétitivité MINALOGIC. Le pôle développe des partenariats industrie-recherche-formation, au niveau national et international, dans les secteurs de la santé, l'environnement, l'énergie, la mobilité et la connectivité, l'imagerie et de toutes les industries intégratrices d'électronique, à la recherche de ruptures ou d'innovations technologiques.
Le projet CILOE fédère des partenaires académiques (INRIA, TIMA et GIPSA-LAB), industriels (CS–SI, Bull, Edxact, ProBayes, Infiniscale) et le CEA-LETI pour mettre en place un environnement de calcul intensif : méthodologies, logiciels, ateliers de développement et infrastructures fortement sécurisées grâce à la mise de disposition de grappe et de grille de grappes de calcul. Pour gérer les évolutions des logiciels des partenaires industriels, l'équipe Moais a fourni l'outil de tests de non-régression ; cet outil a motivé la spécification et l'implémentation de l'outil DSMake, utilisé par InfiniScale. Le développement et l'étude d'algorithmes d'ordonnancement pour DSMake constitue la deuxième partie de cette thèse. De plus, la distribution automatique et efficace des tâches de calcul sur une grille, dont la structure est intrinsèquement hiérarchique, a motivé l'étude et le développement de stratégies de vol de travail adaptées qui constituent la troisième partie de cette thèse.

Contributions

Les contributions de cette thèse peuvent être divisées en deux catégories.

D'une part, nous proposons différents algorithmes d'ordonnancement. Ces algorithmes visent à réduire le temps de complétion de différentes applications sur de multiples plates-formes distribuées. Nous considérons notre problème d'ordonnancement sous un angle original : plutôt qu'une minimisation du temps de complétion en prenant en compte les temps de communication, nous cherchons ici à minimiser deux objectifs différents ; le temps de complétion et le volume de communication.

À partir de cette idée générale, nous proposons des algorithmes tirant parti de différentes informations. Dans un premier temps, l'utilisation de la connaissance du graphe de tâches nous permet le développement de Work-Stealing with COMmunication (WSCOM) ; une variante du vol de travail regroupant les tâches par affinité. Dans un second temps, Hierarchical Work-Stealing (HWS) et Probabilist Work-Stealing (PWS) nous permettent de prendre en compte la topologie réseau pour limiter l'utilisation des liens les moins performants. Pour tous nos algorithmes, nous présentons également une analyse théorique sur le nombre de transferts réalisés et les temps de complétion au pire cas.

D'autre part, l'ensemble de nos algorithmes est évalué de manière rigoureuse au sein d'expériences pratiques et de simulations. Ces algorithmes ont été implantés dans deux intergiciels différents. WSCOM est ainsi utilisé par *Outils de parallèlisation d'applications décrites par un Makefile sur plates-formes distribuées (DSMake)*, un outil de parallélisation d'exécution de Makefiles sur plate-forme distribuée. *DSMake* a été utilisé avec succès pour l'automatisation de tests de non régression. Enfin, HWS et PWS sont implantés dans la bibliothèque *Kaapi*, développée au sein de notre équipe («Moais»).

Les résultats expérimentaux obtenus confirment le bien fondé de l'approche retenue et montrent des gains de performances réels pour différentes applications.

Organisation du document

Ce document est organisé en trois grandes parties.

La première partie introduit formellement différents problèmes d'ordonnancement classiques. Nous présentons l'ensemble des travaux qui forment une base nécessaire pour la compréhension de nos contributions. Nous explicitons notamment le fonctionnement des algorithmes d'ordonnancement par liste et de vol de travail. En particulier, nous présentons certaines techniques de preuves qui s'avéreront utiles aux chapitres suivants. Nous décrivons également les algorithmes hors-ligne prenant en compte les communications lors de l'ordonnancement comme Dominant Sequence Clustering (DSC) ou CLANS. Enfin, nous mettons en relief les difficultés liées à un ordonnancement en-ligne dans le cas de communications non négligeables.

Dans la seconde partie, nous considérons une application particulière : l'ordonnancement en-ligne de l'exécution distribuée de Makefiles. Nous montrons qu'il est possible

de tirer parti de la connaissance du graphe de tâches pour regrouper les tâches par affinité. Nous proposons différents algorithmes d'ordonnancement basés sur le vol de travail. Ceux-ci reflètent différents choix d'implantation possibles ; différentes méthodes de regroupement considérées. Nous effectuons une analyse théorique bornant, pour tout algorithme déterministe, le nombre minimal de transferts à réaliser pour l'obtention de performances. Nous présentons également une analyse théorique de WSCOM dans le cas de graphes «Fork-Join».

Enfin, nous réalisons une validation expérimentale par simulation de WSCOM. Celles-ci nous permettent de montrer un gain de performances permettant d'étendre la variété des plates-formes d'exécution envisageables.

Finalement, nous présentons dans une dernière partie deux algorithmes : HWS et PWS. Nous utilisons alors la connaissance de la topologie réseau de la plate-forme pour limiter les communications les plus coûteuses. Nous effectuons une analyse théorique de ces algorithmes et montrons pour HWS une réduction importante des communications longue distance. Ceux-ci sont également validés en pratique par une série d'expériences réelles réalisées à l'aide de *Kaapi*.

I

Équilibrage de charge : état de l'art

Modélisation et complexité 1

Sommaire du chapitre

Ce travail de thèse s'intéresse aux problématiques d'ordonnancement de différentes applications sur des plates-formes où les temps de communication peuvent s'avérer non négligeables. Nous commençons dans un premier temps par présenter l'ensemble des outils et concepts dont nous ferons usage au cours de notre travail.

Une application est classiquement représentée par un ensemble de travaux à exécuter. Parmi ces travaux (tâches), certains sont indépendants les uns des autres. Ces tâches indépendantes peuvent être exécutées en même temps si plusieurs unités de calcul sont disponibles. De plus, leurs répartitions sur les unités de calcul influencent significativement les performances obtenues. Considérons par exemple, un ordonnancement de 8 tâches distinctes réalisé sur une grappe de 3 machines, tel que le montre la figure 1.1. Notre objectif est d'obtenir le résultat de l'ensemble des calculs le plus rapidement possible ; *i.e.,* de minimiser le temps de complétion. Chacune des tâches est associée à un temps d'exécution noté p_j pour la tâche j. Le placement d'une tâche consiste à l'assigner à un ordinateur et à définir sa date de début d'exécution.

Pour obtenir les meilleures performances, l'ordonnancement doit être optimisé en fonction de l'application et de la plate-forme. Afin de ne pas recalculer manuellement l'ordonnancement des tâches pour chaque plate-forme et chaque application, des algorithmes le calculant pour des familles d'applications et de plates-formes sont nécessaires.

Nous nous intéressons aux applications représentées par un ensemble de tâches avec précédences et nécessitant des communications, car celles-ci sont généralement un frein aux performances dans les applications parallèles.

Pour comprendre certains travaux existant sur l'ordonnancement de tâches, la section 1.1 introduit les notions de complexité, d'algorithmes d'approximation et d'ordonnancement en ligne qui serviront, tout au long de cette thèse, à mettre en évidence

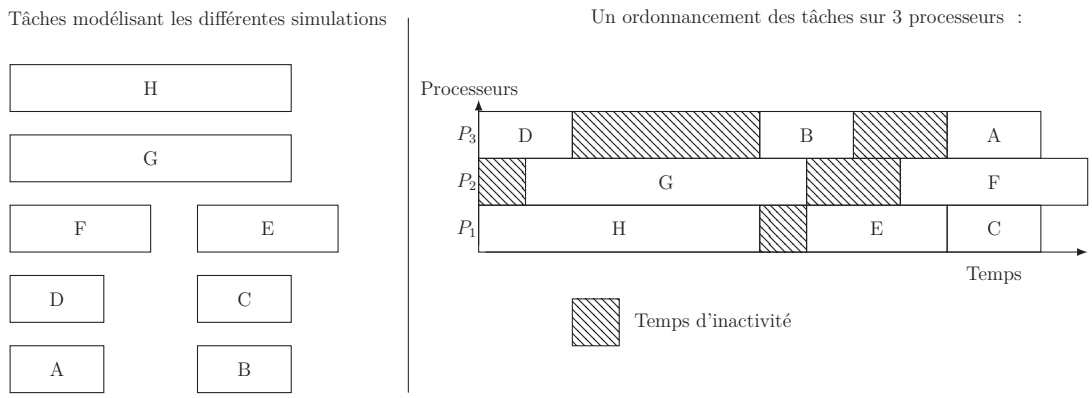


FIGURE 1.1 – Illustration d'un ordonnancement de tâches sur trois machines

les points clefs des problèmes abordés. Ensuite, la section 1.2 présente les algorithmes d'ordonnancement par liste, illustrés sur des tâches indépendantes. Enfin, la section 1.3 met en évidence l'impact des communications sur la complexité des problèmes d'ordonnancement de tâches.

1.1 Taxinomie et approximation

La résolution de problèmes d'ordonnancement est une tâche complexe fortement dépendante des particularités du problème considéré. On distingue néanmoins pour la plupart des travaux de la littérature une approche commune à l'ensemble des différents problèmes.

Dans une première étape, une étude de la complexité du problème permet de mettre en évidence si celui-ci est *NP-difficile*. Cette propriété implique qu'il n'est sans doute pas possible d'obtenir un algorithme calculant la solution exacte en temps polynomial.

Puisque la solution optimale n'est alors pas atteignable en un temps raisonnable, la résolution du problème passe par un algorithme fournissant un ordonnancement dont les performances approchent celles de la solution optimale. Lorsque la performance des ordonnancements calculés est garantie, l'algorithme est alors un *algorithme d'approximation.*

Nous détaillons dans un premier temps les notions de complexité relative aux problèmes NP-difficiles dans la section 1.1.1. Puis, la section 1.1.2 introduit formellement les algorithmes d'approximation pour l'ordonnancement en-ligne et hors-ligne.

1.1.1 Classification et Complexité

L'étude de la complexité est une branche de l'informatique qui étudie la difficulté intrinsèque à chaque problème. Elle a pour objectif de permettre une classification des problèmes ainsi que l'étude des limites théoriques de différents modèles d'exécution.

Les classes de complexité sont généralement définies à partir des problèmes de décision.

Définition. *Un problème de décision*

Un problème de décision est défini par des paramètres et une question dont la réponse est «OUI» ou «NON». Les paramètres sont appelées l'instance du problème.

Généralement une classe de complexité est définie comme étant l'ensemble des problèmes de décision pour lesquels il existe au moins un algorithme de résolution vérifiant les contraintes imposées par cette classe. Une des contraintes les plus couramment utilisées porte sur le temps de calcul nécessaire pour résoudre le problème.

Parmi ces classes, nous nous intéressons principalement à deux d'entre-elles : P et NP.

Classe P : est l'ensemble des problèmes de décision pour lesquels il existe une machine de Turing déterministe trouvant la solution en temps polynomial.

Classe NP : est l'ensemble des problèmes de décision pour lesquels il existe une machine de Turing non déterministe trouvant toute réponse «oui» en temps polynomial.

Une des questions fondamentales en informatique vise à déterminer si $P = NP$. Cette question est cruciale car la classe P définit l'ensemble des problèmes pour lesquels une solution exacte est calculable en temps polynomial. Si $P \neq NP$ alors, certains problèmes de NP ne sont pas dans P et donc ne sont pas solvables rapidement sur les machines actuelles [1]. À l'inverse $P = NP$ impliquerait que tout problème de NP peut être résolu en temps polynomial.

La conjecture actuelle étant que $P \neq NP$, il est donc intéressant d'identifier les problèmes de NP qui ne semblent pas appartenir à P. Pour ce faire, nous présentons ici la notion de problème *NP-complet*, ainsi que la notion de réduction *polynomiale*.

Définition. *Réduction polynomiale*

Soit Π *et* Π' *deux problèmes,* $\Pi' \propto \Pi$ *(il existe une réduction polynomiale de* Π' *à* Π*), s'il existe un algorithme polynomial qui transforme chacune des instances de* Π' *en une instance de* Π *avec une réponse identique.*

Soit Π et Π' appartenant à la classe de complexité NP et $\Pi' \propto \Pi$; si il existe un algorithme solvant Π en un temps polynomial alors Π' est lui aussi solvable en un temps polynomial. Ainsi, moins formellement, Π est au moins aussi *difficile* que Π'.

Définition. *NP-difficile*

Un problème Π *est NP-difficile si* $\forall\ \Pi' \in NP,\ \Pi' \propto \Pi$.

Définition. *NP-complet*

Un problème Π *est NP-complet si*
- Π *est NP-difficile,*
- $\Pi \in NP$.

1. Tant qu'il n'existe pas de machines quantiques

La résolution d'un problème Π, NP-difficile en temps polynomial implique alors, par définition, que tous les problèmes de NP peuvent également être résolus en temps polynomial. Puisque la conjecture actuelle est que $NP \neq P$, les problèmes NP-difficiles sont considérés comme étant non-solvables en temps raisonnable (polynomial).

Après la mise en évidence de l'existence d'un problème NP-complet par Cook [21], la démonstration de la NP-complétude d'un problème s'est simplifiée. Montrer qu'un problème est NP-difficile nécessite uniquement de réduire un des problèmes NP-difficile à ce problème. Une telle preuve se base sur la transitivité de la relation «se réduit à».

Nous montrons par un exemple simple de réduction, qu'il est souvent plus facile de la réaliser à partir d'un problème proche. Pour l'illustrer, nous utilisons l'exemple $P_2||C_{\max}$ dans lequel l'objectif est de minimiser le temps d'exécution ($C_{\max}$) d'un ensemble de tâches sur deux machines. Ce problème étant d'optimisation, il est nécessaire, pour étudier sa complexité, de considérer le problème de décision associé que nous appellerons ORDO.

Définition. *ORDO.*
- *Instance :*
  - *un ensemble de n tâches $(t_1, ..., t_n)$, chacune d'entre elles ayant un temps d'exécution p_i*
  - *un entier K.*
- *Question : Existe-t-il une découpe de l'ensemble des tâches en deux parties distinctes telle que la somme des temps d'exécution de chacune des deux parties soit inférieure à K ?*

Pour montrer que le problème ORDO est NP-complet, il est possible de réaliser une réduction simple à partir de PARTITION [33] qui est NP-complet.

Définition. *PARTITION.*
- *Instance : un ensemble de n entiers : $a_1, ..., a_n$.*
- *Question : Existe-t-il une partition de $1, 2, .., n$ telle que :*

$$A \cup B = 1, 2, .., n,\ A \cap B = \emptyset \text{ et } \sum_{i \in A} a_i = \sum_{i \in B} a_i$$

propriété. *Le problème ORDO est NP-complet [33].*

Démonstration. Pour réduire PARTITION à ORDO, la fonction de réduction transforme chaque entier a_i en une tâche de temps d'exécution $p_i = a_i$ et fixe $K = \sum \frac{a_i}{2}$. Cette transformation est réalisable en temps linéaire.

Pour une instance de PARTITION ayant une réponse positive, l'ensemble des entiers peut être séparé en deux ensembles distincts A et B. Nous avons : $\sum_{i \in A} a_i = \sum_{i \in B} a_i$. Ainsi, $\sum_{i \in A} a_i = \sum_{i \in 1,2,..,n} \frac{a_i}{2} = K$. Notons A' l'ensemble des tâches associées aux entiers appartenant à A. Nous avons : $\sum_{i \in A'} p_i = K$. Ainsi, l'ensemble des tâches est séparé en deux parties dont la somme des temps d'exécution est inférieur ou égale à K. Chaque instance de PARTITION ayant une réponse positive est transformée en une instance de ORDO ayant une réponse positive.

Réciproquement pour une instance de ORDO ayant une réponse positive, nous obtenons de manière similaire deux ensembles de tâches. Sur chaque ensemble, la somme des temps d'exécution est égale à K. De plus, chaque ensemble est équivalent à un ensemble d'entier. Ainsi, sur chaque ensemble d'entiers associé à un des deux ensembles de tâches, la somme des entiers est égale à K.

Enfin, il est possible de montrer de manière simple que ORDO appartient à NP, il est donc NP-complet. □

Les deux problèmes utilisés lors de la réduction sont tous deux NP-complet au *sens faible* ; c'est-à-dire qu'il existe un algorithme optimal pseudo-polynomial. Au contraire, la difficulté des problèmes NP-complet au *sens fort* implique qu'il n'existe pas d'algorithme pseudo-polynomial pour les résoudre. Par exemple, le problème plus général d'ordonnancement de tâches sur p machines ($P||C_{\max}$) dont une instance est présentée figure 1.1, est NP-difficile au sens fort. Pour plus de détail sur la complexité au sens fort ou faible [33].

Comme nous venons de le montrer certaines fonctions de réduction sont vraiment simples et intuitives lorsque les problèmes sont proches. Ainsi, l'étude de complexité met souvent en évidence les liens entre des problèmes proches. Les analyses de complexité réalisées sur les problèmes d'ordonnancement dessinent ainsi les liens entre différentes familles de problèmes.

Nous présentons ici une liste de problèmes classiques, en lien avec le problème d'ordonnancement avec communications. Ceux-ci mettent en évidence l'origine des difficultés de résolution (ce qui rend un problème donné difficile) et seront également réutilisés section 2 lors de la présentation de différents algorithmes d'approximation.

Les problèmes sont présentés par ordre croissant de complexité :

$P|p_i = 1|C_{\max}$ **:** Ordonnancement de tâches unitaires sans précédences. Ce problème est solvable optimalement en temps polynomial, il appartient à P.

$P_2||C_{\max}$ **:** Ordonnancement de tâches indépendantes sur deux machines. Malgré la simplicité apparente du problème, il est NP-difficile au sens faible [33].

$P||C_{\max}$ **:** Problème identique à $P_2||C_{\max}$ mais avec un nombre de machines arbitraire. Ce dernier est NP-difficile au sens fort [32].

$P|prec|C_{\max}$ **:** Ordonnancement de tâches avec précédences sans communications sur une plate-forme homogène. L'ajout de précédences entre les tâches complexifie le problème. Cela est mis en évidence avec le passage de $P|p_i = 1|C_{\max}$ qui est dans la classe de complexité P, à $P|prec, p_i = 1|C_{\max}$ qui est NP-difficile au sens fort.

$Q|prec|C_{\max}$ **:** Ordonnancement de tâches avec précédences sur une plate-forme de machines hétérogènes dont les puissances sont liées par un facteur constant. L'objectif est de minimiser $C_{\max}$ qui est le temps de fin d'exécution de la dernière tâche. Ce problème est une généralisation de $P|prec|C_{\max}$, il hérite donc de sa complexité.

$Q|prec, c_{ij}|C_{\max}$ **:** Ordonnancement de tâches avec précédences et communications sur une plate-forme de machines hétérogènes. Les communications ajoutent elles aussi leur part de complexité : chaque communication peut perturber l'exécution

du travail et les autres communications. Le problème $Q|prec|C_{\max}$ est un cas particulier de ce problème, il est donc au moins aussi difficile.

Le problème central étudié dans cette thèse est $P|prec, c_{ij}|C_{\max}$. Dans ce problème, l'objectif est d'ordonnancer un ensemble de tâches avec des communications sur un plate-forme homogène en minimisant le temps d'exécution global. Ce problème est une généralisation de $P|prec|C_{\max}$, il est donc NP-difficile au sens fort.

1.1.2 Algorithmes d'approximation : hors-ligne et en-ligne

Les contraintes sur le travail à exécuter et la plate-forme d'exécution influencent directement la modélisation du problème considéré. Par exemple, les tâches peuvent être modélisées avec un temps d'exécution bien défini, mais également avec un temps d'exécution inconnu. Dans le premier cas, l'ordonnancement peut être calculé en utilisant les temps d'exécution et il est donc possible de l'obtenir avant l'exécution. À l'inverse, dans le second cas, l'ordonnancement est calculé au fur et à mesure de l'exécution des différentes tâches. On distingue ainsi deux cas : hors-ligne et en-ligne.

Considérons dans un premier temps un exemple de problème hors-ligne. Nous illustrons à la figure 1.1 une instance du problème $P||C_{\max}$, pour un nombre de machines fixé à 3 et un nombre de tâches égal à 8. Le tableau ci-dessous définit les temps d'exécution de chacune des tâches.

Nom de la tâche	A	B	C	D	E	F	G	H
Temps d'exécution	2	2	2	2	3	3	6	6

Pour ordonnancer les tâches de cette instance, nous avons choisi l'algorithme Largest Processing Time (LPT) (Largest Processing Time) classiquement utilisé dans l'ordonnancement de tâches. Cet ordonnancement introduit par Graham [40] est calculé de la façon suivante :

Les tâches sont triées par rapport à leur temps d'exécution, ici pour LPT de la plus grande à la plus petite. Dans cet ordre, les tâches sont ordonnancées pour être exécutées le plus tôt possible. Ainsi, les tâches les plus grandes sont exécutées en premier. L'ordonnancement avec LPT est illustré figure 1.2

L'ordonnancement optimal pour cette instance est fourni sur cette même figure. Ainsi, sur cet exemple, il est clair que l'algorithme LPT ne donne pas l'ordonnancement optimal mais est néanmoins très performant [38].

On peut se demander si LPT montre toujours d'aussi bonnes performances, et ce, pour chaque instance possible en entrée.

Dans les problèmes abordés en ordonnancement, l'objectif est généralement de minimiser ou de maximiser la valeur d'une fonction. Dans notre exemple, le but est de minimiser le temps d'exécution de l'ensemble des tâches. Afin d'évaluer la qualité d'une solution, sa valeur est comparée à la valeur de la solution optimale.

Pour garantir l'efficacité d'un algorithme, nous utilisons la notion de ρ-approximation.

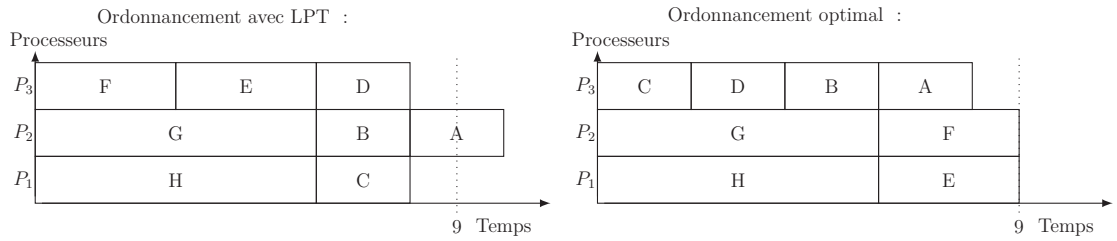


FIGURE 1.2 – Ordonnancement calculé par l'algorithme LPT

Définition. *ρ-approximation*

Un algorithme est un algorithme de ρ-approximation :
- *si $\rho > 1$: pour toute instance du problème, la valeur C de la solution calculée est inférieure à la valeur optimale OPT multipliée par ρ ($C \leq \rho \times OPT$).*
- *si $\rho < 1$: pour toute instance du problème, la valeur C de la solution calculée est supérieure à la valeur optimale OPT multipliée par ρ ($C \geq \rho \times OPT$).*

Pour les problèmes d'ordonnancement abordés dans ce manuscrit, l'objectif est de minimiser le temps d'exécution. Dans ce cas, la valeur ρ est toujours supérieure à 1 sinon l'ordonnancement ne serait pas valide.

Pour l'algorithme LPT ordonnançant les tâches sur m machines, Graham [40] a montré que le ratio d'approximation est inférieur à $\frac{4}{3} - \frac{1}{3\times m}$. Nous présenterons plus en détail une garantie de performance introduite par Graham [40] dans la section 1.2.1, montrant que les ordonnancements par liste dont LPT fait partie sont des 2-approximations.

Considérons maintenant le cas d'un ordonnancement en-ligne. Nous supposons ici que le temps d'exécution des tâches n'est pas connu à priori.

L'algorithme d'ordonnancement en-ligne classique est simple, c'est un algorithme glouton. Dès qu'un processeur n'a plus de travail à effectuer, l'algorithme ordonnance une tâche sur ce processeur. L'ordonnancement en-ligne est ainsi réalisé avec moins d'informations que l'ordonnancement hors-ligne.

Une des questions intéressantes est de savoir à quel point ce manque d'information impacte les performances.

Pour répondre à celle-ci, la performance des algorithmes en-ligne est comparée à la valeur de la solution optimale (pour le problème hors-ligne) dans une analyse de compétitivité [11]. De manière similaire à la définition des algorithmes d'approximation, un algorithme est dit ρ-compétitif si son ratio de performance est borné par une valeur ρ pour toute entrée.

Généralement, l'obtention de bornes inférieures sur un ratio de compétitivité est réalisée à l'aide de techniques par adversaires [4]. En simulant l'algorithme, un adversaire avec la connaissance globale du système est utilisé pour contraindre chaque choix réalisé et mettre ainsi en évidence l'instance conduisant aux ratios de performance les plus grands.

Dans notre exemple d'ordonnancement de tâches indépendantes en-ligne, l'algorithme glouton (déterministe) place les tâches sans connaître leurs temps d'exécution. L'adversaire a donc le droit de choisir le temps de la tâche ordonnancée. Pour calculer une borne inférieure du ratio de compétitivité, il suffit de considérer que les tâches sont choisies dans l'ordre le plus défavorable : de la plus petite à la plus grande. La figure 1.3 met en évidence le pire cas de l'exemple présenté précédemment avec 3 machines et 8 tâches.

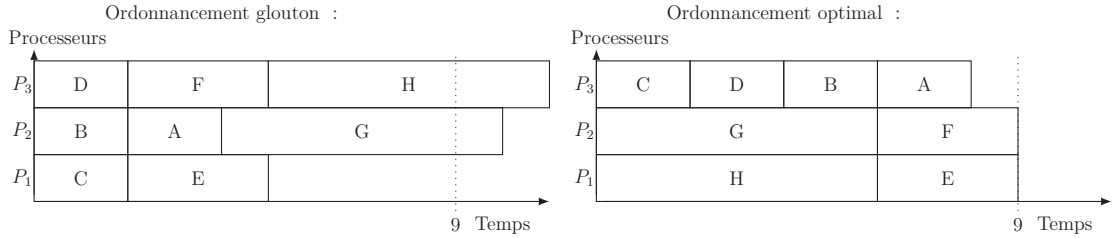


FIGURE 1.3 – Ordonnancement glouton en-ligne face à un adversaire

L'algorithme glouton proposé précédemment pour l'ordonnancement de tâches indépendantes en-ligne est 2-compétitif. L'analyse théorique de ce ratio est identique à la preuve introduite par Graham [40] pour les ordonnancements par liste, présentée section 1.2.1.

1.2 Ordonnancement sans précédences

La famille des algorithmes de liste, qui est la base de la majorité des algorithmes abordés dans cette thèse, a été développée initialement pour l'ordonnancement de tâches sans précédences. Puis, ces algorithmes ont été simplement adaptés aux différents modèles de plus en plus complexes : précédences entre les tâches, communications et processeurs hétérogènes. Pour présenter les algorithmes de liste, la section 1.2.1 détaille le principe avec des tâches indépendantes et des processeurs homogènes. Puis, la section 1.2.2 présente la modélisation des plates-formes hétérogènes et l'adaptation des algorithmes de liste à celles-ci.

1.2.1 Tâches indépendantes, plate-forme homogène($P||C_{\max}$)

Comme présenté en exemple dans la section 1.1.1 relative à l'étude de la complexité, le problème $P||C_{\max}$ est NP-difficile au sens fort [32]. Il n'existe donc pas d'algorithme pseudo polynomial pour le résoudre optimalement.

Ordonnancement par liste

L'algorithme d'approximation LPT, présenté précédemment 1.1.2, fait partie de la famille des ordonnancements par liste. Leur principe est simple : les tâches sont triées

dans une liste, puis ordonnancées dès que possible dans l'ordre de la liste. Cette famille d'algorithmes est utilisable pour résoudre le problème en-ligne et hors-ligne associé à $P||C_{\max}$.

L'ordonnancement par liste est un algorithme important que nous réutilisons tout au long de cette thèse. Nous proposons donc d'examiner en détail les différentes propriétés qui en découlent.

Les tâches sont exécutées sans délai d'attente entre elles, ce qui évite ainsi que des processeurs soient inactifs alors qu'il reste des tâches à exécuter. Notons que cette propriété est dominante : il existe toujours une solution optimale sans délai d'attente. Une autre propriété intéressante des ordonnancements par liste provient de l'existence pour chaque instance d'un ordre des tâches dans la liste qui fournit la solution optimale.

Les ordonnancements par liste sont facilement adaptables à la résolution du problème en-ligne. Deux différences interviennent : sur le moment de la prise de décision et l'ordre des tâches qui ne peut pas être calculé en fonction de leurs temps d'exécution. L'algorithme est le suivant : dès qu'un processeur devient inactif, il prend une tâche de la liste pour l'exécuter. Cette implantation est simple et centralisée car chaque processeur accède à la même liste.

Une telle centralisation introduit des problèmes de concurrence entre les processeurs. Pour éviter ces problèmes de concurrence et ainsi améliorer le passage à l'échelle, le vol de travail qui est largement détaillé section 3.2, reprend l'idée avec une implantation distribuée de la liste de tâches.

Garantie de performances

Une analyse théorique des performances des ordonnancements par liste permet de montrer que quelque soit l'ordre des tâches dans la liste, le ratio de performance est inférieur à $2 - \frac{1}{m}$ avec m le nombre de processeurs. Cette borne a été prouvée initialement par GRAHAM [40]. Nous détaillons ici une preuve basée sur des arguments géométriques.

Théorème 1.1. *Tout algorithme d'ordonnancement par liste fournit un ordonnancement dont le temps d'exécution global est inférieur à* $(2 - \frac{1}{m}) \times OPT$ *(OPT est le temps d'exécution optimal).*

Démonstration. Nous considérons un ordonnancement par liste d'un ensemble de tâches sur m machines. Considérons t_f la tâche exécutée en dernier, p_f son temps d'exécution et s_f sa date de début d'exécution, comme illustré figure 1.4. Puisque la tâche t_f n'a pas démarré avant s_f, tous les processeurs ont été actifs du début de l'exécution jusqu'à la date s_f.

La quantité de travail exécutée à la date s_f sur les m machines est égale à $s_f \times m$. Puisque tous les processeurs ont travaillé durant ce temps, la quantité $s_f \times m$ est inférieure à la quantité de travail global de l'application moins la quantité de travail de la dernière tâche qui n'est pas encore exécutée à la date s_f. Nous obtenons que : $s_f \times m \leq W - p_f$ avec W la quantité de travail de l'application.

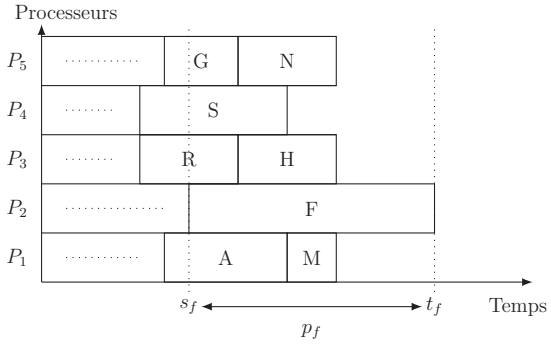


FIGURE 1.4 – Illustration de l'exécution de la dernière tâche

De plus, le temps d'exécution global $C_{\max}$ est égal à $s_f + p_f$. En bornant s_f par $\frac{W-p_f}{m}$, nous obtenons :

$$C_{\max} = s_f + p_f \leq \frac{W - p_f}{m} + p_f = \frac{W}{m} + (1 - \frac{1}{m}) \times p_f$$

Pour finir, le temps d'exécution optimal OPT est supérieur à p_f car la tâche t_f doit être exécutée par un des processeurs, et à $\frac{W}{m}$. Ainsi, on obtient :

$$C_{\max} \leq \text{OPT} + (1 - \frac{1}{m}) \times \text{OPT} = (2 - \frac{1}{m}) \times \text{OPT}$$

□

L'exemple de la figure 1.5 nous montre que la borne est atteinte.

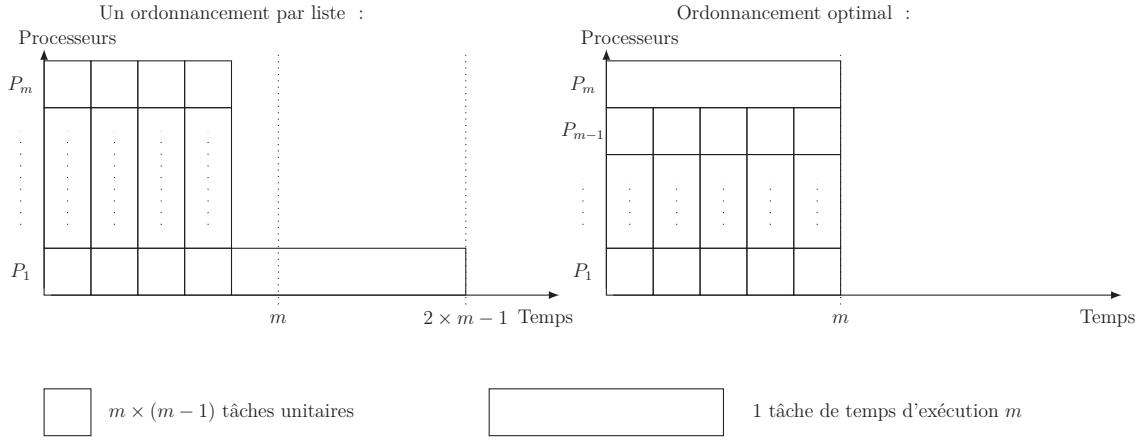


FIGURE 1.5 – Exemple atteignant la borne sur m machines

1.2.2 Tâches indépendantes, plate-forme hétérogène $(Q, R||C_{\max})$

Dans tous les exemples présentés jusqu'ici, les plates-formes étaient considérées homogènes. En pratique, il est assez fréquent sur une grappe de machines qu'il y ait

plusieurs types de machines différentes. Pour obtenir une modélisation proche des contraintes réelles, le modèle de machine doit ainsi être modifié.

Bien que les travaux développés dans cette thèse ne ciblent pas directement ce type de machines, nous détaillons ici les techniques d'ordonnancement de tâches sur plates-formes hétérogènes afin d'expliciter les conséquences de l'hétérogénéité sur les algorithmes d'ordonnancement par liste qui forment la base de nos travaux.

Modélisation d'une plate-forme hétérogène

Sur une plate-forme hétérogène, le temps d'exécution d'une tâche dépend de la machine qui exécute cette dernière et potentiellement de l'instant de l'exécution. L'hétérogénéité entre deux machines peut provenir des caractéristiques des machines (processeurs de puissance différente, cartes mères différentes, ...), mais aussi des applications qui sont exécutées par les machines. En fonction de la provenance de l'hétérogénéité, la modélisation est différente ; nous présentons ici deux modèles différents pour l'ordonnancement sur plates-formes hétérogènes.

Pour modéliser ces plates-formes, chaque tâche i a un temps d'exécution p_{ij} qui dépend du processeur. En fonction de la provenance de l'hétérogénéité, la modélisation du temps d'exécution change.

Les plates-formes dont l'hétérogénéité provient des caractéristiques des machines, sont désignées par la lettre Q dans la notation à 3 champs introduite par GRAHAM [40]. Cette notation permet de caractériser un problème d'ordonnancement avec une formule restreinte. Pour ces machines hétérogènes, la différence de puissance entre elles est constante au cours de l'exécution. Ainsi quelque soit l'instant de l'exécution, le temps d'exécution ne varie pas. De plus, la différence de puissance entre les machines ne dépend pas de la tâche. Plus formellement, pour chaque couple de processeurs $(j, k) \in [0, m] \times [0, m]$ et pour toutes tâches i, le facteur $\frac{p_{ij}}{p_{ik}}$ est constant quelque soit la tâche et l'instant.

Au contraire, lorsque l'hétérogénéité provient non seulement des caractéristiques des machines mais aussi des applications et du type de processeurs (GPU, CPU), les problèmes liés à ces plates-formes sont désignés par la lettre R. Un exemple de ce type de plates-formes est la plate-forme de SetiHome [88] dont les machines sont utilisées par plusieurs utilisateurs en même temps et sont accessibles uniquement quand le propriétaire de la machine n'en a plus le besoin. Les puissances de calcul ne sont plus liées entre elles par un facteur constant au cours du temps. De plus, certains processeurs sont plus adaptés à l'exécution de certaines tâches que d'autres.

Ces plates-formes sont modélisées par une matrice fournissant l'évolution de la puissance de calcul de chacune des machines au cours du temps.

Ordonnancement sur plates-formes hétérogènes

Dans nos problèmes d'ordonnancement, l'objectif est d'obtenir l'exécution de l'ensemble des tâches le plus rapidement. Sur les plates-formes hétérogènes, la comparaison des dates de début d'une tâche ne donne pas directement d'information sur la date

de fin de la tâche. Comme l'illustre la figure 1.6, la tâche C commence plus tôt sur le processeur 1 mais termine plus tard que si elle avait été ordonnancée sur le processeur 2. Contrairement au problème sur plates-formes homogènes, la propriété de dominance des ordonnancements sans attente est perdue : commencer les tâches le plus tôt possible, n'implique pas forcément d'obtenir le résultat le plus rapidement. Ainsi l'ordonnancement optimal inclut potentiellement des délais d'attentes. Certaines machines peuvent être inactives alors qu'il reste des tâches prêtes.

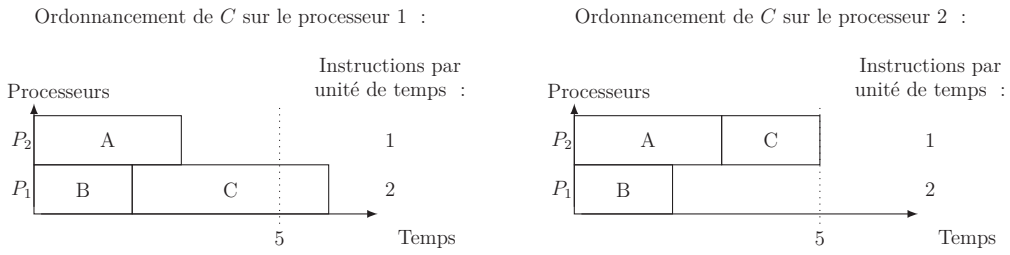


FIGURE 1.6 – Illustration d'un ordonnancement sur plate-forme hétérogène

Pour améliorer les performances des algorithmes de liste, l'ordonnancement d'une tâche n'est plus réalisé en fonction de la date de démarrage de la tâche mais en fonction de la date de fin. Le temps d'exécution de la tâche sur la machine est donc pris en compte dans le choix d'ordonnancement. Les algorithmes en-ligne, qui ne disposent pas de l'information du temps d'exécution des tâches, sont plus fortement impactés que les algorithmes hors-ligne.

Dans cette thèse, nous nous intéressons principalement aux ordonnancements en-ligne ; des travaux [5, 6] ont montré qu'il est possible d'adapter certains algorithmes d'ordonnancement en-ligne en autorisant des migrations ou des duplications de tâches. Nous détaillerons plus en avant ce point au moment de l'étude des ordonnancements en-ligne par vol de travail dans la section 3.

1.3 Relations de précédences et communications

Jusqu'à présent les applications parallèles présentées étaient constituées de tâches indépendantes représentant des application de type «*bag of tasks*» (BoT). Ce type d'applications correspond bien à certaines classes de parallélisme, comme par exemple le parallélisme de données où une même tâche est exécutée de manière indépendante sur différents jeux de données.

En revanche, de nombreuses applications ne peuvent pas être découpées efficacement en tâches indépendantes. Pour de telles applications, certaines tâches nécessitent le résultat d'autres tâches avant de pouvoir être exécutées. De telles dépendances impliquent d'une part des attentes potentielles et d'autre part une communication des résultats.

Pour prendre en compte les relations de précédences dans l'ordonnancement, nous détaillons dans la section 1.3.1 la modélisation des applications et les notations couramment utilisées dans la littérature.

Dans le modèle d'application, la quantité de données transférées entre deux tâches est détaillée. Pour obtenir le temps de transfert réel, un calcul doit être réalisé en fonction de la topologie de la plate-forme, des caractéristiques des liens réseaux, des protocoles utilisés, *etc*, ... Pour calculer ces temps, il existe plusieurs modèles permettant de prendre plus ou moins finement en compte les communications. La section 1.3.2 présente les modèles de communication les plus classiques ainsi qu'une analyse de leurs avantages et de leurs défauts.

1.3.1 Modélisation des applications

Une application parallèle est souvent représentée par un ensemble de tâches qui ont des relations de précédences entre elles. Les relations de précédences donnent un ordre partiel d'exécution des tâches. Une tâche ne peut pas être exécutée tant que toutes les tâches nécessaires à son exécution ne le sont pas. Lorsqu'elles le sont, la tâche est prête à être exécutée. Cette dernière est qualifiée par l'adjectif prête.

Une représentation des relations de précédences fréquemment utilisée en ordonnancement est le «Directed Acyclic Graph» (DAG) (Graphe acyclique orienté). Les nœuds du DAG représentent les tâches à exécuter. L'existence d'un arc (arête) entre deux nœuds représente l'existence d'une contrainte de précédence entre les deux tâches associées aux nœuds. Il est clair qu'il ne peut pas y avoir de cycles dans les relations de précédences sans quoi certaines tâches ne pourraient être exécutées.

Il existe d'autres modèles permettant de représenter des applications comme les «workflows» [92] (graphe de flots de données). Ce modèle permet d'ajouter principalement des conditions et des boucles. Cette représentation permet en autre pour réduire la taille du graphe de l'application.

Comme cela a été vu précédemment, les tâches n'ont pas un temps d'exécution identique. Chaque nœud du DAG est donc annoté en fonction du temps d'exécution de chaque tâche. Cette représentation est générique et permet d'inclure aussi les tâches sans précédence des problèmes abordés plus haut.

En plus des précédences, les tâches peuvent nécessiter des transferts de données. Pour prendre en compte les transferts de données dans la représentation, chaque arête est annotée d'une valeur qui correspond à la quantité de données. La figure 1.7 montre la représentation d'une application par un DAG annoté. Sur ce DAG, la tâche A est une source (un nœud n'ayant aucune arête entrante), et la tâche V est un puits (un nœud n'ayant aucune arête sortante).

Avant de présenter les algorithmes d'ordonnancement, nous introduisons ici quelques notations classiquement utilisées tout au long de cette thèse pour décrire et analyser les algorithmes.

Le DAG est généralement caractérisé par deux variables, une représentant la quantité de travail à effectuer, l'autre la longueur du chemin le plus long.

– W : Quantité de travail présente sur l'ensemble du DAG. De façon équivalente, cette quantité représente le temps d'exécution de l'application sur une machine, également notée T_1.

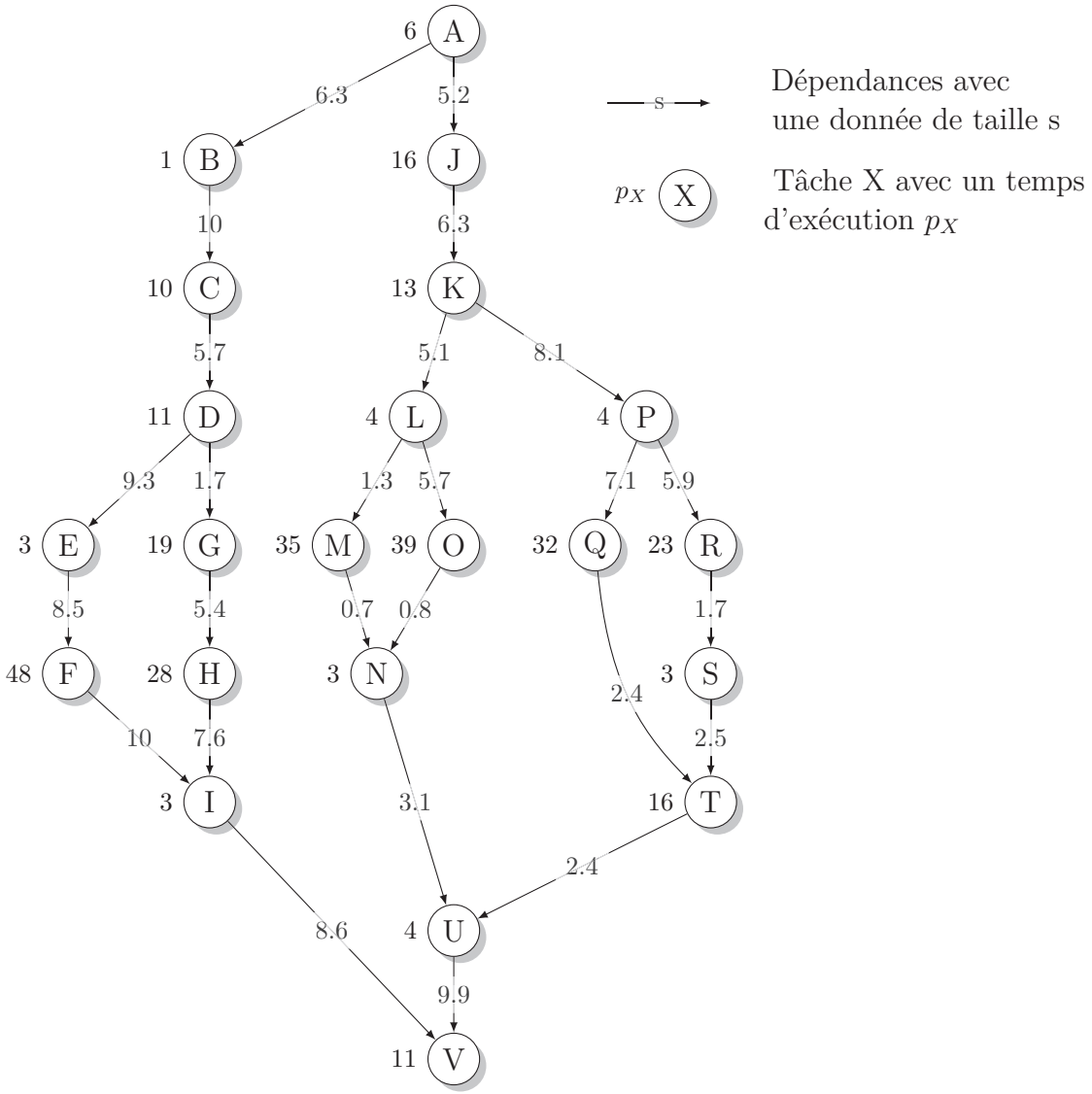


FIGURE 1.7 – Graphe G représentant une application.

– D : La longueur du chemin le plus long au sein du DAG (la somme des temps d'exécution sans les communications). Cette quantité représente aussi le temps d'exécution sur une infinité de machines en négligeant les communications, également notée T_∞.

Dans ce manuscrit, nous utilisons uniquement les notations W et D par souci de clarté. De plus, nous introduisons les notions de profondeur et de hauteur d'une tâche.

– La profondeur d'une tâche est la longueur du chemin le plus long entre une source et cette dernière notée t_level (de l'anglais «top level»).
– La hauteur d'une tâche est la longueur du chemin le plus long partant de celle-ci notée b_level (de l'anglais «bottom level»).

Il est fréquent dans ces deux dernières notations de prendre en compte le temps de calcul mais aussi le temps de communication entre les tâches. Nous prenons en compte dans ces notations uniquement la somme des temps d'exécution. Le tableau ci-dessous détaille ces notations pour le DAG de la figure 1.7.

Quantité de travail (W)	Chemin critique (D)	Hauteur de la tâche Q ($b\_level(Q)$)	Profondeur de la tâche Q ($t\_level(Q)$)
322	102	71	31

Les dépendances dans les DAG peuvent être quelconques, il n'y a pas de restrictions. Or en pratique, certaines méthodes de programmation comme la programmation récursive n'ont pas un schéma quelconque : un appel de fonction dépend uniquement de certains paramètres de la fonction f qui l'appelle et potentiellement de certaines résultats calculés par des fonctions déjà appelées par f. Ce schéma a une influence sur la complexité du problème considéré. Dans la notation à trois champs, il est habituel de détailler le type de structure utilisée. Nous détaillons trois structures que nous réutilisons dans certains algorithmes :

– tree : Le DAG est représentable par un arbre.
– fork-join : si un nœud a plusieurs fils, alors il existe un nœud regroupant les successeurs de ces fils. Cette structure représentant un programme récursif est largement utilisée dans des implantations classiques du vol de travail présenté section 3.2.
– prec : Aucune limitation sur le type de précédences.

Un exemple d'impact de la structure est visible lorsque les temps d'exécution sont unitaires. Si la structure est de type arbre, le problème est solvable en temps polynomial alors que le problème est NP-difficile si les relations de précédences sont quelconques.

1.3.2 Modélisation des communications

La parallélisation d'une application peut engendrer de nombreuses communications. Pour limiter l'impact de ces dernières, une méthode couramment utilisée est le recouvrement des communications par du calcul : durant la réalisation d'une communication, les machines exécutent des tâches.

Le recouvrement des communications par les calculs implique que les durées de celles-ci seront connues à priori, lors de l'ordonnancement des tâches. Ces durées dépendront de la structure du réseau de communication, des caractéristiques des liens, des machines impliquées et des autres communications réalisées en concurrence. Plusieurs modèles permettent une estimation de ces temps de communication plus ou moins réalistes.

Le modèle le plus simple existant est le modèle «délai», pour lequel le temps de communication est une fonction affine. Lorsqu'il n'y a pas de données à communiquer, le temps de transmission est la latence entre les machines impliquées. La pente de la fonction affine dépend du débit entre les machines. Pour que les communications suivent un tel modèle, la plate-forme doit comporter des liens de communications entre chaque machine. La structure de cette plate-forme est une clique. De plus, ce modèle est simple mais ne prend pas en compte plusieurs facteurs comme les coûts d'émission et de réception.

Pour prendre en compte ces coûts, le modèle *LogP* a été développé par Culler et al. [23]. Dans ce modèle, les coûts de communications sont séparés en quatre paramètres :

- L («Latency») : La latence du réseau.
- o («overhead») : Le surcoût logiciel à la réception et à l'émission des messages.
- g («gap») : Le temps entre la réalisation de deux envois d'un même processeur.
- P («processors») : Le nombre de processeurs mis en jeu.

Ce modèle est plus précis, il prend en compte les surcoûts logiciels lors d'une communication. Néanmoins, dans un réseau réel lorsque plusieurs communications passent par le même lien, des effets de congestion du réseau peuvent apparaître. La contention n'est pas prise en compte par le modèle *LogP*.

Aucun de ces modèles n'est parfait :
- le modèle «délai» est simple et couramment utilisé pour des analyses théoriques, mais il manque éventuellement de précision.
- le modèle «*LogP*» est plus précis que le modèle «délai» mais sa complexité fait qu'il est relativement peu utilisé lors d'analyses théoriques.

Il existe également des modèles réseaux plus complexes qui visent à prendre en compte la congestion utilisée pour le développement d'heuristiques comme le détail Sinnen dans le livre [75] chapitre 7. À notre connaissance, de tels modèles ne sont pas, en pratique, utilisés pour le développement d'algorithmes d'approximation.

Pour les algorithmes hors-ligne, les analyses théoriques considérant les communications présentées dans cette thèse, utilisent uniquement le modèle «délai». Malgré l'utilisation de ce modèle simple, la complexité des problèmes d'ordonnancement augmente lorsqu'il y a des communications. Par exemple, pour le problème $P|prec, c = 1, p_i = 1|C_{\max}$, Hoogeveen et al. [44] ont montré qu'il y a une borne d'inapproximation à $\frac{5}{4}$. Cette borne signifie qu'il n'existe pas d'algorithme polynomial ayant un ratio d'approximation meilleur que $\frac{5}{4}$ à moins que $P = NP$.

Ordonnancement avec précédences : hors-ligne 2

Sommaire du chapitre

Ce chapitre présente les algorithmes classiques de la littérature concernant le problème d'ordonnancement hors ligne avec précédences présenté en section 1.3.

Ces algorithmes nous permettrons d'évaluer finement les performances des algorithmes en-ligne, car pour le problème hors-ligne les performances atteignables sont généralement meilleures. Ces gains de performances sont obtenus grâce à l'utilisation des temps d'exécution des tâches, des quantités de données transférées et des caractéristiques de la plate-forme d'exécution.

De plus, les algorithmes hors-ligne permettent de gérer finement les communications. Durant l'exécution, les communications peuvent être réalisées en parallèle de l'exécution de certaines tâches. Ainsi, leur impact sur le temps d'exécution est plus faible. L'analyse des algorithmes hors-ligne nous donnera des pistes de réflexion pour améliorer la gestion des transferts de données dans les algorithmes en-ligne.

Nous détaillons dans un premier temps (section 2.1) l'adaptation des ordonnancements par liste aux problèmes avec précédences et communications. Enfin, nous introduisons dans la section 2.2 les algorithmes d'agrégation regroupant les tâches selon les communications. Ces ordonnancements ont été développés spécifiquement pour réduire l'impact des communications.

2.1 Ordonnancement par liste avec précédences

Les algorithmes d'ordonnancement par liste ont été introduits dans la section 1.2.1 sur des exemples simples sans précédences. Ces algorithmes développés par Graham [40]

considèrent un ensemble de tâches avec précédences. Ensuite, Hwang *et al.* [45] ont adapté et analysé le fonctionnement d'une partie de ces algorithmes pour gérer les communications.

Dans cette présentation des algorithmes de liste, la section 2.1.1 commence par définir les principes communs à ceux-ci. Les modifications apportées pour prendre en compte les communications impactent plus le fonctionnement général de l'algorithme. Nous analysons donc plus spécifiquement quelques unes de ces heuristiques dans la section 2.1.2.

2.1.1 Principes communs des ordonnancements par liste

Dans les problèmes d'ordonnancement abordés, l'objectif est de minimiser le temps de complétion. Dans le cas où les communications ne sont pas prises en compte, cet objectif est lié à la minimisation des temps d'inactivité de chacun des processeurs. Dans l'objectif de minimiser les temps d'inactivité, les algorithmes de liste ne laissent pas de processeurs inactifs si des tâches prêtes sont présentes. La différence principale avec le cas sans précédences réside dans le déblocage de certaines tâches après l'exécution de leurs prédécesseurs. Ainsi, des tâches deviennent prêtes au cours de l'exécution.

Les algorithmes introduit par Graham gèrent une liste de tâches prêtes qui est mise à jour en fonction de l'exécution de celles-ci. L'ordonnancement est calculé en considérant une horloge virtuelle. Le fonctionnement est proche de celui d'une simulation. L'avancement de l'horloge s'effectue en pas de temps liés aux fins d'exécution des tâches. L'ajout d'une tâche dans la liste est réalisé lorsque tous ses prédécesseurs sont exécutés à la date considérée. Cette simulation permet de palier le fait que l'ordonnancement est calculé avant l'exécution. Nous fournissons une description plus formelle du fonctionnement des ordonnancements par liste dans l'annexe 4(page : 152).

Les algorithmes de liste se différencient au moment de choisir la tâche à exécuter. Pour l'ordonnancement de tâches indépendantes, l'influence de ce choix sur les performances a été mis en évidence par l'amélioration du ratio d'approximation. Ce ratio est classiquement égal à 2 pour l'ensemble des algorithme d'ordonnancement par liste. Il est réduit à $\frac{4}{3}$ pour l'algorithme Largest Processing Time (LPT) détaillé dans la section 1.1.2. Pour les tâches avec précédences, le ratio d'approximation ne diffère pas en fonction du choix de la tâche or l'impact sur les performances est généralement visible en pratique.

Le ratio d'approximation pour l'ordonnancement de tâches avec précédences est aussi égal à 2. La preuve peut être obtenu en adaptant la preuve présentée section 1.2.1. Elle a été introduite par Graham [40]. Celle-ci montre que des processeurs sont inactifs à la date t si et seulement si un des processeurs actifs exécute une tâche appartenant au chemin critique à ce même instant. La figure 1.5 (page : 16) le met en évidence pour les tâches indépendantes. Dans ce cas, le chemin critique est modélisé par la tâche avec le temps d'exécution le plus important. Les temps d'inactivité pour l'ensemble des processeurs peuvent être bornés par $D \times (m-1)$ car D est le temps maximal passé à exécuter des tâches sur le chemin critique. Durant l'exécution de ces tâches, il y a toujours au moins une machine active. De plus, le temps passé à travailler par

l'ensemble des processeurs est égal à W. Avec ces bornes, il est possible de montrer que $C_{max} \leq \frac{W}{m} + D * (1 - \frac{1}{m})$. Cette preuve n'est en revanche pas valable dans le cas où il y a des communications.

Dans tous les algorithmes de liste présentés ci-dessous, le coût des communications est déterminé par le modèle délai introduit section 1.3.2. Cette modélisation est utilisée, pourtant elle ne prend pas en compte la congestion du réseau. Les communications présentes entre les tâches peuvent retarder le début de l'exécution d'une tâche lorsque celle-ci n'est pas exécutée sur le même processeur que ses prédécesseurs. Ainsi, les communications introduisent des temps d'inactivité supplémentaires. Afin de réduire ces temps d'inactivité, les temps de communications sont généralement inclus dans le choix de la tâche à exécuter comme dans les algorithmes présenté ci-dessous.

Les temps d'inactivité dûs aux coûts de communications complexifient les analyses théoriques. Ces dernières ne sont pas valable pour l'ensemble des algorithmes de listes, mais uniquement pour l'algorithme Earliest Task First (ETF) [45] présenté section 2.1.2.

2.1.2 Heuristiques : ordonnancement par liste avec coût de communications

Les heuristiques hors-ligne présentées dans cette section tentent de minimiser le temps d'exécution global qui est impacté par les coûts de communications. Ces coûts influenceront la tâche sélectionnée mais aussi le processeur.

La sélection du processeur est affectée par les communications car si la tâche est exécutée par le même processeur que ces prédécesseurs, le coût de communication est nul. La sélection de la tâche est elle aussi affectée par les communications : en provenance de ces prédécesseurs et en direction de ces successeurs. Bien sûr, tous ces coûts varient en fonction de la quantité de données à transférer.

Nous commençons par l'heuristique ETF [45] dont le temps d'exécution est borné théoriquement, puis Heterogeneous Earliest Finish Time (HEFT) [83] qui est fréquemment utilisé comme base de comparaison et enfin «Sufferage» [50] dont le choix de la tâche sort du schéma classique des ordonnancements par liste.

Earliest Task First (ETF)

L'objectif de cette heuristique est de minimiser les temps d'inactivités des processeurs à un instant donné, sans même se soucier de l'impact de ces choix sur la suite de l'exécution.

Ainsi lorsqu'un processeur devient inactif, l'algorithme sélectionne la prochaine tâche à exécuter. Pour chacune des tâches prêtes, l'algorithme calcule la date à laquelle la tâche considérée peut être exécutée le plus tôt possible. Cette date est calculée en fonction des relations de précédences, des communications et de la disponibilité des processeurs. L'algorithme considère la date minimale avant laquelle aucune tâche ne peut être exécutée. Si un processeur devient inactif avant cette date, l'horloge virtuelle est avancée au prochain passage d'un processeur à l'état inactif. La liste de tâches prêtes est mise à jour et l'algorithme recommence le processus de sélection.

Si aucun processeur devient inactif avant la date minimale, une des tâches pouvant commencer à cette date est sélectionnée. La tâche est ordonnancée sur le processeur minimisant la date de début d'exécution. Puis l'algorithme recommence le processus de sélection. Une description formelle de cet algorithme est fournie en annexe A.1.2 (page : 153).

L'ordonnancement des tâches est effectué avant le début de l'exécution. Ainsi, les transferts de données peuvent commencer dès la fin de l'exécution de la tâche qui les produit. En transférant les données dès leur production, l'impact des communications est minimisé. De plus, les données peuvent être transférées pendant que les processeurs émetteur et récepteur exécutent d'autres tâches.

Les performances des algorithmes sélectionnant une des tâches qui commencent le plus tôt ont été bornées par un ratio d'approximation égal à $2 + C$ [45], où C est la somme des temps de communications sur le chemin comportant la plus grande quantité de communications.

Heterogeneous Earliest Finish Time (HEFT)

Cette heuristique est largement utilisée comme référence dans les analyses de performances par simulation car les ordonnancements obtenus ont un temps d'exécution faible [83, 93, 64, 41]. Lors de l'ordonnancement, la liste de tâches prêtes est triée en fonction de la hauteur des tâches. Plus une tâche a une hauteur importante, plus elle est prioritaire. Une description formelle de cet algorithme est détaillée en annexe A.1.3 (page : 154).

L'objectif de cet ordre est d'exécuter les tâches les plus éloignées des puits du «Directed Acyclic Graph» (DAG) en premier pour éviter d'importants temps d'inactivité en fin d'exécution. Cette heuristique évite ainsi le pire des cas, illustré figure 1.5 pour les tâches indépendantes. Ce choix est comparable à celui de LPT mais adapté aux tâches avec précédences.

Le calcul de l'éloignement d'une tâche par rapport aux puits est effectué en considérant le chemin le plus long entre la tâche et un puits. La longueur du chemin prend en compte les coûts de communications et les temps d'exécution. Lors de ces calculs, il serait difficile de prendre en compte la congestion du réseau, même si le modèle réseau le permettait.

Par exemple, lors du calcul de la hauteur d'une tâche, le placement des tâches est complètement inconnu. Sans ce placement, il n'est pas possible de connaître la quantité de données qui transiteront par un lien réseau et donc de modéliser la congestion du réseau future. De plus, de manière identique à ETF sur plate-forme homogène, l'algorithme place la tâche sélectionnée sur le processeur minimisant sa date de fin d'exécution. Le calcul de cette date est réalisé en fonction des transferts de données nécessaires. Le temps de transfert dépend de la congestion du réseau à la date considérée mais pas seulement. Après le placement de cette tâche, l'algorithme peut en ordonnancer une autre juste après qui nécessitera elle aussi des transferts de données. Ces derniers risquent eux aussi de modifier les quantité de données transitant par les liens de communications et ainsi perturber les transferts réalisés pour la tâche ordonnancée précédemment.

La figure 2.1 illustre ce risque de perturbation. Lors du placement de la tâche C, la date de début d'exécution de la tâche A peut être retardée à cause d'une congestion réseau. Celle-ci est due au transfert des données de la tâche C placée après la tâche A.

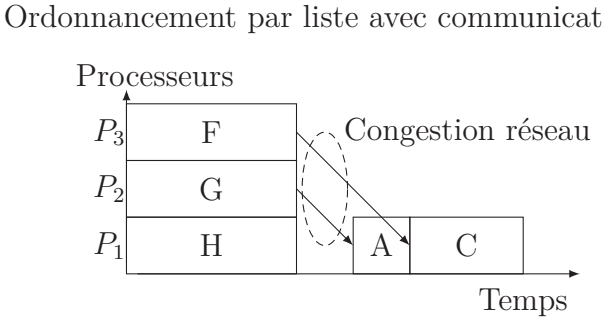


FIGURE 2.1 – Perturbation sur la date de démarrage d'une tâche placée précédemment.

Ainsi, même si le modèle réseau considérait la congestion du réseau, les algorithmes devraient être fortement modifiés pour la prendre en compte.

Enfin, cet algorithme s'adapte à l'hétérogénéité des machines en considérant le temps de fin d'exécution des tâches plutôt que le temps de démarrage comme cela est présenté en section 1.2.2 page 16 pour les tâches indépendantes.

L'heuristique Sufferage

L'objectif lors du placement des tâches sur les machines est de minimiser leur temps de démarrage dans l'ordre de la liste. Or, si l'ensemble des tâches de la liste est considéré, plusieurs d'entre elles obtiennent leur ordonnancement au plus tôt sur la même machine. Le problème est de savoir à quelle tâche sera allouée la machine.

Les heuristiques précédentes ont pour objectif d'optimiser l'exécution d'une tâche bien précise sans se soucier de l'impact de ce choix sur les temps d'exécution des autres.

Sufferage tente au contraire de minimiser les perturbations sur les autres tâches. Par exemple, sur la figure 2.2, les tâches A et B commencent au plus tôt sur la machine 1 à la date 1. Or, si elles sont exécutées sur la seconde machine, la tâche B commence à l'instant 4 et la tâche A à l'instant 2. Ainsi, la tâche B serait plus impactée par le placement de A sur la machine 1 que l'inverse. La tâche B est donc ordonnancée sur le processeur 1. Dans le cas général, pour chaque tâche, l'heuristique calcule le préjudice subi en n'étant pas exécutée sur le processeur le plus adapté.

L'assignation des tâches est réalisée itérativement. À chaque étape, l'ordre de la liste de tâches prêtes est recalculé en fonction de la valeur préjudice. L'assignation d'une tâche sur le processeur le plus adapté est réalisée si celui-ci n'a pas encore été assigné. Dès que toutes les tâches ont été traitées, l'algorithme passe à l'itération suivante en considérant les placements réalisés comme définitifs.

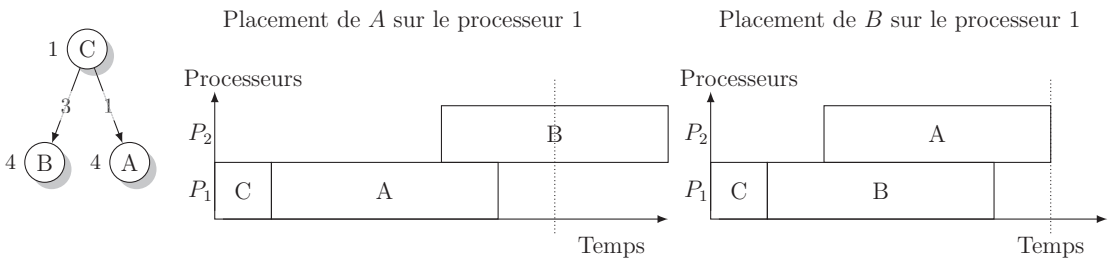


FIGURE 2.2 – Impact du placement d'une tâche sur l'ordonnancement des autres tâches.

Ordonnancement par liste et congestion du réseau

Dans les analyses théoriques des algorithmes, le modèle de coût de communications ne prend pas en compte la congestion au sein du réseau, de plus celle-ci est souvent difficile à considérer dans les décisions d'ordonnancement.

Par exemple, HEFT utilise les coûts de communications dans le calcul des priorités et durant l'assignation de la tâche (date de début d'exécution). Or, le calcul des priorités est réalisé sans connaître l'ordonnancement. Ceci empêche donc de prendre en compte la congestion et/ou la structure du réseau.

Dans ce cas, les coûts de communications sont calculés en négligeant la congestion et en considérant un réseau homogène sur l'ensemble de la plate-forme pour être indépendant des machines qui exécuteront les tâches.

De plus, l'assignation d'une tâche sur un processeur peut engendrer des communications qui perturberont potentiellement les choix réalisés auparavant. Cette perturbation est mise en évidence par la figure 2.1. Ainsi, lors de l'utilisation de ces algorithmes sur plates-formes réelles, leurs performances risquent d'être impactées par la congestion du réseau.

Dans ces algorithmes d'ordonnancement, le placement est effectué pour ordonnancer la tâche au plus vite. Or, ce placement peut impacter fortement la suite de l'exécution. Pour minimiser l'impact du réseau, il existe des méthodes d'agrégation qui regroupent les tâches en fonction des communications.

2.2 Agrégation de tâches

Les algorithmes d'agrégation de tâches («task clustering») ont été développés dans l'idée que le nombre de machines sera suffisant pour exploiter la totalité du parallélisme des applications. Ainsi dans le problème d'ordonnancement, l'algorithme peut supposer que le nombre de machines est infini. Dans ce cas, le placement d'une tâche par machine est une solution envisageable. Avec un tel ordonnancement, le temps d'exécution est égal au chemin critique de l'application en accumulant le temps d'exécution des tâches et les temps de communications. Pour minimiser le temps d'exécution global, l'algorithme peut influencer le placement des tâches. Il peut ainsi regrouper l'exécution de certaines tâches sur la même machine pour éviter les coûts de communications entre deux tâches.

L'objectif est de limiter les communications qui impactent le temps d'exécution global sur une infinité de machines.

Nous détaillons le fonctionnement de ces algorithmes car les problèmes d'ordonnancement auxquels nous nous intéressons incluent des communications. Leur fonctionnement est donc une source potentielle d'inspiration pour le développement d'ordonnancements en ligne.

Nous présentons deux types d'algorithmes différents utilisant un mécanisme d'agrégation de tâches. Le premier présenté en section 2.2.1 cherche à regrouper les tâches de l'application en groupes indépendants. Lorsque l'un de ces groupes est en cours d'exécution, il ne nécessite aucune communication. Le second algorithme présenté dans la section 2.2.2 identifie les communications impactant le temps d'exécution sur une infinité de machines pour éviter de transférer les données.

Après avoir regroupé les tâches sans considérer le nombre de machines, la section 2.2.3 présente quelques méthodes existantes pour ordonnancer ces groupes de tâches sur un nombre fini de machines.

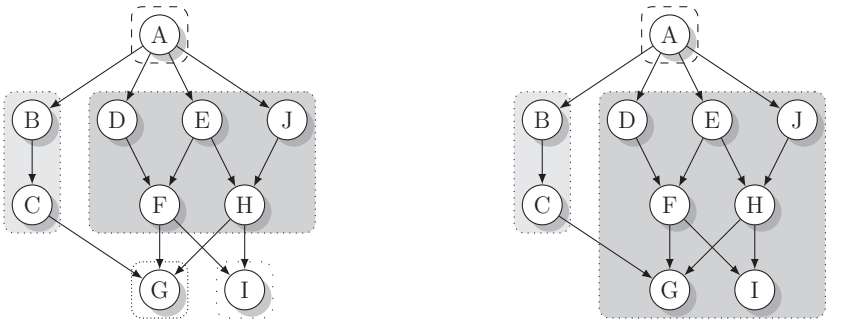
2.2.1 Regroupement de tâches en partie indépendante

Certaines tâches d'une application peuvent nécessiter des quantités de données importantes en comparaison à leur temps d'exécution. Les méthodes regroupant les tâches en partie indépendante permettent de réduire le rapport entre les quantités de données à transférer et le temps d'exécution du groupe. Cette décomposition peut être considérée comme un ordonnancement pour une infinité de machines. Au contraire dans cette partie, les méthodes ne cherchent pas à optimiser directement le temps d'exécution. Elles regroupent les tâches qui nécessitent des transferts vers ou/et depuis un ensemble commun de tâches.

Le premier découpage est basé sur des ensembles de tâches appelés «CLANS» [52]. Un «CLANS» est un agrégat dont les prédécesseurs et les successeurs de toutes les tâches communiquantes avec l'extérieur sont identiques. La figure 2.3 illustre un découpage en «CLANS». Deux «CLANS» indépendants peuvent être exécutés sans réalisation de communications entre la première et la dernière tâche. Cette méthode d'agrégation contraint fortement les groupes de tâches car celles d'un même groupe ont obligatoirement un comportement identique aux autres du point de vue des communications.

Cette idée de réaliser des agrégats indépendants a été reprise en définissant des agrégats convexes [49] qui sont moins contraints. Le découpage d'un DAG en ensembles est convexe si le graphe constitué par les agrégats (un ensemble est équivalent à une tâche) est un DAG. Cette définition est plus souple que celle des «CLANS», car il est possible d'avoir des dépendances différentes entre les tâches tant qu'il n'existe pas de cycle entre les ensembles de tâches. La figure 2.4 montre deux découpages d'un DAG dont un est convexe et l'autre non.

Pour de tels agrégats, Lepère et al. [49], ont montré qu'il existe un découpage fournissant une 2-approximation pour l'ordonnancement des tâches sur un ensemble de machines homogènes. En revanche, la recherche de ce découpage requiert une énumération exhaustive de toutes les découpes. Des heuristiques de décomposition



Découpage en “CLANS” Découpage en regroupement convexe

FIGURE 2.3 – Différence entre les regroupements de type : «CLANS» et «convexe».

Regroupement convexe Regroupement non convexe

FIGURE 2.4 – Illustration de deux regroupements : convexe et non convexe.

récursive fournissant une découpe récursive en agrégats convexes ont été développées et testées [49, 57]. Nous fournissons une description en pseudo-code dans l'annexe A.2.1 (page : 155). La figure 2.5 montre sur un exemple le découpage récursif d'un DAG en agrégats convexes. Ces heuristiques ne fournissent pas toujours une découpe avec un ratio d'approximation égal à 2.

Cette méthode de décomposition en partie indépendante est intéressante pour limiter de manière significative les transferts de données. De plus, ces heuristiques n'optimisent pas l'ordonnancement par rapport au chemin critique. Cette différence est intéressante car la plupart des algorithmes d'ordonnancement tentent de réduire l'impact du chemin critique. Cette méthode aborde ainsi le problème d'une manière transversale.

2.2.2 Réduction de l'impact des communications

Contrairement à l'algorithme d'ordonnancement par liste développé pour limiter les temps d'inactivité, les méthodes d'approximation par agrégation de tâches cherchent à limiter directement l'impact des communications sur le temps d'exécution global.

Les mécanismes d'agrégation de tâches tentent de regrouper celles qui communiquent de grandes quantités sur la même machine. Ces regroupements s'effectuent en fonction de l'impact des communications sur le temps d'exécution global obtenu sur une infinité de machines. Chaque tentative d'agrégation entraîne ainsi un nouveau calcul du temps d'exécution pour mettre en valeur son impact. Pour un nombre de machines infini, la question de l'ordonnancement des différents agrégats sur les machines ne se pose pas. En revanche, l'ordre d'exécution des tâches dans les agrégats doit être défini. Ainsi, le problème d'ordonnancement est restreint à la définition de l'ordre des tâches

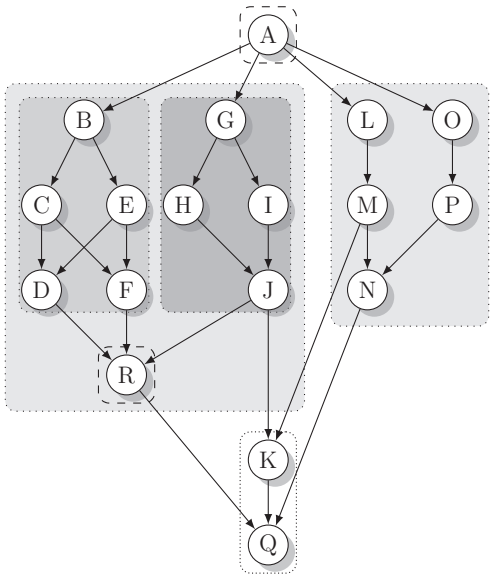


FIGURE 2.5 – Découpage réalisé avec l'algorithme récursif de regroupement convexe.

dans chaque agrégat. Ce gain en complexité permet de limiter le temps de calcul de l'ordonnancement et d'augmenter sa précision même si ce problème d'optimisation reste NP-difficile. Après avoir défini l'ordre des tâches dans les agrégats en ajoutant des arêtes de poids nul dans le DAG, le temps parallèle de l'application est égal à la longueur du chemin critique en considérant les coûts de communications.

Il existe de nombreuses heuristiques différentes pour réaliser l'agrégation des tâches et définir l'ordre d'exécution des tâches dans les agrégats. Leur objectif est d'obtenir le meilleur temps d'exécution sur un nombre non borné de machines, par exemple : Dynamic critical-path (DCP) [47], Edge Zeroing (EZ) [71], Dynamic-level (DL) [74], Mobility directed (MD) [89], Dominant Sequence Clustering (DSC) [91].

Nous avons choisi de présenter dans la section 2.2.2 l'algorithme de «Edge Zeroing» (EZ), qui est la première méthode simple d'agrégation de tâches développée par Sarkar. Elle permet de détailler le fonctionnement général qui a été ensuite réutilisé dans les autres algorithmes. Le second algorithme présenté dans la section 2.2.2 détaille le fonctionnement de DSC dont les longueurs des ordonnancements sont bornées théoriquement. De plus cette méthode a été implantée dans le logiciel *PYRROS* [79] qui fournit aussi un algorithme de placement des agrégats sur un nombre fini de machines.

Edge Zeroing (EZ)

Dans un premier temps, les arêtes du DAG sont triées dans l'ordre décroissant des coûts de communications. Itérativement dans l'ordre décroissant, l'algorithme cherche pour chaque arête à connaître l'impact de l'agrégation de la tâche qui produit les données et de celle qui les lit. Dans un premier temps, nous supposons que chacune des tâches est seule dans l'agrégat les contenant. Si les tâches sont exécutées par la même machine, la communication entre les tâches est considérée comme nulle. Pour évaluer l'impact sur le temps d'exécution, il suffit de comparer la longueur des chemins

critiques des deux DAG : celui avec l'agrégation et celui sans. Si la longueur du chemin critique est inférieure ou égale à celle du DAG sans agrégation, elle est conservée.

Nous avons expliqué le principe pour l'agrégation pour deux tâches, mais il peut y en avoir plusieurs dans chaque agrégats. Dans ce cas, les agrégats sont fusionnés en un seul. Lors de cette fusion, certaines des tâches appartenant aux deux agrégats peuvent être indépendantes (il n'existe pas de chemin liant les deux tâches). L'ordre d'exécution de ces tâches par la machine qui exécutera l'agrégat fusionné doit être défini. Ce choix peut avoir un impact sur le temps d'exécution.

Pour choisir cet ordre, Sakkar a proposé une heuristique basée sur la hauteur de la tâche (b_level). La hauteur des tâches est calculée avant la fusion des deux agrégats. Celle-ci définit l'ordre d'exécution des tâches indépendantes : les tâches les plus hautes sont exécutées en premier. Une arête fictive (sans communication) est ajoutée entre les tâches indépendantes pour définir un ordre.

En fonction de cet ordre, le temps d'exécution du DAG en parallèle est calculé en considérant que l'agrégat résultant de la fusion est exécuté par une machine. De manière équivalente, ce temps est mesuré avec la longueur du chemin critique en fournissant un poids nul aux arêtes incluses dans un agrégat.

Si le temps parallèle avec cette agrégation est inférieur ou égal au temps d'exécution sans effectuer la fusion, l'agrégation est conservée pour l'étape suivante. Afin d'avoir une compréhension plus précise du fonctionnement, nous fournissons une description formelle de l'algorithme dans l'annexe A.2.2 (page : 156).

Dominant Sequence Clustering (DSC)

L'algorithme de EZ ne prend pas en compte le chemin critique. Seul le temps des communications est considéré. Certaines étapes de l'algorithme ne réduisent pas strictement la longueur du chemin critique. Dans cette heuristique, l'objectif est de regrouper les tâches qui se trouvent sur le chemin critique. Ainsi, chaque agrégation aura une influence directe sur le temps d'exécution en parallèle. L'exécution d'un tel algorithme peut être coûteux en temps de calcul. DSC tente donc de mettre en oeuvre cette idée en limitant au maximum la complexité de l'algorithme.

Pour expliquer l'algorithme, nous proposons d'en introduire une version simple et de détailler une partie des idées d'optimisation utilisées.

L'ensemble des tâches non traitées est mis dans une liste. Pour tous les éléments de la liste, la longueur du plus long chemin passant par l'élément est calculée. Une des tâches ayant la priorité maximale, et dont tous ses prédécesseurs ont été traités, est sélectionnée. L'algorithme tente de regrouper celle-ci avec un des agrégats contenant au moins un des prédécesseurs. Cela a pour effet de considérer les communications internes à l'agrégat comme nulles. L'agrégat minimisant la profondeur (t_level) de la tâche est choisi. La tâche est placée dans un nouvel agrégat, dans le cas où la profondeur minimale obtenue par fusion est supérieure à celle obtenue sans fusion. Une fois celle-ci traitée, la priorité des tâches est mise à jour avec le coût des communications qui convient. Cette version de l'algorithme est détaillée en pseudo-code dans l'annexe A.2.3 (page : 157).

Cette version de l'algorithme donne l'ordonnancement optimal pour les anti-arbres (*i.e.,* une source unique). Dans ce cas, un agrégat est créé en regroupant les tâches les plus prioritaires tant que la quantité de travail dans l'agrégat est inférieure au temps de communication. Ensuite, les tâches sont mises dans des agrégats distincts. En revanche, pour les arbres, l'algorithme se limite à l'agrégation d'une seule tâche avec le puits. Ces deux exemples sont représentés figure 2.6.

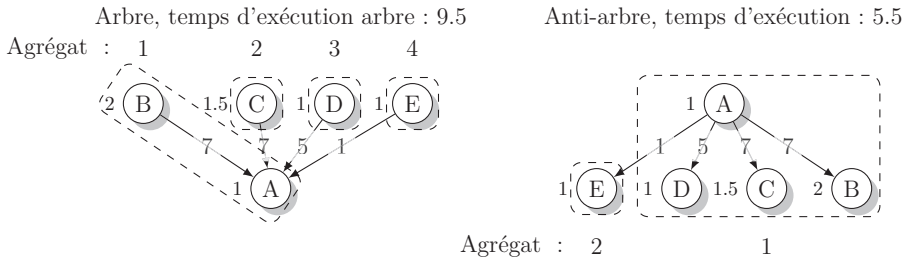


FIGURE 2.6 – Regroupements obtenus avec la version initiale de DSC.

Pour améliorer l'algorithme, l'idée est de regrouper non pas avec un agrégat qui minimise la nouvelle profondeur, mais avec plusieurs agrégats sous certaines conditions. Pour plus de détails voir [90].

DSC est le premier algorithme fonctionnant par agrégation dont la longueur de l'ordonnancement est borné théoriquement. Il est optimal pour les arbres et les anti-arbres. De plus quelque soit l'application, son ratio d'approximation est égal à $1 + \frac{\min p_i}{\max c_{ij}}$ où c_{ij} est le coût de communication entre les tâches i et j et p_i le temps d'exécution de la tâche i.

2.2.3 Repliement sur m machines

Nous avons présenté deux manières différentes d'obtenir des agrégats de tâches pour une application (EZ et DSC). Ces méthodes tentent de limiter l'impact des communications sur le temps d'exécution global. Ces agrégats peuvent être aussi considérés comme un ordonnancement sur une infinité de machines. Lors de l'exécution sur une plate-forme, le nombre de processeurs est rarement supérieur au nombre d'agrégats. Ainsi, il n'est pas possible d'ordonnancer chaque agrégat sur un processeur distinct comme cela était considéré lors de leur création. Certains processeurs exécuteront donc plusieurs agrégats. Cette phase de placement des agrégats sur les processeurs est appelée repliement.

Celui-ci consiste à définir l'ordonnancement final des tâches en considérant les agrégats préalables. Malgré cette découpe pour éviter de nombreuses communications, le calcul de l'ordonnancement reste NP-difficile.

Durant la phase de repliement, l'algorithme a pour objectif de conserver les optimisations des phases d'agrégations. L'algorithme choisi dans un premier temps d'ordonnancer chaque agrégat sur un processeur. Certains processeurs auront à exécuter plusieurs agrégats. Pour montrer l'impact d'un tel choix, nous utilisons un exemple avec une

application dont la structure est un arbre. Sur celle-ci, nous avons choisi le regroupement fourni par DSC qui ordonnance optimalement ce type d'application sur une infinité de machines. La figure 2.7 représente l'application avec les agrégations réalisées par l'algorithme DSC. Sur cet exemple, si l'algorithme de repliement se limite à ordonnancer les agrégats sur les processeurs, l'ordonnancement optimal n'est pas atteignable. Nous pouvons le vérifier ici car l'instance est petite. Le meilleur ordonnancement des agrégats fourni une performance égale à $4 + 3 \times \epsilon$ tandis que l'optimal a un temps d'exécution égal à $3 + 3 \times \epsilon$. Ainsi, les méthodes de repliement ont tendance à considérer l'ensemble des informations présentes dans le DAG pour réaliser l'ordonnancement.

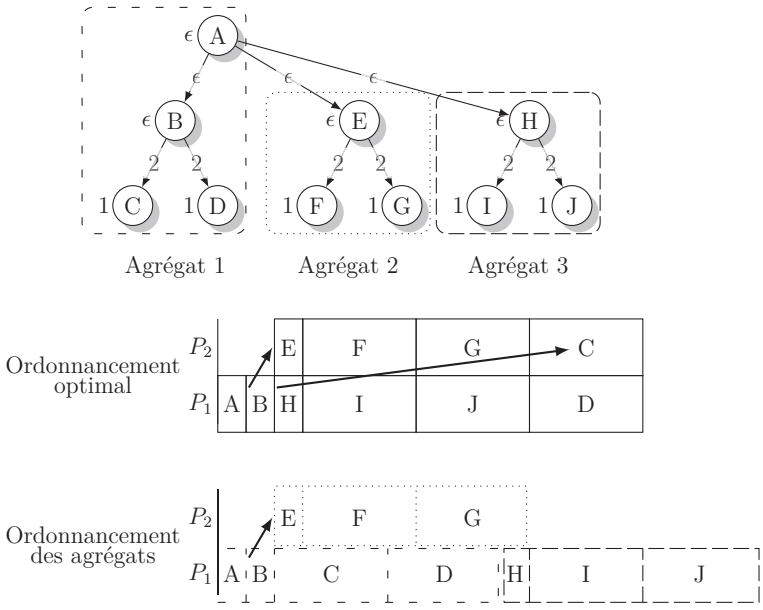


FIGURE 2.7 – Ordonnancement des agrégats obtenus avec l'algorithme DSC.

L'optimisation de cette phase est souvent délicate et dépend généralement de l'application. Parmi les méthodes de repliement existantes, le logiciel *PYRROS* implantant l'algorithme DSC, fourni aussi un algorithme de repliement qui conserve les regroupements préalables. Le fonctionnement de cette méthode est assez simple. À chaque tâche est associée une priorité qui est égale à sa hauteur en négligeant le coût des communications internes aux agrégats. Dès qu'une tâche d'un agrégat est placée sur un processeur les autres le sont aussi. Pour illustrer son efficacité, la figure 2.8 montre l'ordonnancement obtenu dans le cas où le placement optimal peut être calculé en exécutant un agrégat par un processeur unique. L'algorithme implanté dans *PYRROS* transfère des quantités de données significativement plus importantes que l'ordonnancement optimal.

Cette phase de repliement est assez complexe et a un rôle non négligeable sur les performances, comme l'étude [63] le montre pour l'algorithme d'agrégation DSC. Les algorithmes comparés dans cette étude cassent les agrégats pour réaliser le repliement. Dans ces conditions, il est possible de s'interroger sur le gain de la phase d'agrégation.

Les algorithmes d'agrégation présentés ci-dessus utilisent les annotations du DAG

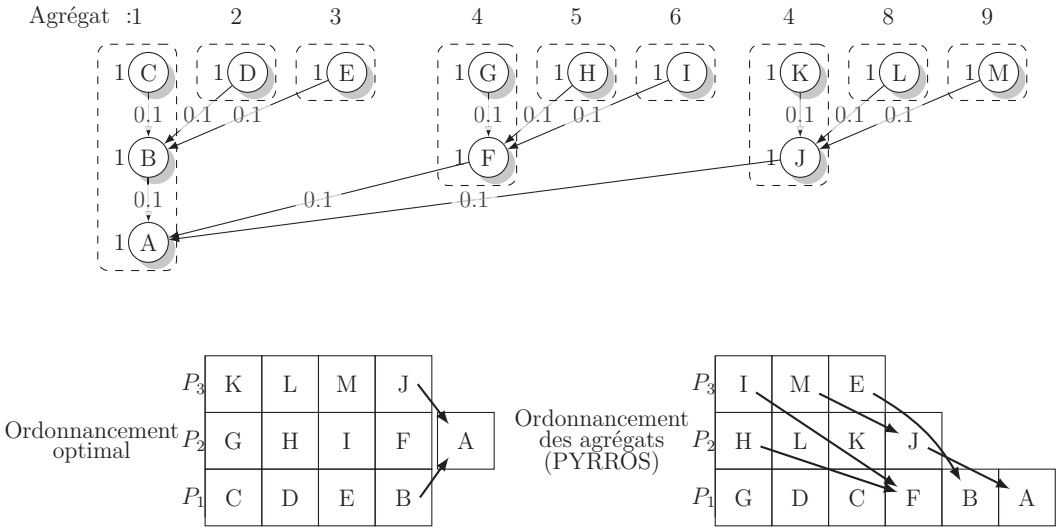


FIGURE 2.8 – Repliement des agrégats sur 3 processeurs.

pour regrouper les tâches. Dans le cadre de nos travaux présentés section 4, les annotations ne sont pas connues. Les méthodes comme DSC qui utilisent le chemin critique comme base de fonctionnement sont difficilement adaptables. En revanche, durant l'exécution, l'algorithme d'ordonnancement tente d'extraire une partie du travail pour l'exécuter sur un processeur distant. Lors de cette extraction, l'algorithme pourrait s'appuyer sur des agrégats convexes obtenus avec l'algorithme de découpe récursive. De plus comme nous le verrons dans la section 3.2.1, les schémas de programmations récursives s'adaptent bien aux ordonnancements en-ligne étudiés.

Ordonnancement en-ligne avec précédences 3

Sommaire du chapitre

Dans les problèmes hors-ligne présentés précédemment, les temps d'exécution des tâches et les quantités de données transférées sont connus avant l'exécution. Or lors de l'exécution d'une application sur une plate-forme réelle, des perturbations peuvent se produire à tout instant. Ces perturbations peuvent provenir d'évènements extérieurs tels que : le partage de la machine avec un autre utilisateur, le partage des liens réseaux... Ainsi, l'utilisateur n'est pas en mesure de prévoir ces perturbations. De plus, l'application peut être dynamique et dépendre de résultats aléatoires. Dans ce cas, le graphe de tâches est découvert à l'exécution. Les algorithmes en-ligne doivent être capable de s'abstraire de la connaissance de l'application pour ordonnancer les tâches.

Parmi les algorithmes en-ligne, la section 3.1 présente l'algorithme glouton et celui de partage de travail. Ces algorithmes ont des lacunes :

- l'algorithme glouton utilise une liste centralisée. Lors de l'accès à la liste, les processeurs doivent attendre leur tour pour obtenir du travail.
- l'algorithme de partage de travail est basé sur une répartition du travail entre les machines au cours de l'exécution. Cette répartition est basée sur la quantité de travail présent sur l'ensemble de la plate-forme. Avec une valeur erronée, il peut y avoir deux conséquences différentes. Si la valeur considérée est inférieure à celle

réelle, un grand nombre de migrations de travail risque d'avoir lieu. Dans le cas contraire, un déséquilibre subsistera.

Le mécanisme de vol de travail présenté dans la section 3.2 est basé sur l'algorithme glouton mais avec une gestion distribuée de la liste de tâches. Cela lui permet de limiter la contention lors de l'accès au travail. De plus, le vol de travail réagit aux inactivités des processeurs pour transférer des tâches d'un processeur à l'autre. Le risque de transférer un travail à multiples reprises est évité en l'exécutant dès sa réception.

En revanche, le mécanisme classique de vol de travail considère que le temps d'accès aux données ne varie pas en fonction du processeur (sauf pour le processeur ayant créé la tâche). Or, l'accès aux données peut varier en fonction du processeur sur les nouvelles architectures. Par exemple sur une grille, les processeur de la même grappe de calculs communiquent plus rapidement entre eux. La section 3.3 analyse les méthodes existantes pour prendre en compte la hiérarchie de la machine dans les algorithmes de vol de travail.

3.1 Ordonnancement en-ligne et transferts de données

Les algorithmes en-ligne permettent de réagir au cours de l'exécution à des évènements qui n'étaient pas prévus. Ces algorithmes ont une connaissance partielle du système. Nous présentons dans la section 3.1.1 un ordonnancement glouton. Cette section détaille aussi le comportement de l'algorithme en présence de communications entre les tâches. Puis, la section 3.1.2 introduit l'algorithme de partage de travail. Cet algorithme a l'avantage d'être distribué et donc d'éviter de nombreuses synchronisations. En revanche, il nécessite plus d'informations sur l'application comme la quantité de travail globale.

3.1.1 Ordonnancement glouton

L'algorithme d'ordonnancement glouton est un algorithme de liste. Cet algorithme n'a aucune information sur les temps d'exécution des tâches. Dès qu'un processeur devient inactif, l'algorithme lui fournit une tâche de la liste. Pour implanter cet algorithme sur une plate-forme distribuée, un processeur M gère la liste de tâches. Lorsqu'un processeur devient inactif, il demande une tâche au processeur M. Ce dernier fournit les tâches aux autres. Le schéma de programmation associé est généralement appelé «Maître-travailleurs» [70].

Cet algorithme peut tenter de minimiser l'impact des communications sur le temps d'exécution car le maître a toutes les données pour le faire si les temps sont connus. Dans le cas où les temps ne sont pas connus, le maître en général attend qu'un processeur se libère pour lui donner du travail. Les données sont donc transférées au dernier moment.

Pour minimiser l'impact des communications, le maître peut décider de réaliser une pré-allocation des tâches. Sans connaître les temps d'exécution, certaines tâches devront être déplacées au cours de l'exécution pour équilibrer la charge. En revanche, la

pré-allocation permet de transférer les données dès leurs production lorsque les tâches ne sont pas déplacées.

Ordonnancement en-ligne avec pré-allocation

Pour réaliser la pré-allocation, l'algorithme a une estimation des temps d'exécution des tâches et de la quantité de données associé à chaque arête du «Directed Acyclic Graph» (DAG). En se basant sur ces estimations, un ordonnancement hors-ligne des tâches est calculé. La figure 3.1 illustre l'ordonnancement obtenu ainsi que le DAG de l'application. Au cours de l'exécution, la tâche E subit une perturbation. Ainsi son temps d'exécution est allongé. À la date $4,5$ où la tâche E devait finir, l'algorithme peut observer que la tâche ne finit pas à temps. Dans notre exemple, le temps d'exécution de la tâche E change du simple au double. L'algorithme a deux possibilités :

- Soit ne déplacer aucune tâche,
- Soit changer l'ensemble de l'ordonnancement en considérant le placement des données.

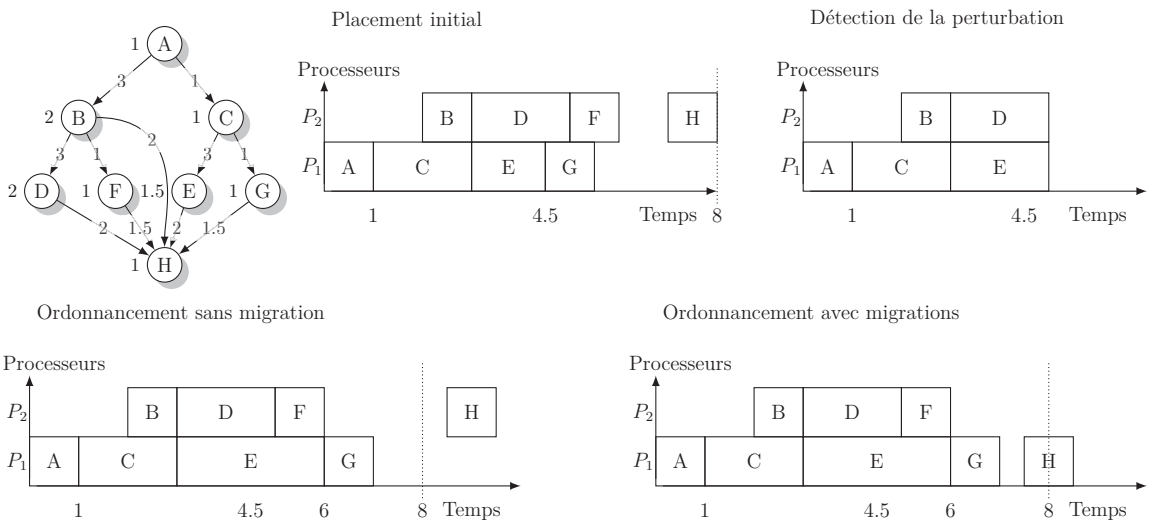


FIGURE 3.1 – Conséquence des perturbations sur l'ordonnancement

Nous considérons que l'algorithme choisit de migrer des tâches. Dans ce cas, l'algorithme d'ordonnancement a la possibilité de modifier le placement de l'ensemble des tâches restantes. Nous allons détailler quelques possibilités pour mettre en évidence l'impact des communications.

Dans le cas où les communications sont négligées, l'algorithme d'ordonnancement attendra qu'un processeur devienne inactif pour lui fournir la tâche la plus prioritaire. Puisqu'il n'y a pas de surcoût dû aux migrations, l'algorithme n'introduit pas de temps d'inactivité. De plus, l'ordre d'exécution qui minimise le temps d'exécution global, est conservé : les tâches les plus grandes (sans considérer les perturbations) sont exécutées en premier.

Lorsqu'il y a des communications, la décision de migration doit prendre en compte les transferts déjà réalisés. Les données déjà transmises seront alors transférées à nouveau.

Dans notre exemple, la décision de migration ne peut pas être réalisée avant la date $4,5$. Pour la tâche H, certaines données comme celle provenant des tâches B, D et F sont prêtes sur le processeur 2. Il faudra les transmettre si la tâche H est migrée.

Dans le cas d'une migration, il est nécessaire de transférer ces données. Elles peuvent être envoyées par le processeur les ayant envoyées initialement ou le processeur sur lequel la tâche était initialement ordonnancée. La retransmission de ces données risque d'entraîner des temps d'inactivités importants. D'autant plus que l'envoi de toutes ces données au même instant risque d'engendrer de la congestion réseau. Ainsi les temps de communications risquent d'être plus longs que prévu.

Après la migration de cette tâche, l'algorithme tentera d'en migrer d'autres. À chaque migration, le nombre de données à retransmettre augmente. Ainsi, tout l'ordonnancement est à revoir dès qu'une perturbation importante apparaît.

Le choix de migrer une ou plusieurs tâches nécessite de recalculer l'ordonnancement des tâches non-exécutées en considérant la retransmission de certaines données. De plus durant ce temps de calcul, l'exécution se poursuit. L'algorithme doit donc être rapide à exécuter et éviter d'introduire des temps d'inactivité durant le ré-ordonnancement.

Enfin, la perturbation n'est pas prévisible. En fonction de son ampleur il n'est pas possible de savoir à l'avance la bonne solution. Dans l'exemple, si la tache E subit une faible perturbation, il ne faut pas réaliser la migration.

Il existe des algorithmes utilisant des méthodes de ré-ordonnancement [43]. Dans notre approche, les temps d'exécution et de communication sont inconnus. Avec ces contraintes, nous pourrons fournir à l'algorithme de fausses informations pour calculer une pré-allocation robuste. En revanche, le DAG est généralement découvert à l'exécution. Dans ce cas, il n'est pas possible d'utiliser un algorithme utilisant une pré-allocation.

3.1.2 Partage de travail : «Load-Balancing»

Comme nous l'avons détaillé, l'algorithme d'ordonnancement en-ligne doit pour être générique s'abstraire des données fournies par l'utilisateur mais aussi du DAG de l'application qui peut évoluer d'une exécution à l'autre. Les ordonnancements par liste font partis des algorithmes d'ordonnancement les plus flexibles sur la prise de décision. Cette flexibilité provient de la possibilité de choisir n'importe quelle tâche prête tout en conservant la borne sur le temps d'exécution lorsqu'il n'y a pas de communications.

En revanche, lorsqu'il en y a, les choix de placement des tâches sont moins simples car chaque décision impacte la suite de l'ordonnancement. Dans les algorithmes de liste, l'objectif principal est de minimiser les temps d'inactivités. Nous présentons le partage de travail qui permet d'utiliser les algorithmes de liste en-ligne en considérant les communications. Cette méthode tente de minimiser les temps d'inactivité en anticipant sur la suite de l'exécution.

Lors du développement de cet algorithme d'équilibrage de la charge, le choix a été de considérer uniquement des tâches indépendantes. L'algorithme de partage de travail est appelé en anglais «Dynamic Load Balancing» [53].

Chaque processeur a un ensemble de tâches à exécuter. Ce travail est enregistré dans la mémoire du processeur qui doit exécuter la tâche. Le risque est qu'un processeur ait plus de travail que les autres. Pour rééquilibrer la charge entre les processeurs, chaque processeur surveille sa quantité de travail . Si un processeur est surchargé, des tâches sont migrées vers les autres processeurs. Ainsi, les tâches sont échangées entre les processeurs pour équilibrer la charge.

Le choix de migration nécessite une méthode pour mesurer la quantité de travail totale et locale à un processeur, mais aussi le coût de la redistribution du travail. Or, au fur et à mesure de l'exécution, la quantité de travail diminue. Il est donc nécessaire de faire transiter des messages pour avoir les informations. Lorsque celles-ci sont connues, l'algorithme doit migrer les tâches afin d'obtenir un système dans lequel aucun processeur ne détectera de surcharge. Dans l'étude [24], un algorithme distribué permettant l'obtention d'un système stable a été développé pour des tâches indépendantes et homogènes.

En considérant les tâches homogènes, l'algorithme peut calculer la quantité de travail sur toute la plate-forme et en local. En revanche, dans le cas général, cela implique des erreurs sur les temps d'exécution qui pourront être compensées par des migrations de tâches.

Malgré des améliorations possibles pour s'abstraire du temps d'exécution des tâches, la prise en compte des relations de précédences et des coûts de communications entre les tâches demande d'importants changements dans l'algorithme. Une possibilité pour s'abstraire des temps d'exécution et des coûts de communication, est de considérer les temps d'exécution égaux à un et les communications négligeables. Dans ce cas, les migrations de tâches risquent d'être fréquentes pour compenser le manque d'informations. Les performances obtenues risquent d'être mauvaises.

3.2 Vol de travail

L'objectif du vol de travail est de s'abstraire de toutes les données fournies par l'utilisateur tout en ordonnançant efficacement l'application sur un nombre de machines non défini. Ce mécanisme a été proposé initialement par Blumofe [8].

Le vol de travail est une méthode de parallélisation de plus en plus utilisée dans plusieurs bibliothèques [20, 31, 36, 67, 86]. Cet engouement pour ce mécanisme d'équilibrage de charge provient de ses nombreux atouts, en voici une liste non-exhaustive :

- algorithme distribué
- passage à l'échelle
- réactivité aux perturbations
- temps d'exécution analysé théoriquement
- nombre de tâches transférées analysé théoriquement.

L'implantation d'un tel mécanisme n'est pas unique, par exemple l'ordre des tâches exécutées peut différer. En fonction de cela, les performances de ce mécanisme peuvent varier.

Dans un premier temps, la section 3.2.1 décrit le fonctionnement général du vol de travail et les points impactant les performances. Puis dans le cas d'utilisations classiques, la section 3.2.2 détaille une analyse théorique du vol de travail. Enfin, le fonctionnement de bibliothèques fournissant un mécanisme de vol de travail est présenté dans la section 3.2.3.

3.2.1 Ordonnancement par liste distribuée

Une des principales propriétés du vol de travail est que l'algorithme est distribué. Dans cet algorithme, les décisions sont généralement prises avec les informations locales à chaque processeur participant à l'exécution. Nous avons donc choisi de détailler l'algorithme en utilisant une vue restreinte à un processeur.

Durant l'exécution, chaque processeur a une liste de tâches qui représente le travail à effectuer. Lors de l'exécution d'une tâche, le processeur exécute les instructions associées qui peuvent générer d'autres tâches. Ces tâches sont stockées pour la suite de l'exécution dans sa liste. L'ensemble des tâches et leurs données décrivent l'application qui est représentée généralement par un DAG comme la section 1.3.1 (page : 19) le décrit.

En fonction de la quantité de travail présente dans la liste, deux états sont distinguables pour les processeurs :

Un travailleur : un processeur qui exécute du travail (une tâche). Sa liste de tâches peux être vide ou non.

Un voleur : un processeur qui n'a plus de travail à exécuter. Sa liste de tâches est vide. Il tente de trouver du travail auprès des autres processeurs.

Tous les processeurs évoluent entre ces deux états. Le passage de voleur à travailleur s'opère en demandant du travail aux autres processeurs. Si un processeur reçoit une demande de travail et s'il a des tâches dans sa pile, il envoie du travail au voleur. Le passage de travailleur à voleur s'effectue en fonction des requêtes de vol et de l'exécution des tâches.

Dans le mécanisme de vol de travail, l'ordre d'exécution des tâches est influencé par plusieurs décisions : la prochaine tâche à exécuter, le processeur volé et la ou les tâches volées. Nous détaillons l'influence de ces décisions sur les performances.

a Ordre d'exécution local

L'ordre d'exécution est un des choix qui est le plus rarement modifié, car l'analyse théorique initialement réalisé par Blumofe et Leiserson [9], montre que la décision de suivre le parcours initial du programme sans création de tâches (l'ordre séquentiel) est efficace. Le respect de l'ordre séquentiel permet de conserver les optimisations réalisées par le programmeur. De plus, ce choix permet de borner l'espace mémoire utilisé par chaque processeur. Cet espace mémoire est dans ce cas inférieur ou égal à l'espace mémoire utilisé par le programme séquentiel. Cet ordre d'exécution est équivalent à un parcours en profondeur du DAG de l'application. Au niveau de la liste, ce parcours peut être réalisé en sélectionnant la dernière tâche ajoutée («Last In First Out (LIFO)»).

Puisque la liste est souvent utilisée avec cet ordre («LIFO»), elle est généralement appelée pile. Par la suite, nous utiliserons le terme de pile qui est le plus largement employé dans la littérature.

L'ordre opposé est le parcours en largeur. Il est rarement utilisé dans le mécanisme de vol de travail. Ce parcours génère un grand nombre de tâches qui pourront être volées. Cela peut permettre de fournir du travail à un grand nombre de machines. En contrepartie, il utilise plus de mémoire et ne permet pas de conserver les optimisations du programme séquentiel (principalement les accès mémoire).

Ces deux ordres d'exécution pourraient être mixés pour tirer parti de l'avantage des deux stratégies.

La pile permet aussi les interactions entre les voleurs et un travailleur. Pour minimiser le temps d'exécution, le voleur doit pouvoir obtenir du travail avec le minimum de surcoût possible. Nous détaillerons ce problème d'accès concurrent à la pile au moment du choix de la ou les tâches volées.

b Envoi d'une requête de vol

Lorsqu'un processeur devient un voleur, il choisit le processeur victime dont il va prendre une partie du travail. Il existe un grand nombre de stratégies différentes. Une des manières de sélectionner simplement la victime est de réaliser un tirage uniforme parmi les processeurs participant à l'exécution. Les analyses théoriques montrent qu'avec une telle sélection de la victime, la probabilité de sélectionner un processeur ayant une quantité importante de travail est grande même avec un nombre limité de vols [2] (dont une version journal est [3]).

D'autres choix ont été analysés, comme celui du processeur ayant la tâche la moins profonde dans sa pile. Cette solution minimise le temps d'exécution, mais elle nécessite de conserver une information du processeur possédant cette tâche sur toute la plate-forme [68].

Pour prendre en compte l'influence des accès mémoires sur des plates-formes hiérarchiques ou non, des travaux ont montré qu'il est intéressant d'orienter les vols vers des tâches ayant des données qui pourront être accédées rapidement par le voleur [1].

Le choix aléatoire reste le plus simple, ne nécessitant aucune information avec un nombre de requêtes de vol borné.

c Réponse à une requête de vol

Sur réception d'une requête de vol, un processeur doit sélectionner le travail à fournir. L'algorithme réalisant la sélection du travail à transférer vers le voleur s'exécute en concurrence du travail effectué par le processeur volé. Classiquement dans le vol de travail, la dernière tâche ajoutée à la pile est la tâche sélectionnée pour être exécutée. Ainsi, les processeurs accèdent à leur pile par le bas (ordre «LIFO»). Pour ne pas perturber le travail de la victime, les tâches volées sont choisies prioritairement en haut de la pile (ordre «First In First Out (FIFO)»). Ce mécanisme est illustré figure 3.2.

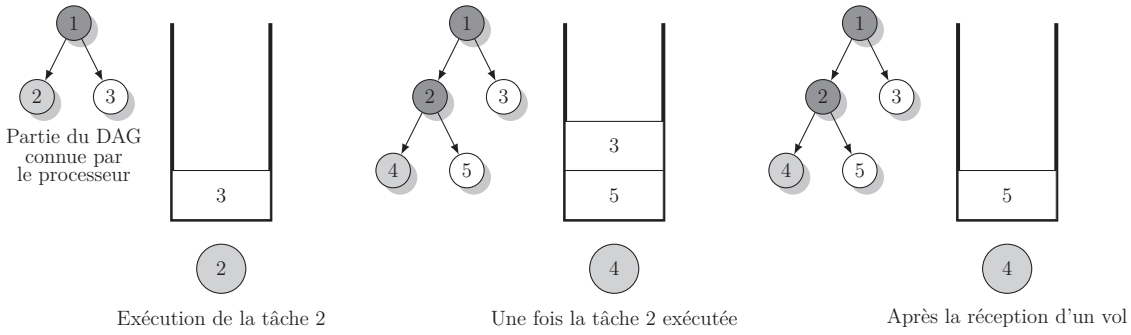


FIGURE 3.2 – Accès à la pile de tâches

Les conflits d'accès risquent d'apparaître uniquement quand le nombre de tâches est faible. Il existe de nombreux travaux permettant la réduction des perturbations lors de l'accès aux tâches, en voici quelques-uns adaptés au vol de travail «T.H.E.» [3], «Non-blocking» [42] et «wait-free» [73], «lock-free» [84].

Il reste à définir le nombre de tâches prises dans la pile. Lors d'un vol, l'objectif est d'équilibrer le travail entre les deux processeurs (la victime et le voleur). Puisque le voleur n'a pas de travail, la quantité de travail prise doit être proche de la moitié de celle présente sur la victime. Cette quantité est en pratique efficace comme les études [27, 34] le montrent. Dans les applications classiques pour le vol de travail, le modèle de programmation est souvent récursif découpant le travail en deux à chaque niveau de récursions. Ainsi, la tâche la plus ancienne dans la pile représente une grande quantité de travail.

Les analyses théoriques comme [3, 9, 10] dont une est détaillée dans la section 3.2.2 reposent sur un parcours en profondeur localement à chaque processeur, un choix aléatoire de la victime et le vol de la tâche la plus ancienne. Ces choix sont ceux classiquement utilisés dans les différentes bibliothèques. Les alternatives sont rarement analysées théoriquement.

Nous venons de présenter les différents points qui peuvent modifier l'ordre d'exécution des tâches pour les plates-formes homogènes. Dans les implantations et les analyses théoriques suivantes, nous considérons ce modèle de machines. L'algorithme classique de vol de travail est détaillé en pseudo-code dans l'annexe A.3 (page : 158).

Le vol de travail a été adapté aux plates-formes hétérogènes par Bender et Rabin [6]. La modification est réalisée au niveau de la tâche volée. Précédemment sur plates-formes homogènes, si la pile de la victime est vide, aucun travail n'est envoyé au voleur. Dans la version modifiée pour plates-formes hétérogènes, lorsque la victime n'a pas de travail dans sa pile et qu'elle exécute du travail, les puissances de calcul du voleur et de la victime sont comparées. Si le voleur calcule plus rapidement que la victime, le travail en cours est migré vers le processeur voleur (plus rapide).

Cette décision permet de profiter de la puissance de calcul des processeurs les plus rapides. Pour utiliser cet algorithme, la puissance des machines doit être connue. Cette information n'est pas toujours simple à obtenir. De plus, le nombre de migrations de tâches est plus important dans cette solution.

3.2.2 Analyse théorique du vol de travail sur plate-forme homogène

L'analyse théorique d'algorithmes d'ordonnancement en-ligne modélise généralement l'évolution de l'ordonnancement au cours du temps. Dans le cas du vol de travail, les processeurs sont dans deux états possible, travailleur ou voleur. Le temps passé à travailler est déterminé par la quantité de travail présente dans l'application. L'objectif des analyses sera de borner le temps passé à voler par chacun des processeurs. Pour obtenir la borne, les analyses calculent classiquement l'avancement de l'exécution en fonction du nombre de vols émis. Quelque soit l'analyse, elles bornent le temps d'exécution et le nombre de tentatives de vol (réussies et/ou échouées). Les bornes classiques [3, 81] sont définies en fonction du nombre de processeurs (p), de la quantité de travail (W) au sein de l'application et du chemin critique de l'application (D) :

- Temps parallèle sur p processeurs : $T_p \leq \frac{W}{p} + O(D) + \log(\frac{1}{\epsilon})$ avec une probabilité supérieure à $1 - \epsilon$,
- Nombres de vols (réussis et échoués) : $\#S \leq O(p \times D)$.

Parmi les analyses théoriques existantes, certaines sont plus ou moins précises et simples. Nous détaillons l'analyse avec les fonctions Strictly Increasing Per Steal (SIPS) [68] qui est à notre avis, la plus intuitive et adaptable à d'autres stratégies de vol.

Nous commençons par détailler les hypothèses classiques utilisées pour analyser le vol de travail. Le premier point est sur la représentation de l'application utilisée : le DAG est constitué de tâches unitaires et d'une arité égale à 2. La restriction aux tâches unitaires n'est pas limitant car une tâche de taille 3 peut être remplacée par trois tâches chaînées. La restriction de l'arité du DAG est uniquement théorique. Pour un DAG d'arité quelconque, il est possible de le transformer en un DAG d'arité 2 en ajoutant des nœuds comme la figure 3.3 l'illustre. Les autres hypothèses sont liées à la plate-forme

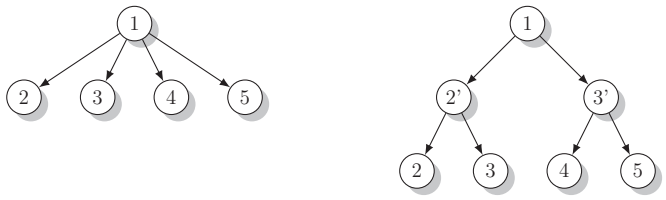


FIGURE 3.3 – Passage d'un DAG d'arité quelconque à un DAG d'arité 2

d'exécution :

- Temps de vol unitaire,
- Perturbation lors de l'extraction du travail négligeable,
- Processeurs homogènes.

Le théorème 3.1 détaille l'analyse théorique réalisée avec les fonctions SIPS pour un mécanisme de vol. Elle peut être étendue à un mécanisme de vol de travail classique. Avant de réaliser la démonstration, nous définissons formellement les fonctions SIPS.

Définition. *Fonctions SIPS*

Une fonction SIPS est une fonction de $\mathbb{N}$ *dans* $\mathbb{N}^p$ *avec* p *le nombre de processeurs caractérisant l'état global de la machine en fonction du temps. Elle se définit telle que* $\forall\ n\ \in\ \mathbb{N}, f(n) = [f_1(n), \ldots, f_p(n)]$.

De manière générale :
- *Si le processeur* i *est inactif à la date* n, $f_i(n) = 0$,
- *Si le processeur* i *est actif à la date* n, $f_i(n) > 0$.

Les fonctions f_i *évoluent au cours de l'exécution avec les conditions suivantes :*
- *Si le processeur* i *est actif et qu'il reçoit une requête de vol à l'instant* n *:* $f_i(n+1) \geq f_i(n) + 1$,
- *Dans le cas où le processeur* i *envoie une requête de vol vers le processeur* j *à la date* n *:*
  - *Si il n'y a pas de tâche reçue, le processeur* i *reste inactif donc :* $f_i(n+1) = 0$,
  - *Si une tâche est reçue, le processeur* i *devient actif et :* $f_i(n+1) \geq f_j(n) + 1$.

Théorème 3.1. *Mécanisme de vol avec une connaissance globale [68]*

Soit une application dont la quantité de travail totale est W *et le chemin critique est* D*, ordonnancée par vol de travail sur une plate-forme homogène de* p *processeurs. Pour un mécanisme de vol, qui exécute localement en profondeur d'abord et qui vole une des tâches les moins profondes présentes sur l'ensemble de la plate-forme, l'application sera exécutée en un temps inférieur à* $\frac{W}{p} + D$ *et avec un nombre de tentatives de vol inférieur à* $p \times D$.

Démonstration. Nous utilisons une fonction SIPS particulière, telle que, pour chaque processeur i, $f_i(n)$ est égale à la profondeur de la tâche la moins profonde de sa pile. En choisissant cette profondeur comme fonction, chaque f_i est bornée par D. La démonstration peut être réalisée pour une fonction SIPS quelconque tant que chaque f_i est bornée par une valeur finie.

Nous considérons maintenant la fonction $F(n)$ définie par :
- $\mathbb{N} \to \mathbb{N}$
- $F(n) = \min_{i \in [1..p] \& f_i(n) > 0} (f_i(n))$

La fonction F renvoie la valeur minimale non nulle pour l'ensemble des processeurs.

Nous analysons maintenant l'évolution de $F(n)$ en fonction des vols. Considérons le cas où il y a p processeurs avec la valeur minimale $F(n)$. Soit la date t où un des processeurs devient inactif, celui-ci envoie une requête de vol vers un processeur qui a la valeur $F(n)$ (induit par la politique de vol). À $t+1$ les deux processeurs auront une valeur supérieure ou égale à $F(n) + 1$.

Dans le cas où un des processeurs procédant une fonction f_i égale à la valeur de F est volé à chaque fois, il faut au plus $p - 1$ requêtes de vol pour faire augmenter F de 1.

Enfin, la fonction F est elle aussi, bornée par la même valeur que les fonctions f_i, dans notre cas D. Lorsque F atteint cette borne, l'exécution est terminée.

Il y a donc au plus $(p-1) \times D$ requêtes de vol si le processeur choisit sa victime parmi les processeurs ayant la plus petite valeur.

Ainsi, nous avons montré que le nombre de requêtes de vols envoyées est borné.

Pour terminer la preuve, nous considérons qu'à chaque instant, un processeur est en train soit de travailler soit de voler. La somme des temps passés à voler est bornée par $(p-1) \times D$ et la somme des temps passés à travailler est bornée par W.

Le temps parallèle est donc inférieur à $\frac{W}{p} + \frac{p-1}{p}D$ si chaque vol tente de voler la tâche la moins profonde.

□

Un extension simple de cette preuve est de considérer qu'en réalisant un certain nombre de vol aléatoire, les processeurs qui ont les tâches de profondeur minimale seront volée. Ce problème est équivalent au problème de «coupon collector» [72]. Dans ce cas, il faut $O(p \times log(p))$ requêtes pour envoyer au moins une requête à chacun des processeurs avec une forte probabilité. Le temps d'exécution est borné par $\frac{W}{p} + O(p \log(p) D)$

Cela ne permet pas d'obtenir la borne la plus fine. Il est possible d'obtenir la même borne que le vol de travail classique avec l'utilisation d'une fonction potentielle que nous ne détaillerons pas ici [68].

3.2.3 Implantations

Les implantations du vol de travail sont nombreuses, il y a par exemple *Cilk* [31], *X10 Work Stealing (XWS)* [20, 48], *Kaapi* [36], *Threading Building Blocks (TBB)* [67], *Satin* [86]. Nous n'allons pas toutes les détailler. Pour commencer, nous analysons le langage utilisé par *Cilk* pour mettre en évidence ses limitations et ses avantages. Son implantation a été détaillée par Frigo et al. [31]. Puis, nous détaillerons les points importants sur la gestion des transferts de données de l'implantation de *Kaapi* que nous avons utilisée comme base pour le développement et l'analyse de nos algorithmes hiérarchiques.

Cilk

Le langage utilisé par *Cilk* (maintenant appelé *Cilk+* par Intel) qui fut le premier moteur efficace de vol de travail, est restreint à deux mots clefs :

- «Spawn» permet de désigner un ensemble d'instructions qui peuvent être exécutées indépendamment des autres instructions.
- «Sync» permet d'exécuter les instructions qui suivent dès que les parties indépendantes préalablement désignées avec un «Spawn» sont terminées.

Avec ce langage, le DAG formé par les nœuds «Spawn» et «Sync» a une structure de type «Fork-Join» introduite en section 1.3.1. Il n'est pas possible de décrire des schémas de programmation plus complexes.

Cela a pour avantage de contraindre le programmeur à développer des codes récursifs. Dans de tels programmes, le principal schéma de programmation utilisé est de type : «diviser pour régner». Cette programmation tend vers une découpe du travail en parties plus petites pour résoudre le problème plus facilement. Les parties générées durant une étape de récursion contiennent généralement une quantité de travail similaire. Le premier processeur exécute les tâches en profondeur. Lors d'un vol, la tâche volée qui

est la plus ancienne de la pile, contient une grande quantité de travail. Le transfert d'une grande quantité de travail lors des vols permet de répartir le travail entre les processeurs efficacement.

De plus dans l'implantation de *Cilk*, le travail effectué après le «Sync» est réalisé par le processeur qui termine la dernière tâche à synchroniser. Cette connaissance du processeur qui exécutera le travail peut permettre de transférer les données dès que le processeur est connu.

Kaapi

Dans la bibliothèque *Kaapi* [37] développée au sein de l'équipe de recherche Moais, le langage utilisé est plus complet. Il est composé principalement du mot clef «Fork» qui permet de créer une tâche avec une liste d'arguments. Pour chaque argument, l'utilisateur désigne s'il est lu et/ou écrit. Avec ces informations, la bibliothèque peut calculer à l'exécution le graphe de flot de données. Ce graphe peut être représenté par un DAG avec le poids sur les arêtes. Les dépendances entre les tâches définissent plus finement les relations de précédences entre les tâches. Ainsi, cette description permet d'exploiter au maximum le parallélisme de l'application.

Dans la suite, nous réaliserons des expériences comparant plusieurs algorithmes de sélection du processeur victime implantés dans la bibliothèque *Kaapi*. Nous détaillons donc deux points importants de l'implantation du vol de travail.

Création des tâches : Dans *Kaapi*, l'exécution est organisée par tâche. Lors de l'exécution d'une tâche i, une nouvelle pile S est créée. Les tâches créées (prêtes ou non) sont ajoutées à la pile associée à la tâche i. Les dépendances sont gérées au sein de cette pile. Dès que l'ensemble des tâches de la pile S ont été exécutées, la tâche i est considérée comme étant terminée. Le ou les résultats générés sont mis à jour dans la pile où la tâche i a été créée. La ou les tâches nécessitant ces résultats dans la pile sont mises à jour. La figure 3.4 illustre le fonctionnement de la gestion des dépendances.

La création des tâches est différente de celle réalisée dans les bibliothèques classiques comme *Cilk* et *TBB*. Généralement, les tâches sont ajoutées dans la pile du processeur qui les active.

Cette différence influence l'ordre dans lequel les tâches sont volées. Les tâches prêtes sont volées en fonction de leur ordre de création et non de leur profondeur. La figure 3.5 illustre un DAG avec des arêtes signalant la création des tâches.

En contrepartie de cette différence, la gestion des tâches permet de répartir les informations concernant celles-ci entre les processeurs. Chaque processeur conserve uniquement l'état (créée, prête, volée, terminée) des tâches qu'il a créées. Ainsi, l'exécution peut être facilement réalisée sur une plate-forme distribuée.

Transfert des données : Sur une plate-forme distribuée, la gestion des transferts est un point important pour minimiser le temps d'exécution global. Pour éviter les

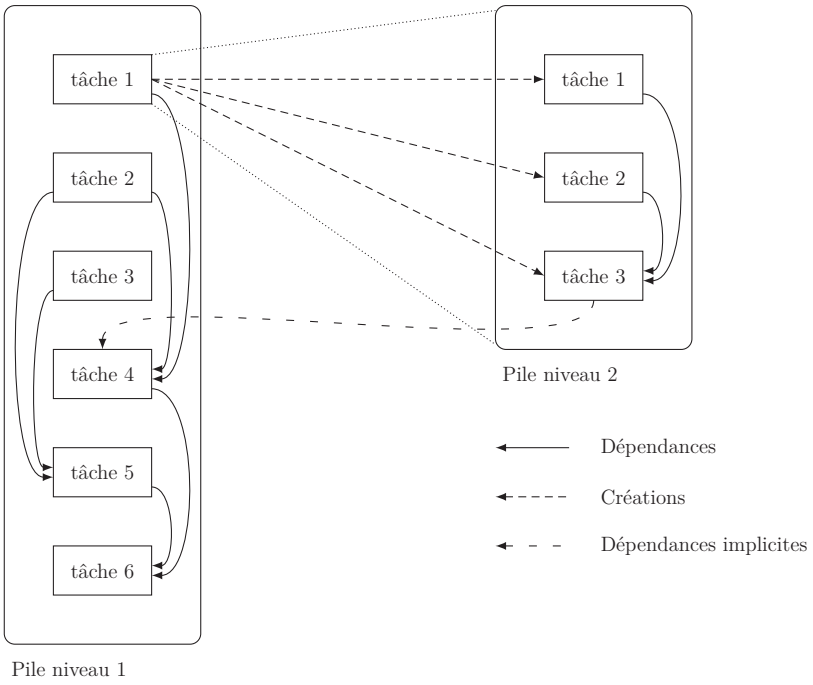


FIGURE 3.4 – Gestion des dépendances au sein de *Kaapi*

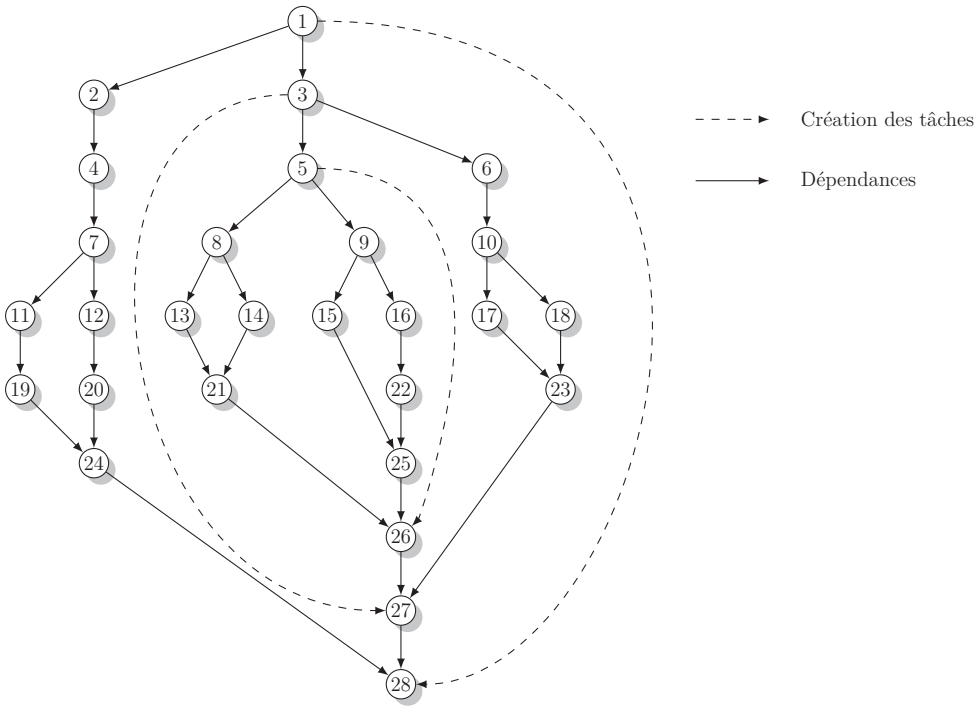


FIGURE 3.5 – Représentation d'un DAG avec les arêtes de créations

sur-coûts, l'intergiciel réalise les transferts au travers d'une couche réseaux utilisant directement les «sockets» [76].

Quant à la gestion des transferts dans *Kaapi*, elle est directement induite par la gestion des tâches décrites précédemment. La gestion est réalisée par tâche ce qui permet de limiter les communications aux processeurs voleur et volé. Lors d'un vol, une tâche prête est choisie. Les données associées sont transférées directement au moment du vol. Cela évite d'attendre les transferts par la suite, au moment des accès aux données durant l'exécution de la tâche. Dès que la tâche volée est terminée, l'ensemble des résultats est renvoyé au processeur l'ayant créée. Dans ce mécanisme, les processeurs qui ne participent pas au vol, ne sont pas affectés par les transferts de données, évitant ainsi des perturbations inutiles.

En contrepartie, certaines données peuvent être transférées plusieurs fois. Par exemple pour l'application de la figure 3.4, le processeur initial créé la la pile de niveau 1. Le processeur i vole la tâche 3, les résultats de celle-ci sont stockés chez lui. À la fin de l'exécution, il renvoie les données au processeur initial. Le processeur i vole maintenant la tâche 5. L'ensemble des résultats de la tâche 2 et de la tâche 3 lui sont renvoyés alors que les résultats de la tâche 3 ont été calculé par lui même.

Nous verrons par la suite comment ces choix ont influencé notre gestion des requêtes de vol pour la prise en compte de plates-formes ayant une topologie de réseau complexe.

3.3 Vol de travail adapté à une plate-forme hiérarchique

Lors de la conception du mécanisme de vol de travail, les communications autres que les vols ont été négligées. Le principal objectif était d'obtenir un équilibrage de la charge au cours de l'exécution qui soit efficace. L'efficacité est obtenue en conservant les optimisations réalisées par le programmeur (ordre d'exécution des tâches) et en évitant au maximum les perturbations sur les processeurs exécutant du travail (accès à la pile de tâches, vol de la tâche la plus ancienne).

Les nouvelles architectures sont de plus en plus complexes avec des coûts d'accès à la mémoire qui dépendent de la zone accédée et du processeur, par exemple sur des architectures Non-Uniform Memory Access (NUMA) , grappe de calculs, grille de calculs. Le vol de travail a été adapté afin de privilégier les accès à la mémoire la plus proche dans le cadre de mémoire partagée. Il existe plusieurs algorithmes qui ont été développés de manière différente en fonction du niveau de hiérarchie considérée (cache, mémoire vive, *InfiniBand*, Ethernet).

La section 3.3.1 introduit certains mécanismes optimisant les accès aux caches et à la mémoire vive. Lors du passage aux plates-formes distribuées, les surcoûts dus aux communications sont plus importants. Les transferts de données et de communications doivent être évités au maximum. La section 3.3.2 présente certaines méthodes utilisées sur les plates-formes distribuées.

3.3.1 Plate-forme à mémoire partagée

Les premières implantations du vol de travail ont été réalisées sur les plates-formes à mémoire partagée. En revanche, ces plates-formes se sont largement complexifiées avec l'arrivée des processeurs multi-cœurs et NUMA.

Nous détaillons dans un premier temps la méthode introduite par Blelloch et al. [1] qui demande à l'utilisateur de spécifier l'affinité entre les tâches. Chaque processeur a une boîte aux lettres dans laquelle les tâches qui ont une affinité avec les tâches précédemment exécutées sont postées. Lors de la création d'une tâche, une copie de la tâche est placée dans la boîte aux lettres du processeur ayant le plus d'affinités. Lors qu'un processeur devient inactif, il vérifie si une tâche est présente dans sa boîte avant de réaliser le vol d'une tâche classiquement. Lorsque l'une des deux copies est prise par un processeur, l'autre copie est invalidée.

Le temps d'exécution de cette heuristique ne peut pas être borné avec une analyse identique au vol de travail car l'ordre de vol des tâches est complètement modifié : le vol d'une tâche dans la boîte aux lettres introduit des vols au milieu des piles de tâches (au lieu du haut de la pile). L'obtention de cette information s'effectue souvent en demandant à l'utilisateur de spécifier les affinités entre les tâches. En pratique, cette méthode se montre efficace, en revanche la description des affinités demande un travail important à l'utilisateur.

Nous détaillons maintenant plusieurs algorithmes permettant de limiter les coût d'accès à la mémoire sans information supplémentaire de la part de l'utilisateur.

Amélioration automatique de la localité des données

Sur les nouvelles architectures basées sur des processeurs multi-cœurs, le vol de travail ne prend pas en compte les aspects de localité des données. Des algorithmes d'ordonnancement de tâches comme Parallel Depth First (PDF) [7, 17] et le vol de travail à fenêtre [80] ont été développés pour améliorer la réutilisation des données.

Ces algorithmes tentent de paralléliser l'exécution sur un processeur (multi-cœurs) en conservant un ordre d'exécution proche de l'exécution séquentielle. Ils s'appuient sur l'hypothèse que l'ordre d'exécution séquentiel minimise le nombre d'accès à la mémoire centrale.

L'algorithme PDF utilise une liste globale. Dans cette liste les tâches prêtes sont triées par rapport à l'ordre séquentiel. Ainsi, il permet d'améliorer la localité des données. L'analyse théorique [80] détaille le gain obtenu avec cet ordre d'exécution. En revanche, PDF augmente la contention et le nombre de synchronisations par l'utilisation d'une liste globale.

Quant à l'algorithme à fenêtre, il travaille sur un ensemble de tâches indépendantes présentes dans un tableau. L'algorithme séquentiel traite chaque élément du tableau dans l'ordre. Pour suivre l'algorithme séquentiel, le vol de travail contraint l'exécution des tâches. Lors d'un vol le processeur peut voler un ensemble de tâches présentes dans une fenêtre comme le montre la figure 3.6. Cette fenêtre est déterminée par la tâche de plus petit indice qui n'est pas encore exécutée et un nombre d'éléments. Chaque

processeur possède sa propre pile de tâches. La contention est donc plus faible que dans PDF. En revanche, il est limité à un ensemble de tâches indépendantes. De plus, cet algorithme nécessite des synchronisations entre les processeurs.

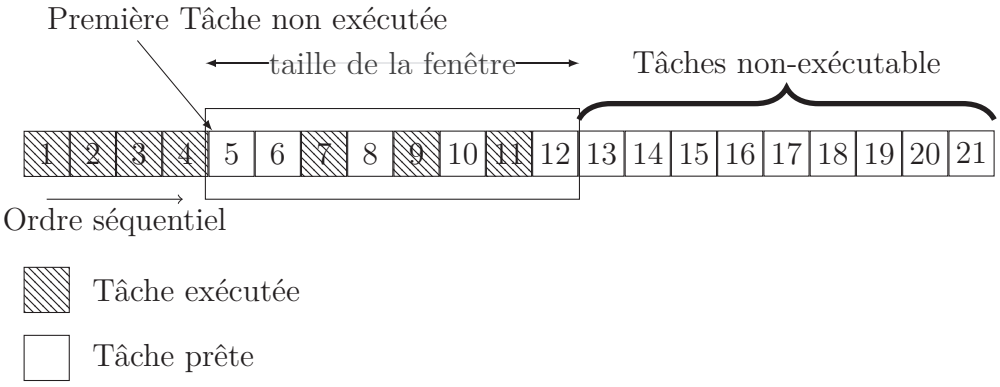


FIGURE 3.6 – Exécution d'une ensemble de tâches indépendantes avec l'algorithme de vol de travail à fenêtre.

Ces deux algorithmes ont été développés dans le contexte des processeurs multi-cœurs où ils n'exploitent qu'un seul niveau de hiérarchie. Leur généralisation à une plate-forme distribuée avec plusieurs niveaux de hiérarchie n'a pas été encore étudiée. De plus, les méthodes utilisées en mémoire partagée ne fournissent pas les mêmes performances en mémoire distribuée car elles génèrent généralement trop de communications ou de synchronisations.

3.3.2 Plate-forme à mémoire distribuée

Sur les plates-formes à mémoire distribuée, les communications entre les processeurs utilisent le réseau d'interconnexion (*InfiniBand*, Ethernet). Le temps de telles communications est largement plus important que celui nécessaire pour faire communiquer deux processeurs de la même machine. De plus, le débit de ces liens est généralement bien plus faible. Ces méthodes de communications ne sont pas en adéquation avec les hypothèses émises pour l'analyse théorique du vol de travail. Pour limiter l'influence des coûts de communications, les choix du processeur volé et de la ou des tâches volées doivent prendre en compte ces coûts de communications. De manière identique, les coûts de communications évoluent lors du passage d'une grappe de calculs à une grille de calculs qui est constituée de plusieurs grappes de calculs.

Nous présentons dans un premier temps un algorithme développé pour les grappes de calculs, puis certains algorithmes développés pour les grilles.

a Grappe de calculs

Sur ce type de plate-forme, la distinction est réalisée entre les processeurs d'une même machine et ceux des autres machines de la grappe. L'algorithme est intuitif :

avant de chercher du travail sur les autres machines, le processeur tente de chercher du travail sur les processeurs de la même machine avant d'en voler un autre sur les autres machines de la plate-forme.

La première implantation a été réalisée dans la librairie *Kaapi*. L'objectif est de voler du travail ayant des données sur la même machine en priorité.

Cette politique de vol est juste une idée qui a été développée dans *Kaapi*. Il y a plusieurs points qui sont sujets à discussion, comme par exemple, le nombre de requêtes de vols en local avant de voler un processeur distant et l'impact sur les temps d'inactivité.

Pour la recherche de travail sur une plate-forme, une étude [39] basée sur des simulations parfaites par chaînes de markov a montré qu'en testant un faible nombre de machines pour obtenir du travail, les performances sont proches de celles obtenues en testant l'ensemble de la plate-forme, même si celle-ci comporte un grand nombre de machines. Dans le cas de *Kaapi*, le nombre de processeurs sur une même machine est souvent relativement faible sur les plates-formes utilisées (environ 8-cœurs). Le choix de réaliser un seul vol avant de voler une autre machine est donc justifiable. De plus, l'augmentation du nombre de vols locaux a tendance à réduire largement le travail présent sur la machine avant de faire un vol distant. Le risque est d'avoir tous les processeurs d'une machine inactive au même instant et de gaspiller du temps d'exécution.

b Grille de calculs

Les grilles de calculs sont un regroupement de grappes de calculs. Les communications intra-grappes sont généralement plus performantes que les communications inter-grappes, surtout en considérant la performance par rapport au nombre de machines partageant les liens.

Sur ce type de plates-formes, les expériences réalisées par l'équipe Ibis avec la bibliothèque *Satin* montrent que le temps d'exécution avec une politique de vol aléatoire peut être amélioré en pratique [25]. Les trois heuristiques [85] proposées pour améliorer les performances sont :

Cluster-aware Hierarchical Stealing (CHS) : Chaque processeur est considéré comme étant un nœud d'un arbre avec un processeur père et deux processeurs fils. Cette structure est utilisée au moment des vols. Lorsqu'un processeur devient inactif, il commence par envoyer une requête de vol à un de ses fils. Tant qu'aucune tâche n'est trouvée, la requête est transmise de père en fils pour rechercher du travail dans tout le sous arbre. Si il n'y a pas de travail dans le sous arbre, la requête est envoyée au père du voleur initial. Le but de cette heuristique est de favoriser le travail local et de limiter les vols distants.

Cluster-aware Load-based Stealing (CLS) : La plate-forme d'exécution est découpée en grappes de machines. Les requêtes de vols sont restreintes à la grappe associée au voleur. Dans chaque grappe, un maître est désigné pour communiquer avec les machines externes. Chaque machine fournit au maître des informations sur la quantité de travail restant localement. Avec ces informations, le maître prend

la décision de réaliser un vol vers une machine quelconque d'une autre grappe (distante). Le but de cette stratégie est de diminuer l'utilisation réseau entre les grappes en limitant le nombre de machines utilisant ces liens de communications.

Cluster-aware Random Stealing (CRS) : Les machines sont organisées en grappes. Chaque machine a, à sa disposition deux types de vols : asynchrones et synchrones. Chaque machine peut envoyer un seul vol asynchrone à un instant donné. Les vols asynchrones sont restreints aux vols des machines externes. Au contraire, les vols synchrones sont limités aux vols intra-grappes. Classiquement, un processeur devient voleur dans le cas où il n'a plus de travail localement. Cette heuristique favorise le travail local et le recouvrement des communications distantes avec du travail et/ou des communications locales.

Ces heuristiques ont été utilisées sur de nombreux exemples tels que Fibonacci, N-Reines ou encore le produit de matrices. Le gain de performance avec ces heuristiques est visible sur de grandes plates-formes hiérarchiques.

Ces heuristiques permettent d'améliorer les performances en comparaison à la politique aléatoire. Mais chacune de ces heuristiques n'est pas sans défauts. Le principal défaut est l'absence d'analyse théorique permettant de borner le temps d'exécution global et le nombre de vols.

En reprenant les algorithmes un par un, nous relevons quelques inconvénients :

CHS : cette heuristique utilise un arbre représentant la plate-forme pour diriger les vols en priorité vers les machines locales. L'algorithme est si restrictif qu'un vol entre deux grappes a lieu uniquement si il n'existe aucune tâche à exécuter dans la grappe. Ainsi, lors d'un vol distant, le nombre de processeurs inactifs peut être assez important dans la grappe du processeur qui envoie la requête.

CLS : Les maîtres gèrent la quantité de travail présente dans leur grappe. La mesure de la quantité de travail est réalisée en fonction du nombre de tâches présentes sur chacun des processeurs de la grappe. Cette mesure est faussée car dans un schéma de type «diviser pour régner», les tâches avec une profondeur faible représentent une quantité de travail bien plus importante que les tâches avec une profondeur forte. De plus, le choix du moment où le maître envoie une requête de vol, est réalisé à partir de nombreux paramètres dépendant de l'application. Enfin, lors du vol d'une tâche d'un maître, il est probable que la tâche volée représente peu de travail.

CRS : Les processeurs ont à leur disposition deux types de vols. Lorsqu'un processeur devient inactif, il envoie deux requêtes de vols (si il n'y a pas de requête asynchrone en attente de réponse). La requête synchrone est généralement plus rapide. Si le processeur reçoit une tâche avec la requête synchrone, le processeur l'exécute. Il est possible que durant l'exécution de cette tâche, il reçoive du travail en réponse de la requête asynchrone. Cette tâche est ajoutée à sa pile. Elle peut être volée une nouvelle fois par un vol asynchrone et ainsi migrer d'une grappe à une autre plusieurs fois.

Malgré ces désavantages, ces algorithmes permettent d'obtenir dans certaines conditions de meilleures performances en pratique, que la politique de vol aléatoire. Parmi ces trois heuristiques, CRS est l'heuristique qui obtient les meilleures performances [85].

De plus, cet algorithme est assez simple à implanter et ne nécessite pas le réglage de paramètres.

II

Vol de travail et communications

Introduction

Dans les chapitres précédents, nous avons détaillé différents algorithmes d'ordonnancement hors-ligne et en-ligne. Les algorithmes hors-ligne ont un net avantage sur la gestion des communications :

- En permettant le recouvrement des communications par l'exécution de tâches. Ces algorithmes ont été présentés dans la section 2.1.
- En permettant le regroupement des tâches partageant des données volumineuses. Cette méthode a été introduite dans la section 2.2.

Cependant, les ordonnancements hors-ligne obtiennent cet avantage grâce à la connaissance des temps d'exécution des tâches et des communications. Or, ces temps sont souvent soit inconnus soit inexacts. Par exemple si une machine est exploitée par plusieurs utilisateurs, les temps d'exécution varient en fonction du travail en cours. Pour ordonnancer les tâches sans connaître leur temps d'exécution, nous nous intéressons aux algorithmes en-ligne. Parmi ces ordonnancements, le vol de travail présente de nombreux atouts.

D'un autre coté, la performance des algorithmes en-ligne est généralement impactée par les communications lorsque celles-ci sont conséquentes. Les communications impactent le temps d'exécution global pour de multiples raisons. Par exemple lors des vols, la tâche la plus ancienne est volée sans savoir si le vol de celle-ci va générer de nombreux transferts ou non. De plus, la gestion des transferts est souvent négligée :

- *Cilk* laisse le système d'exploitation gérer les transferts automatiquement,
- KAAPI restreint la description des dépendances et peut transférer des données inutilement dans certains cas.

Nous souhaitons améliorer la gestion des transferts de données du mécanisme de vol de travail.

Le chapitre 4 présente un algorithme Work-Stealing with COMmunication (WSCOM) basé sur le mécanisme de vol de travail. Dans cet algorithme, l'application est décrite par un DAG sans restrictions sur sa structure. Pour adapter le vol de travail à ce type de DAG, nous proposons d'ajouter des tâches sans coût de calcul. Ces tâches orienteront les vols vers des parties indépendantes du DAG. Ainsi, nous espérons réduire le nombre de données transférées au cours de l'exécution. Le chapitre 5 évalue les performances de cet algorithme. Dans un premier temps, nous proposons une analyse sur le nombre de transferts effectués au cours de l'exécution. Enfin, dans un environnement simulé, nous montrons que WSCOM réalise moins de transferts que les heuristiques hors-ligne et en-ligne existantes. Cette réduction du nombre de transferts permet de limiter l'impact de la congestion. Ainsi sur une plate-forme simulée de type grappe, WSCOM obtient des temps d'exécution inférieurs à ceux obtenus avec des algorithmes hors-ligne disposant des temps d'exécution des tâches et la quantité de données transférées.

WSCOM : vol de travail avec communications 4

Sommaire du chapitre

Le problème auquel nous nous intéressons est la parallélisation d'une application décrite par un Makefile [14] sur une plate-forme distribuée. Le langage utilisé par les Makefiles permet une description d'un ensemble de tâches à exécuter avec leurs relations de précédences et les transferts de données nécessaires. La structure du «Directed Acyclic Graph» (DAG) représentant l'application n'est pas restreinte par ce langage.

L'exécution d'une telle application est soumise à des contraintes spécifiques :
- temps d'exécution des tâches inconnus,
- quantités de données transférées entre deux tâches inconnues,
- structure du DAG fixée au début de l'exécution,
- topologie réseau inconnue.

Nous cherchons à résoudre le problème d'ordonnancement en-ligne correspondant, tout en essayant de limiter l'impact des communications.

Nous proposons dans ce chapitre différents algorithmes d'ordonnancement basés sur le vol de travail.

Celui-ci nous permet d'obtenir un ordonnancement efficace avec peu d'information sur l'application (voir section 3.2.2). En revanche, le nombre de données transférées n'est pas forcement lié au nombre de vols. Ces transferts peuvent donc générer d'importants temps d'inactivité.

La prise en compte des coûts de communications est difficilement réalisable car les quantités de données transférées par une arête ne sont pas connues. Même si elles étaient connues, la prise en compte de cette information complexifie largement les méthodes d'ordonnancement.

Afin de minimiser l'impact des communications, nous utilisons une approche bi-objective. Le premier objectif est d'équilibrer la charge entre les machines. Pour ce faire, nous cherchons prioritairement à minimiser le temps d'exécution de l'application sans considérer le coût des communications. Tout en conservant l'équilibrage de la charge, nous tentons de plus d'optimiser un second objectif qui est la minimisation du nombre de communications effectuées. Nous espérons qu'en réduisant le nombre de transferts, nous réduirons l'impact des communications, et ce, même sans connaissance de la topologie réseau.

Notre problème d'ordonnancement nous permet de disposer avant l'exécution de la structure du DAG. Nous souhaitons utiliser cette information au moment des vols pour prendre un ensemble de tâches ayant une grande affinité entre elles (les tâches utilisent un ensemble de données identiques).

Nous commençons, dans la section 4.1, par mettre en évidence les points faibles du vol de travail appliqué directement à notre problème. La section 4.2 présente ensuite les modifications apportées au vol de travail pour extraire des blocs de tâches fortement connectés entre elles et faiblement avec le reste de l'application. La section 4.3 présente enfin plusieurs manières de gérer l'exécution du DAG ainsi que les transferts de données.

4.1 Communications et vol de travail

Nous détaillons dans un premier temps la réalisation des communications par le vol de travail classique et nous exhibons différentes pistes d'amélioration.

La gestion des communications dans le vol de travail a été détaillée dans le cadre de l'implantation de *Cilk*, section 3.2.3. Dans ce logiciel, les transferts de données sont gérés implicitement par le matériel et le système d'exploitation, car l'exécution est limitée aux plates-formes à mémoire partagée.

Si nous cherchions à utiliser ce modèle de programmation en mémoire distribuée, nous aurions à gérer les transferts de données. Ces transferts proviennent de l'envoi des données après la réalisation d'un vol et du rapatriement des résultats. Les envois des données peuvent être effectués :

- au dernier moment lors de l'accès aux données,
- au moment du vol : le voleur est l'un des processeurs qui accédera aux données. Mais la tâche volée peut ne pas accéder aux données, par exemple la tâche utilise uniquement les indices d'un tableau pour découper le travail.
- à tout instant entre le vol et l'accès.

Nous détaillons maintenant le nombre de transferts réalisés par le vol de travail classique dans le cas où les données sont transférées au moment du vol.

Nombre de transferts

Avec le modèle «Fork-Join» utilisé par *Cilk*, les transferts ont lieu potentiellement :

- lors d'un vol,
- lors de la fusion des résultats des tâches volées (par exemple, lors d'une synchronisation utilisant le mot clef «Sync»).

Ainsi, pour chaque vol, il y a au plus deux transferts de données. Le nombre de transferts est donc proportionnel au nombre de tâches volées pour les DAG ayant une structure «Fork-Join».

Nous utilisons maintenant le modèle d'applications générales comme celles décrites par les Makefiles. Les DAG extraits des Makefiles, peuvent avoir une structure quelconque. Nous montrons par un exemple que le nombre de transferts effectués n'est pas proportionnel au nombre de vols. L'exemple de DAG utilisé est représenté par la figure 4.1.

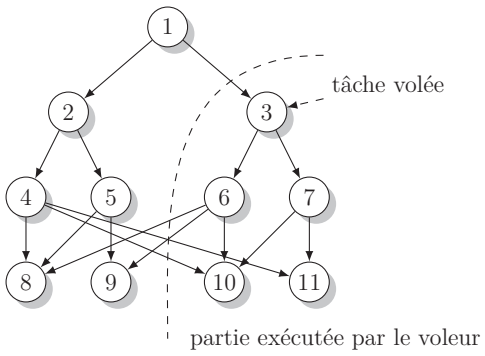


FIGURE 4.1 – Dé-corrélation du nombre de vol et du nombre de transferts.

Dans le cas où l'application est exécutée par deux processeurs, le premier travaille sur la partie de gauche et le second sur la partie de droite. Les tâches ont le même temps d'exécution. Durant l'exécution, il n'y aura qu'un vol réussi comme le montre la figure 4.1. Or, le nombre de transferts est égal à 5. Cet exemple peut se généraliser pour un nombre quelconque de transferts.

Lorsque le DAG est quelconque, les transferts ne sont pas réalisés uniquement entre le voleur et le volé. Pour effectuer les transferts, l'émetteur et le récepteur doivent être identifiés. Ainsi la gestion des transferts n'est pas triviale pour la parallélisation d'applications décrites par un Makefile.

Mise en pratique sur un cas simple

Dans cette section, nous appliquons directement le vol de travail sur un exemple simple afin de mettre en évidence les améliorations possibles.

La figure 4.2 illustre l'application utilisée comme exemple. Ce schéma est assez fréquent dans les Makefiles. Nous considérons l'exécution de cette application par un ensemble de processeurs. À l'initialisation, le nombre de tâches prêtes est assez grand, un placement de celles-ci peut être effectué. Sans connaître leur temps d'exécution, le travail sera déséquilibré dans la plupart des cas. Dans ce cas, certains processeurs devront voler du travail. Nous considérons que l'ensemble des tâches est dans la pile d'un processeur. Le processeur 1 commence par exécuter une tâche prête, le second tente de voler le processeur 1. La tâche prête exécutée fait partie du bloc a. Au moment du vol, le processeur choisit la ou les tâches volées. Ce choix impacte directement le nombre de transferts.

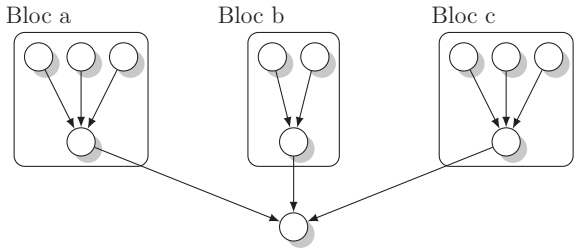


FIGURE 4.2 – Illustration d'un DAG de Makefile classique avec une découpe en blocs.

Ainsi, il est clair que l'algorithme doit éviter de fournir une tâche fortement liée à celle en cours d'exécution (comme une de celles du bloc a qui n'est pas encore exécutée). De plus, si une seule tâche est fournie au processeur 2, cela n'est pas suffisant pour en rendre d'autres prêtes. Ce processeur risque de lancer une nouvelle requête de vol rapidement. Il reprendra une tâche qui n'est pas nécessairement en relation avec celle qu'il vient d'exécuter.

Dans cet exemple, l'algorithme a intérêt de voler plusieurs tâches pour être efficace. Par exemple, les tâches du bloc c (ou même les blocs b et c) sont relativement indépendantes du bloc a. De plus, le ou les blocs volées contiennent des quantités de travail permettant d'équilibrer la charge efficacement.

4.2 Extraction d'un ensemble de tâches

Pour réaliser la parallélisation d'une application décrite par un Makefile, le mécanisme de vol de travail peut être appliqué directement mais son efficacité risque d'être réduite comme nous l'avons détaillé dans la section 4.1.

Afin de réduire l'impact des communications et d'améliorer l'équilibrage de la charge, nous souhaitons extraire au moment du vol un ensemble de tâches représentant la moitié du travail présent sur la machine volée et travaillant sur des données communes.

Si le choix de l'ensemble de tâches est calculé au moment du vol, la découpe du DAG sera réalisée en concurrence avec le travail effectué par la victime. Nous souhaitons éviter de perturber celle-ci. L'idée est d'ajouter des tâches virtuelles (une tâche sans travail qui génère des tâches) au sein du DAG et d'utiliser le mécanisme de vol de travail classique.

Lors du vol d'une telle tâche, le voleur ajoute dans sa pile les successeurs de la tâche virtuelle. Ainsi, les tâches virtuelles permettront d'extraire des parties indépendantes du DAG. Ces tâches sont ajoutées au DAG de l'application avant son exécution. Les processeurs utilisent le mécanisme de vol de travail classique sur le DAG modifié. Le coût d'un vol est exactement le même que pour le vol de travail classique.

L'ajout des tâches virtuelles va modifier la partie du DAG volée lors d'une requête, mais aussi orienter l'exécution locale vers un ensemble de tâches travaillant sur des données proches. Il n'existe pas une unique méthode pour créer le DAG de tâches virtuelles. Nous détaillons trois méthodes différentes avec leurs avantages et leurs inconvénients.

La première méthode tente d'améliorer l'équilibrage de la charge en essayant de fournir exactement la moitié du travail. Nous obtenons cela en ajoutant un arbre binaire permettant de diviser le travail en deux à chaque niveau. La section 4.2.1 introduit l'utilisation d'une telle structure pour les tâches virtuelles.

Ensuite, nous nous intéressons à la possibilité d'adapter des solutions existantes pour réaliser l'ajout des tâches virtuelles. La section 4.2.2 détaille l'utilisation des regroupements calculés par des algorithmes d'agrégations dans la génération des tâches virtuelles.

Enfin, la section 4.2.3 présente l'algorithme Work-Stealing with COMmunication (WSCOM) permettant le transfert d'une sous partie du DAG. Les relations entre les tâches virtuelles sont obtenues en inversant les relations entre les tâches de l'application.

4.2.1 Extraction de la moitié du travail

Certaines études [27, 34] ont montré que le vol de travail est plus efficace lorsque le voleur prend un ensemble de tâches contenant approximativement la moitié du travail présent sur la victime. Ainsi, l'équilibrage de la charge entre les machines est amélioré.

L'obtention d'un tel comportement est possible en ajoutant des tâches virtuelles ayant une structure d'arbre binaire équilibré. Dans les deux sous-arbres liés à la tâche racine, le nombre de taches présentes sera proche. Les deux sous-arbres auront la même quantité de travail si le travail est réparti équitablement entre les tâches.

De plus, cet arbre représente un schéma classique de programmation souvent associé au vol de travail qui est le schéma de type «diviser pour régner». Ce type de programmation divise le travail en plusieurs parties de plus en plus petites à chaque niveau de récursion. Pour équilibrer la charge lors d'un vol, le voleur n'a qu'à prendre la tâche la plus haute dans la pile pour récupérer la moitié du travail.

La figure 4.3 présente l'utilisation d'un arbre binaire sur l'exemple précédent.

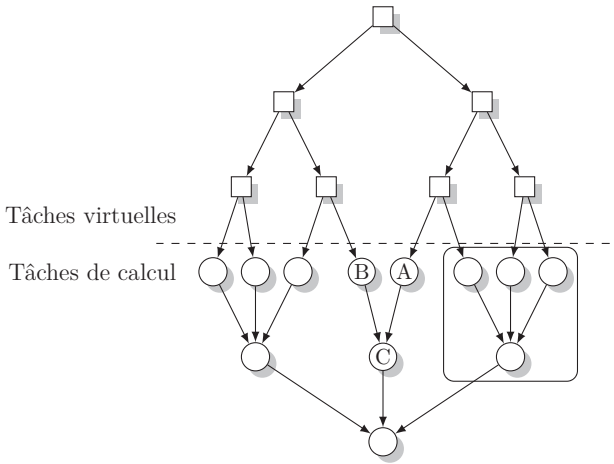


FIGURE 4.3 – Illustration d'une découpe avec un arbre binaire d'un DAG de Makefile.

La figure 4.4 illustre le début d'une exécution plausible par vol de travail du DAG introduit par la figure 4.3 sur trois machines. Lors du premier vol, le travail volé contient la tâche A. Celles-ci sont indépendantes des autres tâches (le bloc mis en évidence sur la figure 4.2). De plus, la tâche A est liée à une autre partie de l'application au travers de la tâche C.

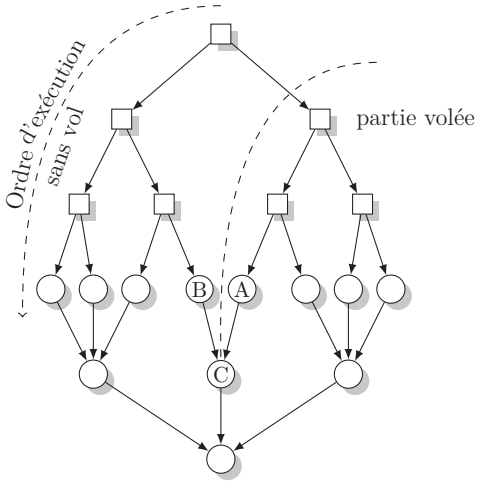


FIGURE 4.4 – Illustration de la partie prise lors d'un vol.

Dans cet exemple, la découpe de l'arbre binaire suit la structure du DAG de l'application. Ainsi, les successeurs des tâches virtuelles seront généralement fortement liés dans le DAG de l'application. L'objectif est d'éviter des DAG comme celui illustré figure 4.5. Dans un cas général, il est plus difficile d'éviter ce genre de situations.

Le comportement de cet algorithme peut être obtenu en modifiant légèrement le vol de travail classique. Il suffit qu'au moment d'un vol le voleur prenne la moitié des tâches ayant la profondeur la plus faible dans la pile. Pour les expériences, nous utiliserons un mécanisme de vol de travail appelé WS_HALF qui réalise le vol de la moité du travail présent dans la pile de la victime (sans sur-coût).

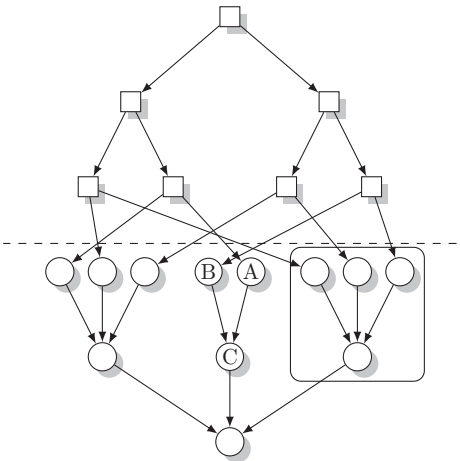


FIGURE 4.5 – Exemple de DAG généré ne préservant pas la localité.

Néanmoins, la construction du DAG avec les tâches virtuelles n'est pas triviale si l'algorithme tente de maximiser la localité dans chacun des sous-arbres. De plus, l'ensemble des tâches volées ne représente pas forcément une partie connexe. Cette solution risque de provoquer des transferts de données inutiles.

4.2.2 Extraction d'un agrégat de tâches

L'extraction de la moitié du travail permet d'équilibrer efficacement la charge entre les processeurs mais elle ne tente pas de minimiser le nombre de transferts. Nous souhaitons développer une méthode utilisant la structure du DAG. Parmi les algorithmes de la littérature, ceux présentés en section 2.2 calculent des agrégats de tâches qui travaillent sur des données communes. Nous détaillons l'utilisation de ces méthodes dans la génération des tâches virtuelles.

Parmi ces méthodes, nous commençons par les algorithmes présentés en section 2.2.2 (page : 30) qui cherchent à réduire l'impact des communications. Ces algorithmes comme Dominant Sequence Clustering (DSC) [91] ont tendance à regrouper les parties à ne pas paralléliser à cause d'un coût de communication prohibitif par rapport à la quantité de calcul. Or, dans notre problème, nous ne connaissons pas les temps de calcul et les coûts de communications. Il serait possible d'utiliser DSC avec des temps unitaires, mais ce n'aurait aucun impact car seules les parties du graphe séquentielles seraient regroupées. Ces algorithmes ne forment donc pas de bons candidats pour la génération des tâches virtuelles.

Nous nous intéressons donc à l'utilisation de l'algorithme récursif de regroupements convexes pour extraire une partie indépendante du DAG.

Regroupements convexes récursifs

Nous rappelons ici rapidement le principe des regroupements convexes présenté section 2.2.1. Un découpage du DAG de l'application est convexe, si en représentant chaque regroupement par une tâche, le graphe orienté ainsi formé est un DAG. Il existe de nombreuses manières de découper un DAG en respectant cette propriété comme nous l'avons détaillé section 2.2.1. Nous nous intéressons à l'algorithme introduit par Lepère [49]. Celui-ci est basé sur une découpe récursive en regroupements convexes.

Dans une première étape, l'algorithme commence par rechercher deux ensembles de tâches, A et B. Ces ensembles doivent être indépendants, *i.e.,* sans aucune relation de précédences. Ainsi, ils pourront être exécutés sans communication. L'algorithme a pour objectif de maximiser la quantité de travail présent au sein du plus petit ensemble. La figure 4.6 illustre les ensembles A et B obtenus sur un exemple. Une décomposition en regroupements convexes est calculée à partir de ces agrégats.

Le point central de cet algorithme est la recherche des ensembles A et B. Ce problème est NP-difficile [35]. Il existe néanmoins des heuristiques pour obtenir de tels ensembles. La qualité de la solution est évaluée en fonction des temps d'exécution. Dans notre problème, la qualité de la solution est évaluée en fonction du nombre de tâches présentes

dans le petit des deux agrégats (dans A ou dans B). L'objectif est de maximiser cette quantité. Puis, cet algorithme est appliqué récursivement aux regroupements A et B.

L'objectif est d'utiliser la découpe fournie par cet algorithme pour extraire du travail au moment du vol. Pour ce faire, nous ajoutons des tâches virtuelles qui orienteront les vols vers les agrégats indépendants lorsqu'ils seront prêts.

La figure 4.6 montre une découpe obtenue pour un DAG. Expliquons maintenant comment réaliser l'ajout des tâches virtuelles sur cet exemple.

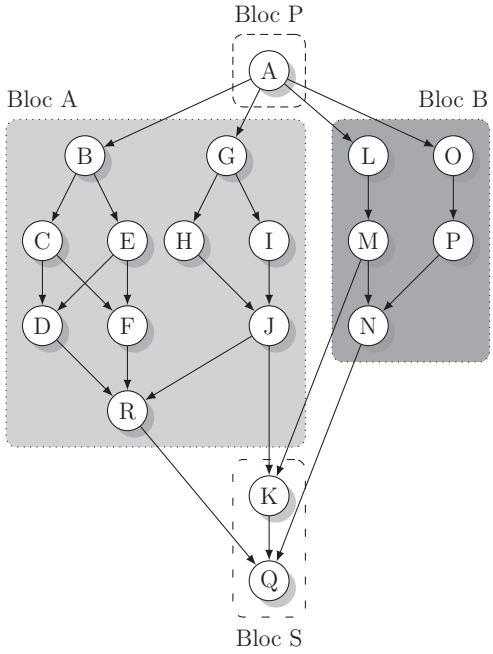


FIGURE 4.6 – Illustration d'une découpe en groupes convexes.

Dans le découpage effectué, il y a entre deux et quatre agrégats différents à chaque niveau de récursion. Les quatre types d'agrégats possibles sont :

- L'agrégat des prédécesseurs communs aux tâches des deux groupes indépendants noté P,
- L'agrégat des successeurs communs aux deux regroupements indépendants noté S,
- Les deux agrégats indépendants A et B.

Lors de la découpe, l'algorithme crée au minimum les agrégats A et B.

Chaque tâche de l'agrégat P, a parmi ses successeurs directs ou indirects au moins une tâche de A et une de B. Pour exécuter l'ensemble des tâches de A ou de B, les tâches de l'agrégat P doivent être toutes exécutées. De même, l'agrégat S pourra être exécuté uniquement après l'exécution de l'ensemble des tâches présentes dans $A \cup B$.

L'objectif des tâches virtuelles est de permettre le vol d'un des agrégats en entier. Nous associons donc une tâche virtuelle à chacun de ces agrégats. Pour chaque agrégat X, les tâches terminales sont celles qui n'ont pas de successeur présent dans X. L'ensemble des tâches terminales est noté X_t. À l'opposé, les tâches initiales sont celles qui n'ont pas de prédécesseurs dans X. Cet ensemble est noté X_i.

La tâche virtuelle associée au regroupement A, a comme prédécesseur l'ensemble des tâches appartenant à P_t et comme successeur l'ensemble de celles appartenant à

A_i. Nous avons la même chose pour l'agrégat B. Pour l'agrégat S, la tâche virtuelle associée a comme prédécesseur l'ensemble des tâches appartenant à $A_t \cup B_t$ et comme successeur l'ensemble de celles appartenant à S_i. Et enfin pour l'agrégat P, la tâche virtuelle a comme successeur l'ensemble des tâches appartenant à P_i.

L'ajout des tâches virtuelles est réalisé de manière identique à chaque niveau de la récursion. La figure 4.7 illustre le DAG résultant de cet ajout. Ces tâches sont représentées par des carrés dans la figure. Les tâches virtuelles de chaque niveau de récursion doivent être reliées à celles du niveau de récursion précédent.

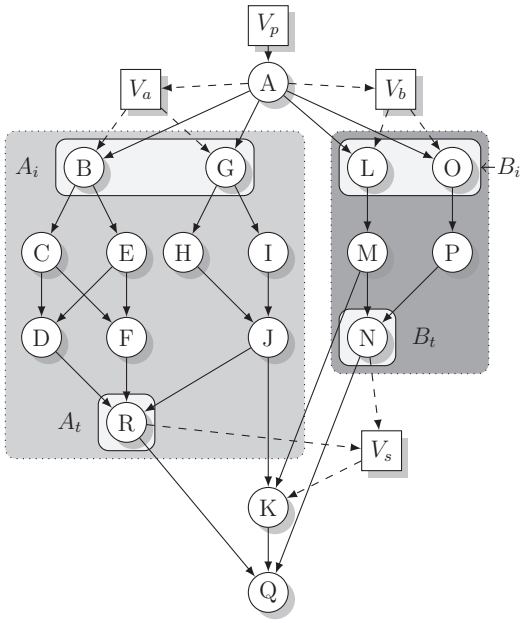


FIGURE 4.7 – DAG obtenu après l'ajout des tâches virtuelles.

Cette découpe allonge en général sensiblement le chemin critique de l'application à cause des dépendences virtuelles ajoutées entre les tâches. En revanche, avec de telles tâches virtuelles, le processeur qui commence à en exécuter une, travaillera sur une partie indépendante du DAG.

4.2.3 Extraction d'un sous DAG

Dans les sections précédentes, nous avons détaillé la possibilité d'ajouter des tâches virtuelles en se basant sur des algorithmes existants.

Nous proposons maintenant un algorithme original qui extrait au moment d'un vol un sous-ensemble du DAG. Cet algorithme est basé sur une découpe récursive du travail à effectuer. Il parcourt le DAG des puits vers les sources pour construire le DAG avec les tâches virtuelles. Nous appelons cet algorithme «WSCOM»(en anglais Work-Stealing with COMmunications). Nous détaillons maintenant le fonctionnement de l'algorithme.

Principe

Nous illustrons le principe de fonctionnement de WSCOM dans le cas où le graphe de tâches forme un anti-arbre. En effet, son fonctionnement est alors très clair.

Lors de la description d'une application avec un Makefile, l'utilisateur commence par définir la cible finale. Puis, il ajoute les cibles nécessaires à sa réalisation. Ces cibles sont indiquées par une liste de dépendances associées à la cible finale. Récursivement pour chacune, l'utilisateur indique la commande à exécuter et la liste des dépendances. L'anti-arbre implique que les travaux présents dans la liste de dépendance sont indépendants. Les dépendances nous fournissent ainsi un support pour extraire du travail. Un DAG simple de makefile est illustré figure 4.8.

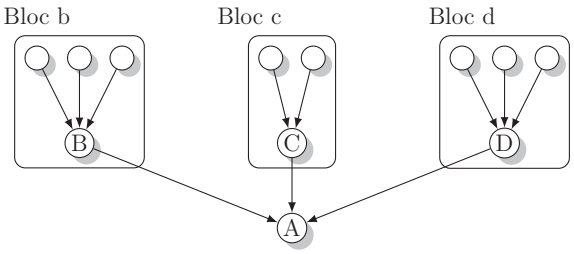


FIGURE 4.8 – Illustration d'un DAG de Makefile classique.

Nous présentons l'utilisation des dépendances pour extraire des parties du DAG en détaillant le mécanisme de l'algorithme. Dans la figure 4.8 lorsqu'un processeur travaille sur le bloc b de tâches, le voleur doit s'orienter vers les tâches du bloc c ou d. Le processeur qui travaille sur le bloc b exécute les prédécesseurs de la tâches B. Le voleur doit prendre les prédécesseurs de la tâches C ou D.

Pour obtenir cela, WSCOM utilise la symétrie du DAG (*i.e.,* le DAG de l'application en inversant le sens des arêtes) pour générer les tâches virtuelles. La figure 4.9 représente le DAG avec les tâches virtuelles créées à partir du DAG de la figure 4.8.

En utilisant le vol de travail classique sur ce DAG, le premier processeur travaillera sur les successeurs de la tâches V_B qui sont les prédécesseurs de la tâche B. Lorsqu'un voleur vole une tâche virtuelle comme V_C ou V_D, il se retrouve à travailler sur les successeurs de la tâche volée. Ces successeurs sont aussi les prédécesseurs de la tâche de calcul associée à la tâche virtuelle. La partie du DAG obtenue après le vol d'une tâche virtuelle est mis en évidence dans la figure 4.9.

En plus de la symétrie du DAG, nous avons ajouté des arêtes entre les tâches virtuelles et leur symétrique. Lors de l'exécution d'une tâche virtuelle, tous ces successeurs directs sont ajoutés à la pile de tâches. Ainsi, certaines tâches dans la pile ne sont pas prêtes. Cet ajout de tâches non prêtes ne pose pas de problème particulier (voir section 3.2.3 page : 47) Les tâches qui ne sont pas prêtes ne peuvent pas être volées.

La figure 4.10 montre sur un exemple l'état de la pile avant et après avoir traité une tâche de création. La gestion de la pile est proche de celle réalisée dans le mécanisme de vol de travail. Lorsqu'un processeur est inactif, il vole la tâche au sommet de la pile d'un autre processeur sélectionné aléatoirement comme dans le mécanisme de vol de travail classique.

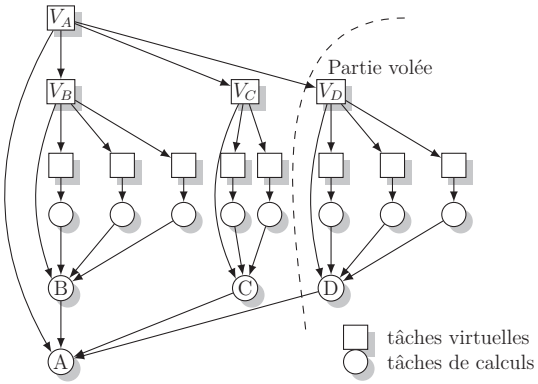


FIGURE 4.9 – Ajout de tâches virtuelles en utilisant la symétrie.

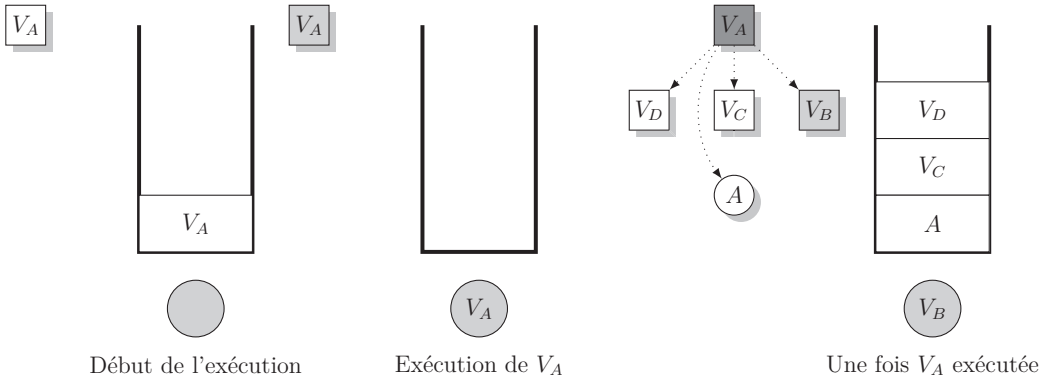


FIGURE 4.10 – Illustration de l'exécution du DAG de la figure 4.9.

Structure de DAG quelconque

L'exemple utilisé précédemment est simple et ne permet pas de mettre en avant tous les détails de l'algorithme. La principale différence provient du fait que les tâches ont au plus un successeur. Dans le cas d'un DAG quelconque, certaines tâches virtuelles ont plusieurs arêtes entrantes. Nous détaillons maintenant la gestion de l'exécution dans ce cas.

La figure 4.11 montre un exemple où deux tâches B et C dépendantes de la même tâche A. Le symétrique des tâches est composée de la tâche virtuelle associée à A appelée V_A. La tâche V_A peut être accédée par un chemin provenant de la tâche V_B et de la tâche V_C. Dans un cas classique, les algorithmes d'ordonnancement attendent l'exécution des deux tâches pour exécuter la tâche V_A. Or dans notre cas, les tâches sont virtuelles. Pour exécuter la tâche A ou l'ensemble des tâches dépendantes de A, il n'est pas nécessaire d'attendre l'exécution des deux tâches V_B et V_C.

Les dépendances entre les tâches virtuelles ne sont pas identiques aux dépendances entre les tâches. Nous les distinguerons dans les figures suivantes par des lignes en pointillées.

Le travail représenté par V_A n'est exécuté qu'une seule fois, l'algorithme d'ordonnancement doit donc choisir quelle tâche de V_B ou V_C crée V_A. Nous nommons ce problème le conflit de création de tâches. La sélection de l'arête correspondante peut être effectuée

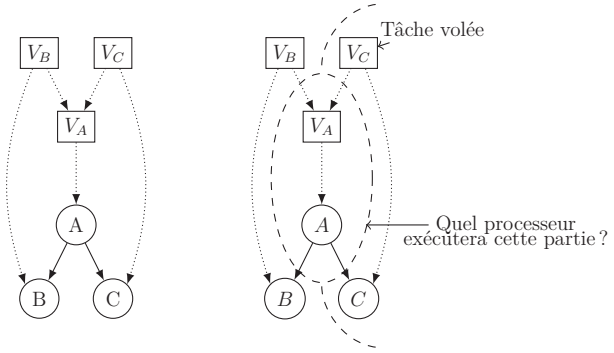


FIGURE 4.11 – Deux tâches dépendantes d'une même tâche.

avant où pendant l'exécution. Nous détaillons dans la section 4.3.1 ces deux possibilités.

4.3 Gestion de l'exécution

Nous présentons maintenant plus en détail le fonctionnement de WSCOM.

Le mécanisme de vol de travail est utilisé généralement sur des DAG dont la structure est restreinte. Par exemple *Cilk* utilise un DAG de type Fork-Join, KAAPI utilise un modèle de programmation permettant de gérer les dépendances uniquement par celui qui crée la tâche. Dans notre cas, les dépendances sont quelconques. Pour exécuter l'ensemble des tâches, les processeurs doivent savoir si la tâche est prête et où se situent les données nécessaires.

De plus, nous avons ajouté des tâches virtuelles. Elles peuvent être exécutées même si certains de leurs «prédécesseurs» (virtuels) ne le sont pas. Un mécanisme permettant de gérer leur exécution est donc nécessaire.

Durant l'exécution, l'algorithme d'ordonnancement doit permettre aux processeurs de savoir si une tâche est prête ou non. Il doit aussi gérer les transferts de données.

La section 4.3.1 décrit la gestion des dépendances utilisées par nos mécanismes de vol de travail pour permettre l'exécution sur plate-forme distribuée. Puis, la section 4.3.2 introduit la gestion des communications au cours de l'exécution afin de limiter leurs impacts.

4.3.1 Gestion des dépendances

Dans les applications considérées, les relations de dépendances sont quelconques. Avant l'exécution d'une tâche, le processeur recherche si elle est prête ou non. Pour optimiser l'exécution, cette recherche doit être efficace. La gestion des dépendances peut être effectuée de deux manières :

- Gestion à l'exécution de la tâche : le processeur qui tente d'exécuter la tâche demande aux processeurs qui possèdent les prédécesseurs, si ceux-ci sont exécutés.
- Gestion à l'exécution des prédécesseurs : le processeur qui termine l'exécution d'une tâche fournit l'information aux processeurs qui possèdent un des successeurs.

Dans le cas de la gestion à l'exécution de la tâche, le processeur attend la réponse pour pouvoir exécuter celle-ci. Les messages sont envoyés sur le réseau, les temps d'attente peuvent donc être importants. Cela risque de ralentir l'exécution du travail.

Dans le cas de la gestion à l'exécution des prédécesseurs, le processeur envoie les messages et exécute une autre tâche sans attendre la réponse. Le processeur devra juste s'assurer qu'il n'y a pas eu d'erreur durant l'envoi.

Pour éviter les temps d'attente, nous utilisons donc la gestion à l'exécution des prédécesseurs. Dans la gestion des dépendances, les processeurs ont besoin de connaître le processeur auquel l'information de la fin d'exécution doit être envoyée. Pour un DAG avec E arêtes et N nœuds, nous comparons plusieurs méthodes permettant de gérer les dépendances :

- Lorsqu'un processeur exécute une tâche dont un des successeurs n'est pas dans sa pile locale, il envoie à tous les processeurs l'information (p messages par nœuds).
- Lorsqu'un processeur exécute une tâche dont un des successeurs n'est pas dans sa pile locale, le processeur contacte les processeurs concernés en suivant les vols (au plus p messages par nœuds, la recherche peut mener à l'ensemble des machines).
- Une Distributed Hash Table (DHT) regroupe l'information. Pour chaque tâche, la DHT est prévenue de son possesseur. Lorsqu'un processeur exécute une tâche dont un de ces successeurs n'est pas dans sa pile locale, le processeur contacte la DHT qui recherche les possesseurs des successeurs (pour un successeur au plus 2 messages pour le contacter et un message par nœud pour mettre à jour la DHT).

L'algorithme que nous avons développé est donc basé sur une table de hachage distribuée (Distributed Hash Table (DHT) [77]). Nous affectons la gestion d'un ensemble de tâches à chacune des machines participantes à l'exécution. Le processeur qui se charge d'une tâche, regroupe les informations sur l'exécution des prédécesseurs, sur le possesseur de la tâche et sur l'exécution de celle-ci. Dès qu'une tâche devient prête, l'information est envoyée au processeur la possédant.

Ainsi, il est possible de savoir si une tâche est prête ou non au cours de l'exécution. Comme nous l'avons détaillé section 4.2.3, les tâches virtuelles sont gérées différemment des tâches de calcul. L'exécution d'une tâche virtuelle peut être effectuée avant la fin de l'exécution de l'ensemble de ses prédécesseurs.

Pour une tâche virtuelle comme la tâche V_A dans la figure 4.11, l'algorithme d'ordonnancement doit choisir l'arête qui activera son exécution. Cette décision peut être prise avant ou durant l'exécution. Nous présentons ici les deux possibilités.

Avant l'exécution de l'application

Dans le DAG de l'application, il y a deux types de tâches : calcul et virtuelle. Les tâches de calculs sont celles présentes initialement dans le DAG de l'application. Les tâches virtuelles sont celles ajoutées pour orienter les vols vers une partie du sous-DAG. Dans le cas d'un conflit de création, un seul prédécesseur doit réaliser la création de la tâche virtuelle. Il suffit donc de supprimer toutes les arêtes entrantes de la tâche en conflit sauf une pour résoudre le problème.

Dans le cas de WSCOM, les choix pour l'ensemble des tâches virtuelles qui ont plusieurs arêtes entrantes sont équivalents à la sélection d'un des arbres couvrant du DAG.

Pour détailler l'impact de l'arbre couvrant sélectionné, nous utilisons un exemple illustré par la figure 4.12.

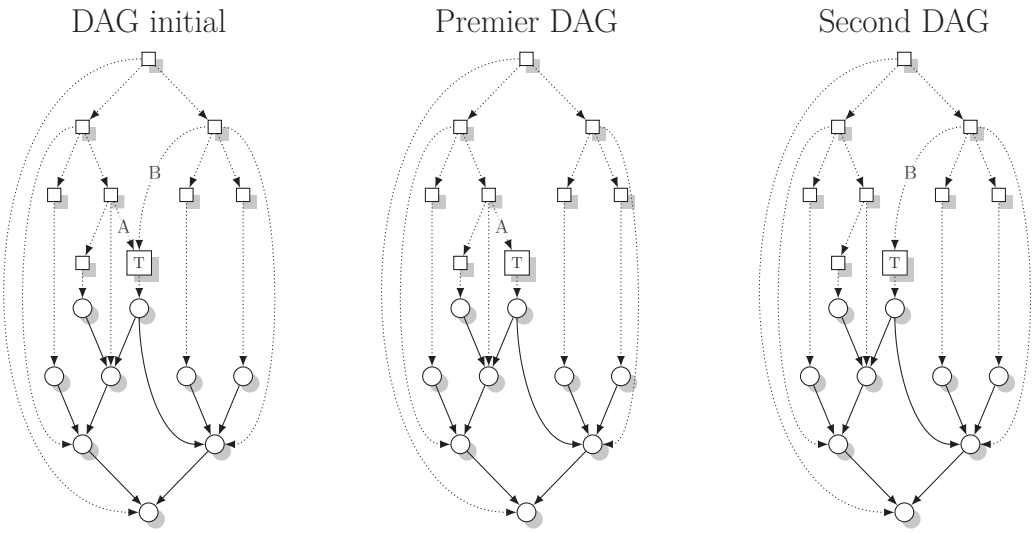


FIGURE 4.12 – Longueur du chemin critique en fonction du choix réalisé.

L'algorithme sélectionne l'arête qui créera la tâche T dans cet exemple. Les deux possibilités sont illustrées dans la figure 4.12.

Dans un premier cas, l'algorithme choisit la possibilité détaillée par le premier DAG de la figure 4.12. Nous pouvons calculer la profondeur du DAG en considérant le temps d'exécution des tâches unitaires, qui est de 8. Dans le cas où l'algorithme choisit l'autre arête, la profondeur du DAG est de 7. Or, pour le vol de travail, la profondeur du DAG impacte directement le temps d'exécution et le nombre de vols. Au premier abord, nous pourrions penser que la deuxième solution fournira un meilleur temps d'exécution.

La différence entre les deux profondeurs est due à une réduction du nombre de tâches virtuelles sur le chemin critique. Le temps d'exécution de ces tâches est généralement bien plus faible que le temps d'exécution des tâches de calcul. La différence sera donc négligeable.

En revanche, ce choix définit la partie qui aura accès à la tâche T et ces potentielles dépendances. Ce choix est réalisé sans connaître ni les temps d'exécution ni les quantités de données transférées. L'algorithme ne peut donc pas savoir quelle partie contient le moins de travail.

Réalisé avant l'exécution, le choix doit être effectué sans connaître l'impact sur la répartition du travail entre les processeurs. Nous avons développé un algorithme choisissant un arbre couvrant, utilisant un parcours en largeur. L'algorithme conserve la première arête accédant au nœud. Il choisit donc un des arbres couvrant, minimisant la profondeur du DAG. Nous appelons cet algorithme $\text{WSCOM}_{\text{tree}}$.

Durant l'exécution de l'application

Le processeur qui créera le sous-DAG associé à une tâche virtuelle, peut être déterminé au cours de l'exécution. Durant celle-ci, une DHT est déjà présente pour gérer les dépendances entre les tâches. L'idée est d'utiliser cette même DHT pour gérer la création des tâches virtuelles.

Ainsi, chaque processeur ajoutera la tâche virtuelle à sa pile. Lorsqu'il l'exécutera, il communiquera avec la DHT pour savoir s'il crée les tâches filles ou non.

La DHT fournit la tâche virtuelle au premier processeur qui la réclame. Pour justifier ce choix, nous utilisons un exemple illustré figure 4.13 qui représente une sous-partie d'un DAG plus complexe. Deux processeurs exécutent ces tâches. Un processeur exécute

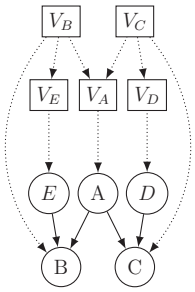


FIGURE 4.13 – Exemple détaillant les choix possibles pour la DHT.

la tâche E, et l'autre la tâche D. Le processeur qui exécute la tâche E termine en premier. Il tente de contacter la DHT pour générer la suite.

- Si la DHT ne l'autorise pas à créer les tâches filles, le processeur envoie une requête de vol vers un autre processeur. La partie volée risque de n'avoir aucun lien avec le travail effectué précédemment. Cela générera des transferts supplémentaires.
- Si la DHT l'autorise, le processeur exécutera la tâche A. Ainsi, le résultat sera obtenu plus rapidement. La suite de l'exécution poursuivra sur les tâches B et C avec les mêmes processeurs.

Dans ce cas, la DHT doit fournir la tâche virtuelle au premier processeur demandant son exécution.

Dans le cas où la tâche A est volée, cela générera deux transferts de données. Si cette partie du DAG avait été exécutée par deux processeurs, il n'y aurait eu qu'un transfert. Or, la DHT ne connaît pas les temps d'exécution des tâches ni les transferts de données. Dans notre approche, nous tentons d'équilibrer la charge avant de minimiser les transferts. La DHT doit donc fournir la tâche au voleur. Ainsi, ces trois machines ont du travail à exécuter.

La DHT de notre implantation fournit la tâche au premier processeur la demandant pour deux raisons distinctes :

- Si la tâche a été volée, cela permet d'équilibrer la charge,
- Sinon d'éviter qu'un processeur vole du travail non connecté à celui précédemment exécuté.

4.3.2 Gestion des transferts de données

Après avoir géré la création des tâches et leurs dépendances, l'exécution nécessite la présence des données lues et écrites par la tâche, sur la machine.

Considérons deux tâches A et B telles que B exécute un calcul sur les données produites par A. B ne pourra pas être exécutée avant que A ne soit terminée et les données produites transférées.

Pour envoyer les données produites par A, le processeur qui exécutera B doit être connu. Or la tâche B peut être volée à tout moment lorsqu'elle est prête. De plus dans le mécanisme de vol de travail classique, la tâche B est créée sur le processeur l'activant. Même sans vol, le processeur potentiel pour la tâche B est connu au dernier moment.

La première solution est d'attendre qu'un processeur tente d'exécuter la tâche. Ce processeur demandera aux processeurs ayant des données nécessaires de les transférer vers lui. Afin de préciser les principes de fonctionnement de WSCOM, une description en pseudo-code est réalisée par l'algorithme 1. Cette solution fonctionne mais le transfert des données peut introduire d'importants temps d'attente avant l'exécution des tâches et ainsi allonger le temps d'exécution.

Nous détaillons la réalisation d'envoi des données en avance, au processeur qui exécutera potentiellement la tâche. Cette partie est principalement reliée à l'algorithme WSCOM utilisant la symétrie du DAG. Une approche identique peux être utilisée avec l'algorithme récursif de regroupement convexe.

Pré-transferts des données

Dans les algorithmes hors-ligne, les données peuvent être envoyées dès leur génération car le processeur qui exécutera la tâche destinataire des données est connu.

Pour connaître le processeur plus tôt dans l'exécution, le premier point est de créer la tâche avant qu'elle ne devienne prête. Pour créer les tâches en avance, nous choisissons lors de l'exécution d'une tâche virtuelle d'ajouter la tâche de calcul reliée à celle-ci. Lors de l'utilisation de la symétrie, la tâche de calcul est ajoutée à la pile, et ce, dès l'exécution de la tâche virtuelle.

Il est fort probable que la tâche de calcul ne change pas de processeur. Les processeurs produisant des données pour cette tâche peuvent envoyer les données à celui qui l'a ajoutée à sa pile. Dans le cas où cette tâche serait volée, les données seront retransmises vers le voleur. La tâche ne peut pas être volée plus d'une fois car dès qu'elle est volée, le processeur l'exécute.

Dans le cas où les données sont volumineuses, il est préférable d'empêcher les vols des tâches de calcul. Nous appelons cet algorithme WSCOM_{pf} (pf : prefetching).

Cette restriction du vol aux tâches virtuelles peut impacter l'équilibrage de la charge. Puisque nous ne connaissons pas le temps d'exécution des tâches, un processeur peut générer un grand nombre de tâches dans sa pile sans pouvoir les exécuter. Lorsqu'elles deviennent prêtes, elles ne peuvent plus être volées. Ainsi, le processeur qui a créé beaucoup de tâches devra les exécuter seul.

Algorithme 1 : Work-Stealing with COMmunication (WSCOM) avec envoi des données lors de l'exécution d'une tâche

```
Data : G(V,E) ;                                                    /* DAG */
Data : Pile_id ;     /* Pile de tâches associée au processeur courant */
T_cur = non défini ;
Puits=Puits (G) ;                                  /* Un seul puits dans G */
if le processeur a l'identifient 0 then
    T_cur = Virtuelle(S) ;        /* Création d'une tâche virtuelle de S */
while la tâche Puits n'est pas exécutée do
    while T_cur est défini do
        if T_cur est une tâche virtuelle then
            if !Exécutée (T_cur) then
                T_calcul = Tâche_de_calcul_associée(T_cur) ;
                for I ∈ Prédécesseurs(T_calcul) do
                    push (Pile_id,Virtuelle(I)) ;
                push (Pile_id, T_cur) ;
        else
            Rapatriement_des_données (T_cur) ;
            Exécution (T_cur) ;
            MAJ_dépendances (T_cur) ;
        T_cur = Pop_tâche_prête(Pile_id) ; /* Suppression d'une tâche prête
        */
    P = Sélection_aléatoire_d'un_processeur () ;
    Tâche = Vol (P) ;
    if Tâche est défini then
        push (Pile_id, Tâche) ;;
    T_cur = Pop_tâche_prête(Pile_id) ;
```

Dans les expériences, nous montrerons que cette restriction a un impact variable en fonction des DAG considérés.

Analyse de WSCOM 5

Sommaire du chapitre

Nous avons présenté au chapitre précédent différents algorithmes d'ordonnancement basés sur l'ajout au vol de travail de tâches virtuelles. Au sein de ces algorithmes, différentes variantes ont été proposées.

Nous nous intéressons désormais à l'évaluation des performances de ces algorithmes. Pour commencer, la section 5.1 introduit une analyse théorique de la quantité de transferts effectués par les algorithmes en-ligne déterministes.

Nous réalisons également une série d'expériences pratiques. Nous évaluons les performances des différents algorithmes proposés en les comparant aux meilleurs algorithmes de la littérature. La section 5.2 détaille cette analyse basée sur des simulations. Enfin, la section 5.3 synthétise les contributions de la partie II et présente des possibilités d'élargir le travail autour de l'algorithme Work-Stealing with COMmunication (WSCOM).

5.1 Analyse théorique

Dans cette section, nous nous intéressons à l'étude des transferts de données par les algorithmes en-ligne. Pour réaliser cette étude, nous utilisons les notations classiques

que nous rappelons rapidement ici. La quantité de travail présente au sien l'application est notée W, son chemin critique D et le nombre de processeur est noté p.

La section 5.1.1 montre qu'aucun algorithme en-ligne déterministe obtenant un temps d'exécution inférieur à $\frac{W}{p} + \sqrt{D}D$ ne peut obtenir un ratio de compétitivité inférieur à $\sqrt{D} - o(\sqrt{D})$ sur le nombre de communications. Puis, nous prouvons dans la section 5.1.2 que le nombre maximal de transferts effectués par WSCOM pour des «Directed Acyclic Graph» (DAG) ayant une structure de type «Fork-Join» est de l'ordre de $O(p \times D)$.

5.1.1 Borne inférieure du nombre de transferts

Nous supposons, pour notre analyse, une arité du DAG égale à 2. Lorsque l'arité est plus grande, le nombre minimal de transferts est plus important.

Nous nous intéressons maintenant à montrer que tout algorithme en ligne *déterministe* doit réaliser au moins $\sqrt{D} - o(\sqrt{D})$ transferts pour obtenir un temps d'exécution proche de l'optimal.

Comme nous l'avons détaillé, l'algorithme en-ligne ne connaît pas les temps d'exécution des tâches. Nous basons notre analyse sur la création d'un adversaire qui choisit les temps d'exécution des tâches pour forcer l'algorithme d'ordonnancement à transférer des données. En plus de l'adversaire, nous utilisons un schéma de DAG particulier que nous appelons un ensemble de peignes. Un peigne est une ligne de taches qui ont deux successeurs, le prochain sur la ligne et un autre. Enfin, un ensemble de peignes est composé de peignes ayant tous une profondeur différente, ceux-ci sont reliées du plus grand au plus petit par une chaine de tâches. La figure 5.1 illustre ce schéma de DAG.

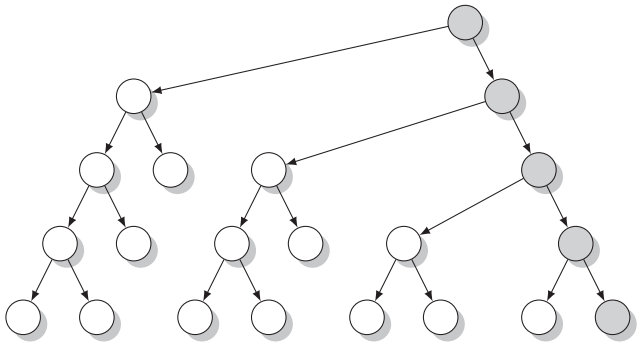


FIGURE 5.1 – Illustration d'un ensemble de peignes.

a Analyse sur un ensemble de peignes

Sur cet exemple, nous souhaitons montrer que pour un nombre de processeurs supérieur à 1, le nombre de données transférées est supérieur à $\sqrt{D} - o(\sqrt{D})$. Dans le cas contraire, le temps d'exécution est alors supérieur à $\frac{W}{p} + \sqrt{D}D$.

Pour réaliser cette analyse, l'adversaire force les processeurs à effectuer $\sqrt{D} - o(\sqrt{D})$ transferts sur la ligne de tâches grises. Dans le cas où l'algorithme décide de limiter le nombre de transferts, l'adversaire est alors capable de créer un déséquilibre de charge.

Lemme 5.1. *Nombre de transferts sur un ensemble de peignes Tout algorithme d'ordonnancement de tâches déterministe en-ligne qui exécute un ensemble de peignes de tâches sur p machines($p > 1$) en un temps inférieur à $\frac{W}{p} + \sqrt{D} \times D$, doit réaliser au moins $\sqrt{D} + o(\sqrt{D})$ transferts quand D est grand.*

Démonstration. Définissons d'abord sur une analyse de l'exécution a posteriori la notion de bloc. L'ensemble des tâches grises est découpé en blocs contigus de tâches exécutées par un même processeur. La figure 5.2 illustre la hauteur, la profondeur et la taille d'un bloc.

L'adversaire que nous proposons pour cette démonstration impacte le temps d'exécution des tâches. Afin de conserver la même longueur de chemin critique, l'adversaire assigne aux tâches un temps entre 1 et $D - d_i$ où d_i est la profondeur de la tâche. Les tâches qui ne sont pas des feuilles sont unitaires. Ainsi, la profondeur maximale est conservée égale à D. À chaque instant, l'adversaire affecte la valeur maximale aux tâches exécutées par le processeur travaillant sur le peigne le moins haut. Les autres processeurs obtiennent des tâches unitaires. Dès que le processeur travaillant sur les peignes les plus profonds commence l'exécution de son $(\sqrt{D}+1)^{\text{ième}}$ peigne d'un bloc contigu, l'adversaire continue d'assigner aux tâches de ce processeur une quantité de travail maximale et assigne à toutes les autres tâches la plus petite durée possible (*i.e.,* 1 si la tâche n'a pas encore commencé à être exécutée et s si la tâche s'est déjà exécutée t unités).

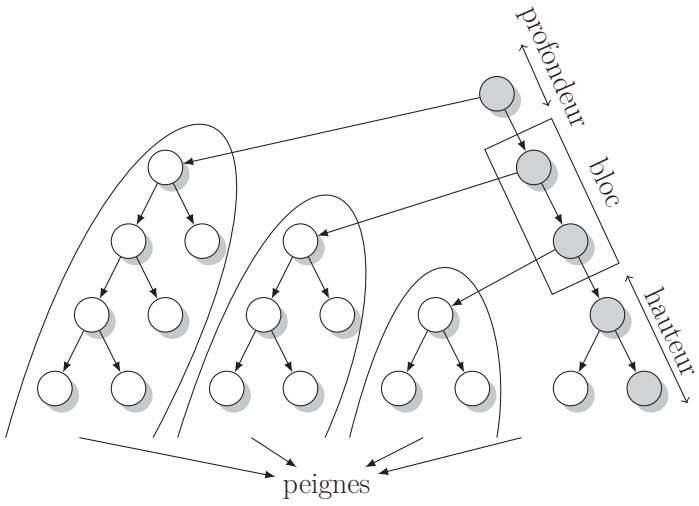


FIGURE 5.2 – Illustration d'un bloc.

La preuve fonctionne ainsi : si la taille des blocs est trop petite, le nombre de transferts entre les blocs sur la ligne de tâches grises sera trop grand. Dans le cas contraire, l'adversaire contraindra l'algorithme en-ligne à transférer les peignes présents sur le processeur ayant un bloc de grande taille, pour obtenir un temps d'exécution inférieur à $\frac{W}{p} + \sqrt{D}D$.

Nous distinguons deux cas : soit les blocs ont une taille inférieure à $\sqrt{D}$, soit il en existe au moins un de taille supérieure à $\sqrt{D}+1$.

- Dans le cas où la taille des blocs est toujours inférieure à $\sqrt{D}$, le nombre de blocs est supérieur à $\sqrt{D}$. Les processeurs auront donc réalisé au moins $\sqrt{D}$ transferts ce qui conclut la preuve pour ce cas.
- Nous considérons maintenant le cas où au moins un processeur exécute un bloc de tâches de taille supérieure à $\sqrt{D}+1$. Nous notons r le processeur qui exécute un bloc de taille supérieure à $\sqrt{D}+1$ et de hauteur maximale. Notons b le bloc associé au processeur r. Nous considérons le moment où le processeur r exécute la $(\sqrt{D}+1)^{\text{ième}}$ tâche grise du bloc b.
  L'adversaire continue alors d'assigner une quantité de travail maximale aux tâches que le processeur r exécutera. Les tâches qui seront exécutées par les autres processeurs, ont une quantité de travail minimale.
  Soit f un entier égal à $2\sqrt[4]{D}$.
  Il y a deux possibilités en fonction de la hauteur du bloc b.
  - Dans le cas où le bloc b a une hauteur inférieure à $f\sqrt{D}$, les tâches grises ayant une hauteur supérieure à $f\sqrt{D}$ sont divisées en blocs de taille inférieure à $\sqrt{D}$. Il y a donc au moins $\sqrt{D}-f$ blocs différents. Le nombre de transferts minimal est de l'ordre $\sqrt{D}-f$, ce qui est supérieur à la limite fixée.
  - Dans le second cas, nous savons maintenant qu'il y a au moins un bloc de taille $\sqrt{D}+1$ et dont la hauteur du $(\sqrt{D}+1)^{\text{ième}}$ peigne est supérieure à $f\sqrt{D}$.
    Nous rappelons que l'adversaire a affecté ou affectera aux tâches exécutées par le processeur r une quantité de travail maximale. Les tâches exécutées par les autres processeurs ont une quantité de travail minimale.
    Nous analysons maintenant la quantité de travail présent sur le processeur r, et le nombre de tâches présentes sur les autres processeurs. Dès que ce processeur a commencé à exécuter ce bloc, les autres processeurs ont eu des tâches avec une quantité de travail minimale.
    Les $\sqrt{D}+1$ peignes du bloc b ont appartenu au processeur r au moins un instant après l'exécution de la tâche grise correspondante. Le bloc b a une taille supérieure à $\sqrt{D}$ et une hauteur supérieure à $f\sqrt{D}$. Appelons h_{min} la hauteur du bloc. On a $h_{min} \geqslant f\sqrt{D}$.
    Le peigne le moins haut contient le moins de travail. Dans ce peigne, le processeur r a une quantité de travail égale à $h_{min} - 1 + \sum_{k=1}^{h_{min}-2}(k) = \frac{(h_{min})^2 - h_{min}}{2} + 1 \geqslant \frac{f^2 D - f\sqrt{D}}{2} = V_1$.
    Notons s le processeur qui a le plus de tâches localement. Notons N le nombre de blocs exécutés par le processeur s avec $s \neq p$ et M le nombre de peignes qu'il a pris. La figure 5.3 illustre la cas considéré. Chaque processeur autre que le processeur r a donc au plus N blocs plus M peignes et aussi les restantes en dessous du bloc b.
    Le nombre de peignes M pris par un processeur est inférieur à $\sqrt{D}$ car pour chaque peigne, un transfert est nécessaire. Nous compterons donc $N+1$ bloc sur le processeur s pour prendre en compte les peignes pris sur d'autre processeurs.
    Le nombre maximal de tâches dans un bloc de taille $\sqrt{D}$ est égal au nombre de

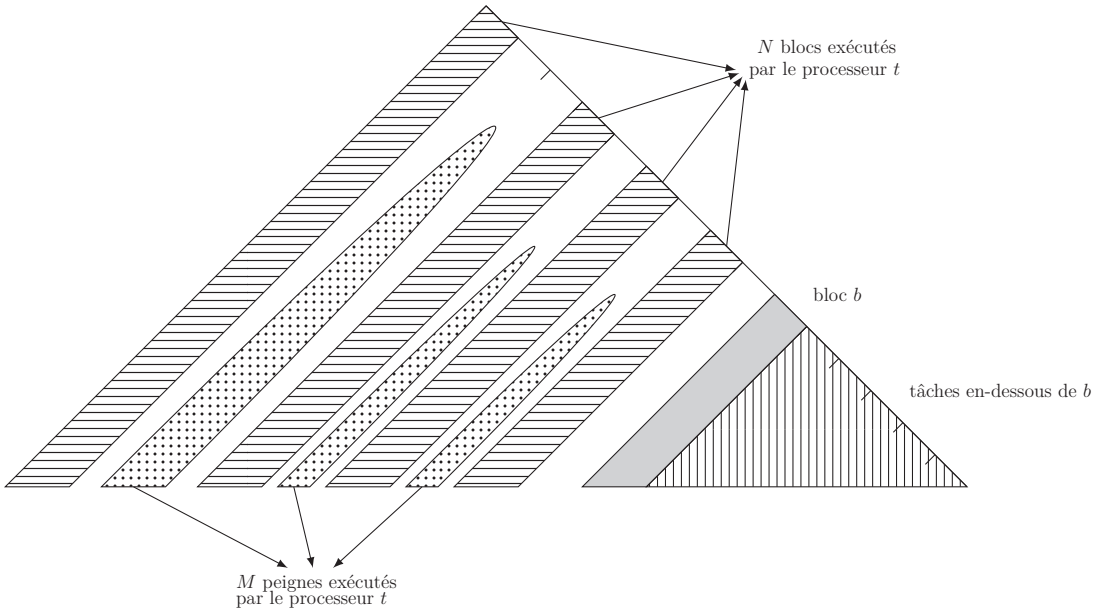


FIGURE 5.3 – Illustration de la répartition des tâches entre les processeurs s et r.

tâches dans le bloc partant de la racine. Ce nombre est inférieur à $\sum_{k=0}^{\sqrt{D}-1}(2(D-k)-1) = (2\sqrt{D}-1)D - 2\sqrt{D}+1$. Ainsi, les $N+1$ blocs ont au plus $V_2 = (N+1)((2\sqrt{D}-1)D - 2\sqrt{D}+1)$ tâches.

Le nombre de tâches restantes en dessous du bloc b est égal à $V_3 = 1 + \sum_{k=2}^{h_{min}}(2k-2) = h_{min}^2 - h_{min} + 1$.

Calculons maintenant l'instant où le processeur r sera le seul à avoir du travail. Le processeur r doit pour cela exécuter une quantité de travail supérieure au nombre maximal de tâches présentes sur chaque autre processeur qui est au plus : $V_3 + V_2 = h_{min}^2 - h_{min} + 1 + (N+1)((2\sqrt{D}-1)D - 2\sqrt{D}+1)D < h_{min}^2 + 2(N+1)\sqrt{D}D = f^2 * D + 2(N+1)\sqrt{D}D$.

Or dans un peigne du processeur r, la quantité de travail est supérieure à : $\frac{f^2 D}{2}$. Nous rappelons que f^2 est égal à $4\sqrt{D}$.

Ainsi, le processeur r aura dans chaque peigne une quantité de travail supérieure à $\sqrt{D}D$. Il lui suffit d'exécuter $N+2$ peignes pour qu'il soit le seul à avoir du travail.

L'algorithme d'ordonnancement doit donc alors transférer $\sqrt{D} - N - 2$ peignes pour éviter que le processeur r exécute un peigne sans que les autres n'aient du travail. Nous remarquerons que le processeur r ne peut pas exécuter un peigne ayant une quantité de travail supérieur à $\sqrt{D}D$ sans que aucun autre processeur n'exécute du travail. Dans le cas contraire, la condition sur le temps d'exécution n'est plus respectée.

De plus, le processeur s a introduit dans l'exécution des peignes au-dessus du bloc b, au moins $2N-2$ transferts (2 transferts entre 2 blocs). Le nombre de transferts est donc au moins $\sqrt{D} + N - 4$.

□

b Ordonnancement hors-ligne

Dans l'exemple précédent, la solution optimale découpe le travail en p parties équilibrées avec un nombre de transferts égal à $2p$. Pour cela, il suffit de calculer W/p et de découper les tâches grises en p blocs telle que le travail par processeur soit quasi équilibré comme l'illustre la figure 5.4. Chaque processeur avance en priorité sur la ligne de tâches grises de manière à activer le travail des autres le plus rapidement possible.

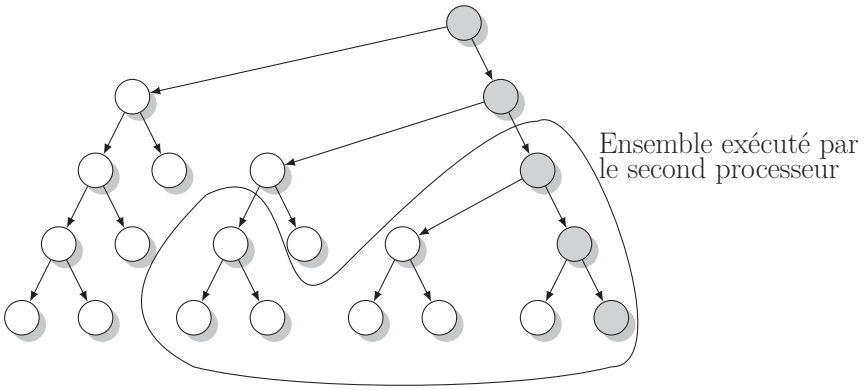


FIGURE 5.4 – Répartition efficace d'un ensemble de peignes sur deux processeurs avec un minimum de transferts.

c ratio de compétitivité sur le nombre de transferts

La solution hors-ligne optimale (pour minimiser le nombre de transferts) réalise p transferts. Avec le lemme 5.1, nous obtenons ainsi un ratio de compétitivité sur le nombre de transferts de $\frac{\sqrt{D}+o(\sqrt{D})}{p}$ Nous étendons maintenant ce résultat pour un ratio de $\frac{\sqrt{D}}{4} + o(\sqrt{D})$.

Théorème 5.2. *Nombre de transferts effectués par un algorithme en-ligne*

Tout algorithme d'ordonnancement de tâches en-ligne qui exécute le DAG illustré par la figure 5.5 sur p machines($p \geqslant 4$) en un temps inférieur à $\frac{W}{p} + \sqrt{D}D$, doit réaliser au moins $\frac{p}{4}\sqrt{D} - o(p\sqrt{D})$.

Démonstration. Nous négligeons ici les transferts réalisés sur l'arbre binaire au dessus des ensembles de peignes et aussi sa profondeur ($\log p \ll D$).

Sur les $\frac{p}{2}$ ensembles de peignes, nous appliquons le même raisonnement que dans le lemme 5.1. Nous distinguons deux cas, soit il y a plus de $\frac{p}{4}$ ensembles de peignes qui ont un bloc de taille supérieure à $\sqrt{D}$ dont la hauteur supérieure à $2\sqrt[4]{D}\sqrt{D}$, soit il y en a moins de $\frac{p}{4}$.

- Lorsqu'il y a moins de $\frac{p}{4}$ ensembles ayant la propriété, il y a donc plus de $\frac{p}{4}$ ensembles dont les tâches grises sont divisées en blocs de taille inférieure à $\sqrt{D}$. Dans ce cas sur plus de $\frac{p}{4}$ ensembles, le nombre de transferts est supérieur à $\sqrt{D} - o(\sqrt{D})$ transferts. Sur l'ensemble du DAG, le nombre de transferts est donc supérieur à $\frac{p}{4}\sqrt{D} - o(p\sqrt{D})$.

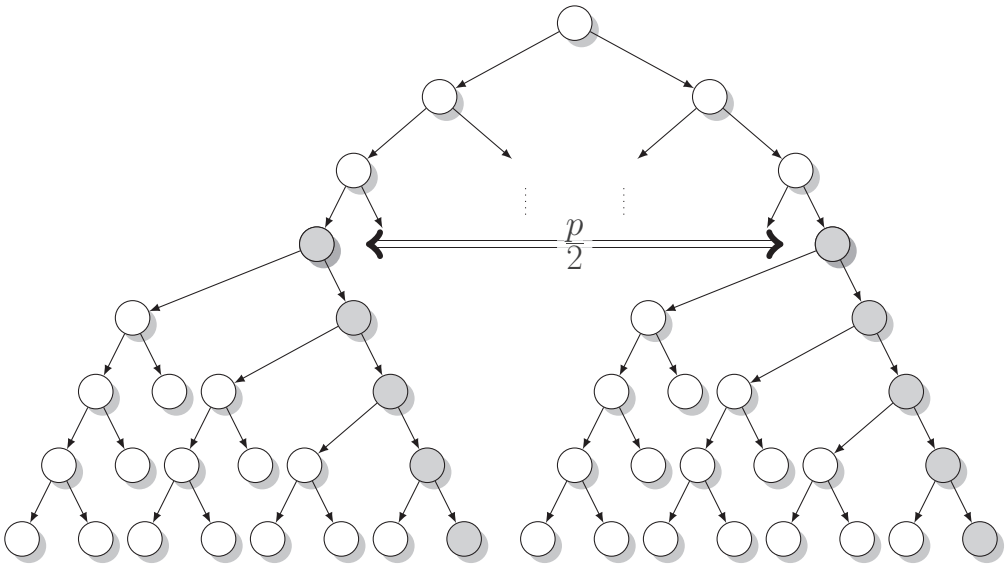


FIGURE 5.5 – Illustration du DAG exécuté sur p machines.

– Dans le cas où la majorité des ensembles ont un bloc de taille supérieure à $\sqrt{D}$, nous considérons un des ensembles ayant cette propriété pour montrer que lors de l'exécution de cet ensemble il y a $\sqrt{D} - o(\sqrt{D})$ transferts.
Notons e l'ensemble considéré, b le bloc de taille supérieure à $\sqrt{D}$ associé à e et r le processeur exécutant le bloc b.
Nous associons au processeur r un autre processeur s qui a soit aucune tâche localement, soit les tâches sous le bloc b de l'ensemble et des blocs de taille inférieure à $\sqrt{D}$ dont les tâches ont un temps d'exécution minimale.
Le processeur s existe car il y a au plus $\frac{p}{2}$ processeurs avec des tâches dont le temps d'exécution n'est pas égal au temps minimal.
Aucun processeur ne peut pas rester inactif un temps équivalent à l'exécution de p peignes du bloc b. Le temps d'exécution est égal au temps travaillé par les processeurs plus le temps inactif divisé par le nombre de processeurs : $T_p = \frac{T_{travail}+T_{inactif}}{p}$. Ainsi, si un processeur reste inactif durant p peignes du bloc b, le temps d'exécution sera supérieur à $\frac{T_t+p\sqrt{D}D}{p} = \frac{W}{p} + \sqrt{D}D$.
Le processeur s a localement l'équivalent de N blocs et les tâches en dessous du bloc b. L'obtention de ces tâches a généré au moins $N+1$ transferts. De plus, ces tâches permettrons au processeur s de rester actif au plus durant l'exécution de $N+1$ peignes par le processeur r.
Pour ne pas rester inactif durant l'exécution des autres peignes, il doit reprendre du travail, soit un peigne du processeur r soit commencer l'exécution d'un nouveau bloc.
Chaque peigne pris au processeur r ajoute un transfert (que ce soit le processeur s ou non).
Si le processeur s commence l'exécution d'un peigne, il sera exécuté en moins de temps que l'exécution d'un peigne par le processeur r. De plus, cette exécution génère un transfert aussi.
Si le processeur s commence l'exécution un nouveau bloc sur un ensemble, nous

laissons ce processeur exécuter ce travail. Nous changeons le processeur associé par un processeur qui n'est associé à aucun autre processeur exécutant un bloc de taille supérieure à $\sqrt{D}$ et qui a localement que des tâches unitaires (ou aucune tâche). Nous reprenons comme avec l'ancien processeur s.
Ainsi pour plus de $\sqrt{D} - p$ peignes exécuté par le processeur r, nous pouvons associer l'exécution d'un bloc qui a généré un transfert.
Pour chaque ensemble de peignes avec un bloc de taille supérieure à $\sqrt{D}$, il y a donc $\sqrt{D} - p$ transferts.
Afin d'obtenir un temps inférieur à $\frac{W}{p} + \sqrt{D}D$, l'ordonnancement en-ligne doit effectuer $\frac{p}{4}\sqrt{D} - o(p\sqrt{D})$ transferts.

□

5.1.2 WSCOM et les communications

Dans l'algorithme WSCOM, nous ajoutons des tâches virtuelles afin d'orienter les vols sur des parties indépendantes du DAG. Nous analysons ici l'impact des tâches virtuelles sur le nombre de transferts effectués.

Nous détaillons le nombre de transferts effectué lors de l'exécution d'un application ayant une structure de type «Fork-Join» illustré par la figure 5.6.

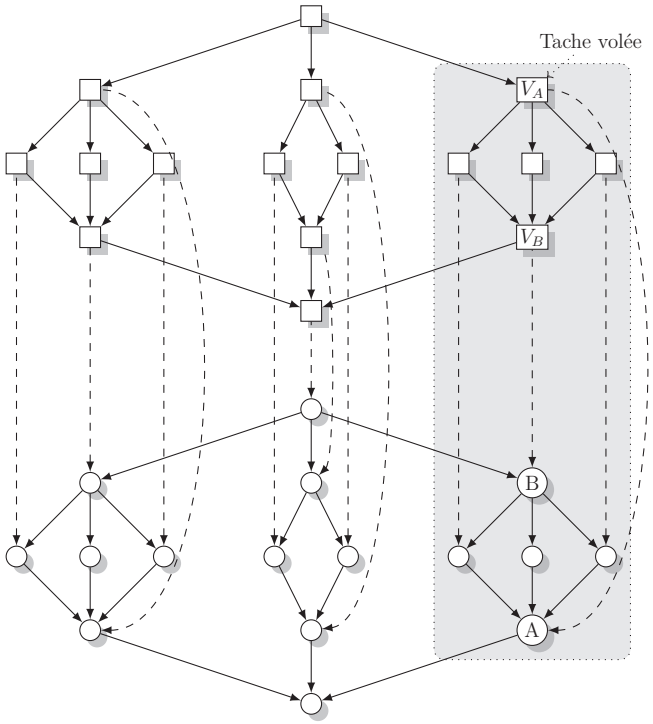


FIGURE 5.6 – DAG «Fork-Join» avec des tâches virtuelles.

Pour énumérer le nombre de transferts effectués par WSCOM lors de l'exécution d'un DAG «Fork-Join», nous détaillons dans un premier temps le nombre de transferts induit par le vol d'une tâche virtuelle. Nous considérons ensuite le nombre de transferts induit par le vol d'une tâche de calcul.

Nous supposons qu'un processeur vole une tâche virtuelle comme sur la figure 5.6. Sur la figure, le processeur ajoute dans sa pile la tâche V_A. Lors de son exécution, il ajoute dans sa pile les successeurs de la tâche V_A (dont la tâche de calcul A). Il exécute alors un des successeurs et ajoute la tâches virtuelle V_B. Il ajoute alors la tâches de calcul B. Si ce processeur ne reçoit pas de requête de vol, le nombre de données transférées est égal à 2 (un transfert pour la donnée lue par B et un pour la donnée créée par A). Cela est valable quelque soit la tâche virtuelle volée.

Les tâches A et B ne peuvent pas être séparées lors du vol d'une tâche virtuelle pour tout DAG «Fork-Join». Le nombre transferts est donc inférieur à deux pour chaque vol d'une tâche virtuelle.

Pour le vol d'une tâche de calcul, le nombre de transferts est inférieur à l'arité maximale A_r. Or, le nombre de vol réalisé est inférieur à $O(p \times D)$. Le nombre de transferts est donc inférieur à $O(A_r \times p \times D)$. Cette analyse prend en compte les arbres entrant, les DAG de type «Fork-Join».

Pour un arbre entrant, il y a au plus 1 transfert par vol de tâche virtuelles. La borne est donc la même que pour les DAG de type «Fork-Join».

Nous détaillons maintenant un exemple d'arbre sortant illustré par la figure 5.7. Lors du vol de la tâche virtuelle V_A, le processeur ajoutera dans sa pile les tâches en

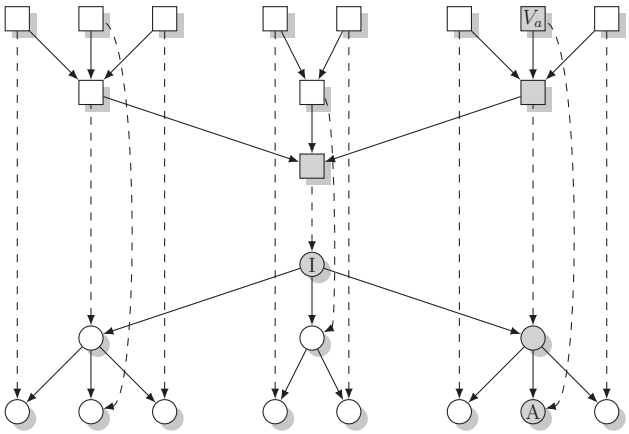


FIGURE 5.7 – Ajout des tâches virtuelles sur un arbre sortant.

gris sur la figure. Pour chaque tâche grise exécutée, le nombre de transferts sera de l'ordre de l'arité maximale moins 1. Par exemple, le vol de la tâche V_A engendre un nombre de transferts inférieur ou égal à $A_r \times D$ pour un seul vol. Dans le cadre d'une application décrite par un arbre sortant, cette heuristique n'est pas adaptée. En plus, si la tâche I a un temps d'exécution important, les autres processeurs volerons l'ensemble des tâches virtuelles. Le processeur qui a exécuté la tâche I aura alors peu de tâches à exécuter et devra terminer son exécution en volant des tâches une par une.

5.2 Analyse expérimentale de WSCOM

Nous avons développé WSCOM pour réduire l'impact des communications. Afin d'évaluer cet algorithme, nous avons choisi de nous orienter vers des simulations.

L'usage de simulations nous permet d'évaluer les performances de WSCOM en fonction du nombre de processeurs, de la quantité de données utilisées par l'application et des caractéristiques réseau.

De telles expériences auraient également pu être réalisées à l'aide d'outils d'émulation comme :

- KRASH [59] ou Wrekavoc [15] d'émuler des processeurs de puissance plus faible,
- P2PLab [54] permettant d'émuler un réseau avec des caractéristiques plus faible que le réseau réel.

Néanmoins, pour des raisons de rapidité et de facilité de mise en place, ainsi que de précision pour les algorithmes hors-ligne (absence de variations), nous avons préféré l'utilisation de simulations.

Dans un premier temps, la section 5.2.1 détaille l'implantation du simulateur, et les choix effectués pour réaliser les comparaisons. Puis, la section 5.2.2 présente une comparaison de l'algorithme WSCOM avec plusieurs algorithmes en-ligne et hors-ligne sur des DAG générés aléatoirement. Enfin, la section 5.2.3 introduit les résultats obtenus sur des DAG d'applications réelles.

5.2.1 Méthodologie

Notre simulateur est basé sur l'outil *SimGrid* [18]. *SimGrid* permet de simuler une plate-forme et les communications au niveau flux de données. La méthode utilisée pour modéliser les communication permet de simuler de larges plates-formes avec de la congestion du réseau [28]. Pour réaliser ces expériences, nous avons eu besoin d'une puissance de calcul importante. Nous avons donc réalisé l'ensemble des expériences de ce chapitre sur la plate-forme GRID'5000 [16].

Nous présentons dans un premier temps, l'implantation de WSCOM au sein de *SimGrid* et l'utilisation d'autres algorithmes d'ordonnancement hors-ligne. Puis, nous détaillons les plates-formes utilisées lors de la conduite des expériences. Enfin pour comparer les algorithmes d'ordonnancement, nous présentons les caractéristiques des applications utilisées.

Simulateur

Pour implanter notre simulateur, nous avons souhaité développer un outil générique permettant la modélisation de différents algorithmes basés sur le vol de travail.

Le simulateur a été développé avec le module «MSG» au sein de *SimGrid*. Ce module permet d'avoir une interface proche de celle de «*Message Passing Interface (MPI)*» pour interagir avec le noyau de simulation réseau.

Au début de l'exécution, le simulateur lit un fichier au format »dot» décrivant le DAG de l'application. La lecture du fichier est réalisée en utilisant la bibliothèque «Graphviz» [29]. Une fois l'application chargée, nous plaçons les tâches sur les machines. Nous avons implanté plusieurs placements initiaux des tâches : sur une machine, aléatoire, round-robin, ou fourni dans le DAG.

Dans cet algorithme de vol de travail classique, les tâches sont ajoutées à la pile du processeur qui les active. Pour réaliser cet ajout sur une plate-forme distribuée, nous avons choisi d'utiliser une Distributed Hash Table (DHT) qui donne le droit d'ajouter la tâche au processeur la rendant prête. L'algorithme de vol de travail est conforme à celui détaillé initialement par Blumofe et Leiserson [31].

Puisque la tâche est ajoutée dans la pile lors de son activation, il est difficile de prévoir son placement avant son exécution pour le vol de travail classique. Les transferts de données sont donc effectués au moment de l'exécution de la tâche.

Pour implanter les variantes de WSCOM, nous avons ajouté des tâches virtuelles. Lorsque ces tâches existent, l'ajout des tâches de calcul dans la pile est réalisé au moment de l'exécution de la tâche virtuelle associée. Dans cas, nous utilisons aussi une DHT dont le fonctionnement est décrit dans la section 4.3.1.

Pour la réalisation des transferts de données, un processus sur la machine se charge de faire les envois et les réceptions. Cela nous a permis d'envoyer des données de deux manières différentes :

– au moment où la tâche est exécutée (WSCOM).
– dès la production des données (WSCOM$_{pf}$)

Le DAG chargé au début de l'exécution contient les temps d'exécution des tâches et les quantités de données à transférer. Bien entendu, aucune de ces informations n'est prise en compte par le vol de travail et les variantes de WSCOM. En revanche, ces informations sont fournies à *SimGrid* pour simuler l'exécution des tâches et des transferts de données.

Pour comparer nos algorithmes aux algorithmes hors-ligne, nous avons ajouté au sein de *SimGrid* un moyen d'interpréter les informations contenus dans les fichiers «dot». Cela nous a permis d'utiliser des algorithmes d'ordonnancement hors-ligne développés avec *SimGrid*.

Plate-formes simulées

La plate-forme dans *SimGrid* est décrite par un fichier «xml» [12]. Les informations contenues dans ce fichier détaillent la structure du réseau (la latence et le débit) et la puissance des machines simulées.

Pour nos expériences, nous avons considéré deux types de plates-formes.

La première structure est en forme de clique. Le graphe associé est complet. Sur ce type de plate-forme, il n'y a pas de congestion du réseau : Aucun lien n'est partagé entre deux machines. Lors de l'ordonnancement des tâches avec un algorithme hors-ligne, les temps de communication de l'application sont connus exactement. Ce type de structure réseau n'est pas réaliste. Pour obtenir une telle plate-forme, les machines nécessiteraient un nombre de cartes réseaux égal au nombre de machines présentes.

La seconde structure est identique à un ensemble de machines connectées par un commutateur («switch»). Cette structure est généralement utilisée au sein des petites grappes de calcul. Le graphe correspondant est une étoile. Sur cette plate-forme, la congestion peut se produire au niveau des liens de communications entre le commutateur

et une machine. Pour cela, il suffit que deux machines envoient d'importante quantité de données à une même autre machine.

Pour les simulations réalisées, nous avons utilisé un nombre de machines variant de 1 à 50. La plate-forme est homogène, les machines ont donc la même puissance.

Ces plates-formes sont simples mais elles permettent de considérer un cas sans congestion et un cas avec de la congestion. La figure 5.8 illustre les deux structures pour 5 machines.

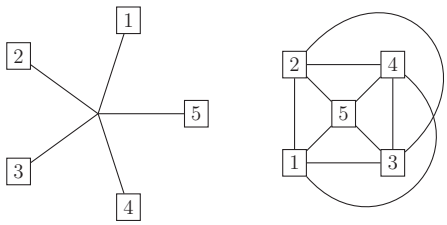


FIGURE 5.8 – Représentation d'une clique et d'une grappe de calcul avec 5 machines.

Les performances des liens de communications sont définies par la latence et le débit. La latence est choisie égale à 0,1 milliseconde. Le débit est quant à lui égal à 1 Gbit par seconde.

DAG représentant une application

Les DAG représentant les applications entrées de notre outil basé sur *SimGrid* peuvent introduire des biais dans l'analyse des algorithmes d'ordonnancement. Certains ordonnancement sont sensible à certains paramètres comme l'étude [62] le montre.

Dans notre étude, nous utilisons différents types de DAG. En premier, nous utilisons des DAG générés aléatoirement, avec plusieurs méthodes de génération. Puis, nous utilisons des DAG extraits d'exécutions réelles.

Les DAG générés aléatoirement proviennent de l'outil *Graph-GENerator (GGEN)*[22]. Cet outil comporte différentes méthodes de générations classiques de DAG aléatoire. En utilisant plusieurs algorithmes de génération, nous espérons limiter les biais présents lors de la comparaison.

Nous avons choisi deux méthodes classiques de génération : «TGFF» [26] et «layer-by-layer» [82].

Les DAG générés comportent un nombre de nœuds proche de 500. Une quantité de travail est associée à chacun de ces nœuds. La répartition du travail est choisie uniformément dans un intervalle allant de 7 à 25 secondes.

En plus de cette quantité de travail, chaque arête a une quantité de données à transférer. Ces quantités varient uniformément entre 0 et une valeur maximale. La valeur maximale change en fonction de la série d'expériences réalisée. Nous avons utilisé

au plus un valeur égale à 2 Gigabits. Dans nos exemples, la quantité moyenne de travail reste plus grande que le temps de communication.

Puis dans une second analyse, nous utilisons également des DAG provenant d'exécutions réelles. Pour obtenir ces DAG, nous avons modifié l'outil Make afin de mesurer le temps d'exécution de chaque commande et de connaître les dépendances avec les fichiers et les autres commandes. Cette méthode nous permet d'extraire les DAG des Makefile sans modifier le comportement de Make.

Nous nous sommes basés sur le gestionnaire de paquets source MacPort qui comportent de nombreux paquets. De ces paquets, nous avons récupéré environ 500 DAG. L'exécution de ces compilations a été réalisée sur un processeur Intel Xeon X5472 avec un fréquence de 3GHz et 6 Go de RAM. Les caractéristiques des DAG en moyenne sont détaillées dans le tableau suivant :

Nombre de nœuds	Nombre d'arêtes	W	D	Quantité de données
750	1500	300 sec	22 sec	400 Mo

Algorithmes d'ordonnancement

Dans la comparaison, nous souhaitons mettre en évidence l'avantage obtenu en utilisant la structure du DAG. Pour cela, nous comparons WSCOM à plusieurs algorithmes en-ligne.

De plus, nous souhaitons montrer que la réduction de l'impact des communications nous permet d'avoir de bonnes performances. Nous comparons ainsi WSCOM aux algorithmes hors-ligne qui auront toute l'information nécessaire pour réduire l'impact des communications (hormis l'impact de la congestion du réseau).

Nous nous comparons au mécanisme de vol de travail classique afin de mettre en évidence l'impact de l'utilisation de la structure sur l'exécution. Voici une description des algorithmes de vol travail utilisés comme base de comparaison :

Work-Stealing (WS) : Au début de l'exécution, toutes les tâches sont placées dans la pile d'un processeur. De plus lors d'un vol, le voleur prend une des tâches les plus anciennes de la pile.

WS_HALF : le positionnement des tâches au début de l'exécution est identique à celui de WS. En revanche, lors d'un vol, le voleur prend la moitié des tâches les plus anciennes dans la pile de a victime.

WS_RR : Au début de l'exécution, toutes les tâches prêtes sont placées sur les machines avec une répartition en «Round-Robin». Lors d'un vol, le voleur prend une des tâches les plus anciennes dans la pile.

WS_RRHALF : Lors d'un vol, la moitié des tâches ayant la plus faible profondeur sont volées. Le placement des tâches est identique à celui de l'algorithme WS_RR.

WS_{convex} : Cet algorithme est basé sur celui détaillé section 2.2.1 (page : 29). Avant l'exécution, une décomposition du DAG en regroupements convexes est réalisé. Des tâches virtuelles sont ajoutées au DAG pour diriger les vols vers un groupe convexe de tâches.

Les variantes du vol de travail (WS,WS_HALF,WS_RR et WS_RRHALF) peuvent être impactées par l'ordre des tâches dans la pile du processeur. Pour obtenir les meilleures performances avec ces algorithmes, nous avons choisi un ordre topologique. Cet ordre permet d'avoir les tâches proches dans le DAG proche dans la pile.

L'heuristique WS_{convex} est quant à elle limitée en performance par le fait que lors de la découpe, les temps d'exécution des tâches ne sont pas connus. De plus, le temps de calcul de la découpe est parfois important en fonction de la structure du DAG. En particulier, ce temps de découpe nous a empêché de tester cet algorithme sur les DAG extraits de Makefile.

Dans la comparaison avec les algorithmes hors-ligne, nous souhaitons principalement montrer que notre approche fournit des performances comparables. Parmi les algorithmes hors-ligne, nous avons choisi 8 heuristiques : Heterogeneous Earliest Finish Time (HEFT) [83], CPOP [83], MinMin [50], MaxMin [50], Sufferage [50], BIL [55], HBMCT [66] et PCT [51]. Ces algorithmes font partie des algorithmes de liste présentés dans la section 2.1.

5.2.2 Ordonnancement de DAG générés aléatoirement

Dans ces expériences, nous souhaitons dans un premier temps comparer les variantes de WSCOM pour savoir si l'utilisation de la DHT donne de meilleures performances que l'utilisation d'un arbre couvrant. Ces variantes ont été introduites dans la section 4.3.1. Cette comparaison est présentée section a sur les DAG générés avec la méthodes «layer-by-layer».

La seule variation de performances en passant d'une méthode de génération à l'autre est présente sur la variante $WSCOM_{tree}$.

Les expériences présentées dans les sections b et c détaillent les résultats obtenus avec la méthode de génération «TGFF».

Après avoir observé les différences entre les variantes de WSCOM, nous souhaitons mettre en avant les sur-coûts dûs à WSCOM par rapport au vol de travail classique mais aussi les gains. La section b présente une comparaison entre WSCOM et les variantes du vol de travail présentées section 5.2.1.

Enfin dans les variantes de WSCOM, nous avons ajouté un mécanisme permettant l'envoi des données dès leur production. Nous tentons de mettre en avant les performances obtenues. Pour cela, nous souhaitons donc comparer aux performances qu'il est possible d'obtenir avec les algorithmes hors-lignes. Ces algorithmes nous fournissent une performance proche de celle optimale atteignable en pratique. La section c compare l'algorithme WSCOM en-ligne avec des ordonnancements hors-ligne.

a Comparaison de WSCOM et $WSCOM_{tree}$

Lors de la présentation des algorithmes, nous avons introduit plusieurs variantes de WSCOM dont $WSCOM_{tree}$ qui ne fait pas usage d'une table de hachage distribuée.

Cette variante choisit avant l'exécution un arbre couvrant pour définir la création des tâches virtuelles.

Sur les DAG générés aléatoirement avec «TGFF», WSCOM_{tree} a des performances similaires à l'algorithme WSCOM. Nous ne présentons pas ici les résultats de ces expériences. Des différences de performance sont obtenues sur les DAG générés aléatoirement avec l'algorithme «layer-by-layer». Cette différence provient principalement de l'arité maximale du DAG qui est plus élevée. Le nombre d'arbres couvrants différents est donc plus important. Dans ce cas, le choix de l'arbre couvrant peut avoir un impact.

Dans cette série d'expériences, nous considérons des DAG générés avec l'algorithme «layer-by-layer» ayant une faible quantité de communication (inférieure à 100 Mb). L'exécution est réalisée sur une plate-forme avec une structure en clique.

La figure 5.9 montre l'évolution du temps d'exécution en fonction du nombre de machines. Nous précisons que la courbe de droite présente un agrandissement d'une sous-partie de la courbe de gauche. WSCOM obtient de meilleures performances. Cette amélioration de performance est due à l'utilisation de la DHT qui permet de reporter le choix effectué par l'arbre couvrant au moment de l'exécution. Nous pensons qu'en réalisant ce choix au cours de l'exécution WSCOM évite certains vols.

Dans les expériences suivantes, nous utilisons donc les versions de WSCOM utilisant la DHT.

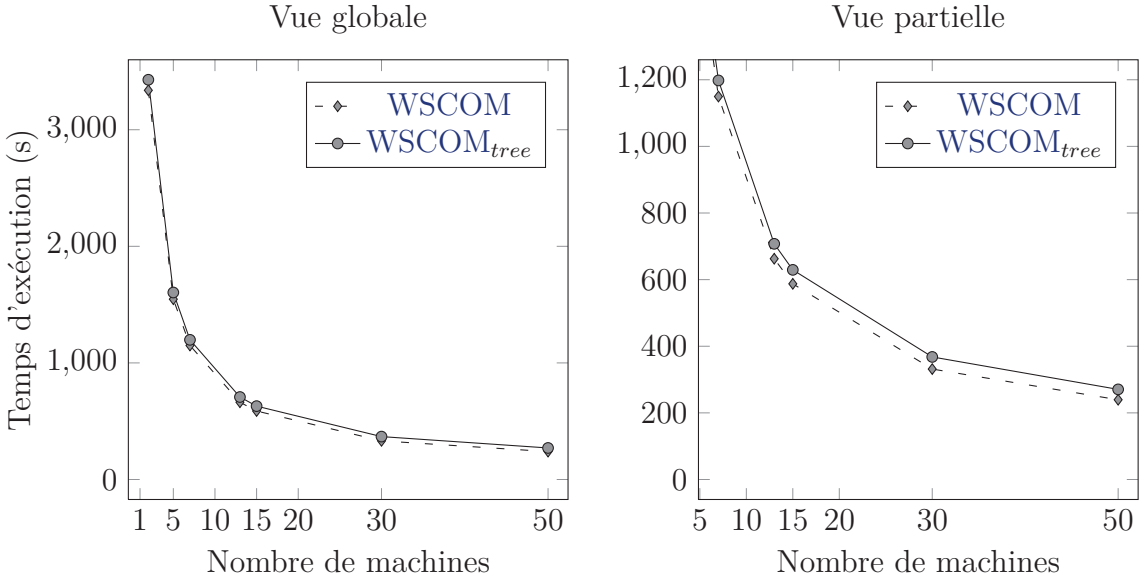


FIGURE 5.9 – Ordonnancement de DAG aléatoires générés avec par l'algorithme «layer-by-layer» avec des données volumineuses.

b Comparaison avec des ordonnancements en-ligne

Nous souhaitons mettre en avant ici les gains liés à WSCOM. WSCOM tente d'améliorer les performances en utilisant la connaissance de la structure du DAG de l'application.

Lorsqu'il y a peu de données, le mécanisme interne de WSCOM pourrait ajouter un sur-coût. Nous ne présentons pas ici les courbes avec peu de données, mais la différence entre le vol de travail classique et WSCOM est négligeable (inférieure à 2%).

Nous comparons donc dans un premier temps les différents algorithmes en-ligne lorsqu'il y a beaucoup de communications.

Ensuite, nous analysons les performances des différentes variantes de WSCOM et nous les comparons à l'algorithme WS_{convex} ainsi qu'à la meilleure variante du vol de travail classique. Ces variantes ont été présentées section 5.2.1.

Comparaison des ordonnancements par vol de travail

La figure 5.10 montre l'évolution du temps d'exécution en fonction du nombre de machines. La figure de gauche est la vue globale tandis que celle de droite met en avant la partie de la courbe avec un grand nombre de machines. Cette figure compare ainsi les heuristiques sur les DAG générés aléatoirement. Pour ces expériences, le volume de données maximal est égal à 2 Gb et la plate-forme utilisée est une structure en étoile. Les communications impactent fortement les temps d'exécution d'autant plus qu'il y a potentiellement de la congestion sur les liens.

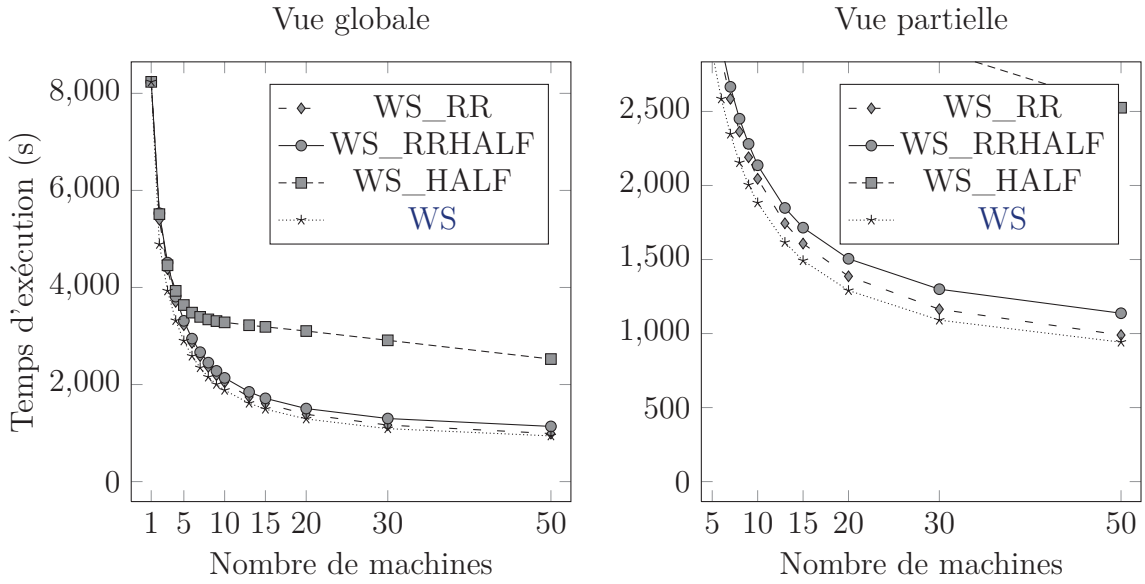


FIGURE 5.10 – DAG aléatoires ayant des données volumineuses ordonnancés par vol de travail sur une grappe.

Les performances de WS_HALF sont largement inférieures à celles des autre algorithmes.

Sur cette figure, nous pouvons remarquer que les heuristiques réalisant le vol d'une tâche s'en sortent mieux. Ce résultat laisse penser que le vol d'une tâche est plus approprié lorsque les communications sont coûteuses. Pour appuyer ce constat, nous donnons une intuition : le vol inapproprié d'une tâche pénalisera moins que le vol d'un ensemble de tâches avec plusieurs tâches inappropriées.

Notons également que, dans ces expériences, l'algorithme WS_RR ne profite pas de l'avantage du placement initial. Celui-ci peut, en effet, placer des tâches qui ont beaucoup d'affinité sur deux processeurs différents.

Comparaison de WSCOM avec le meilleur ordonnancement par vol de travail

Dans cette partie, nous comparons nos algorithmes à celui fournissant la meilleure performance en moyenne sur le jeu de données considéré qui est le vol de travail classique. Nous ne nous comparons pas à l'algorithme qui prend pour chaque DAG le meilleur algorithme car une telle caractérisation avant l'exécution n'est pas réalisable.

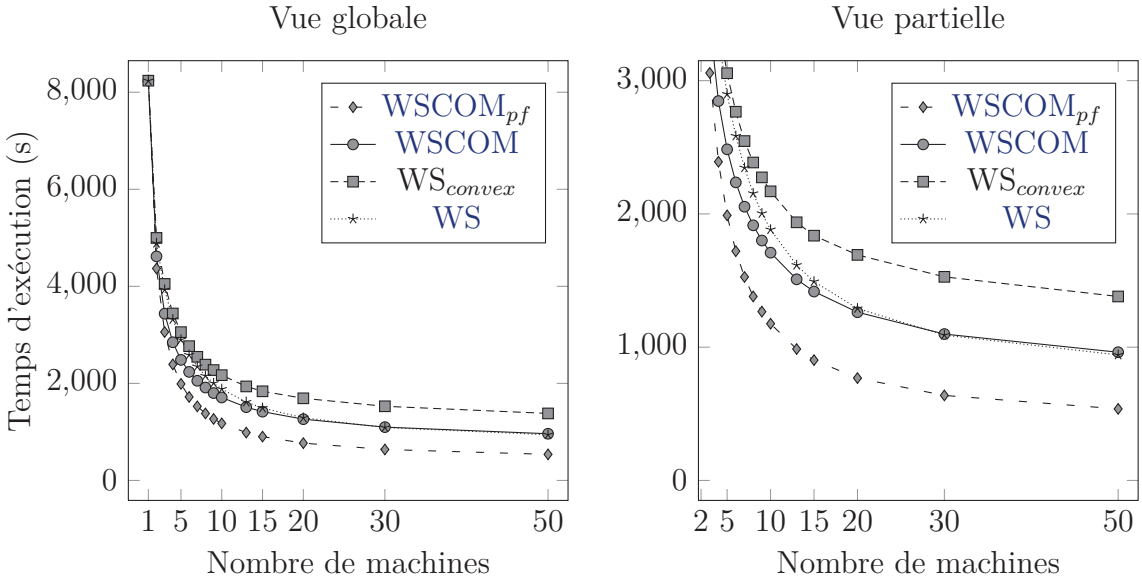


FIGURE 5.11 – Ordonnancement de DAG aléatoires ayant des données volumineuses avec WSCOM, WS$_{convex}$ et WS.

La figure 5.11 illustre l'évolution du temps d'exécution en fonction du nombre de machines. Cette figure compare nos algorithmes avec l'algorithme de vol de travail classique. WSCOM fournit de meilleures performances que les autres algorithmes. Le gain est de plus de 15% lorsque le nombre de machines est compris entre 2 et 13. Ce gain est explicable en comparant les quantités de données transférées. La figure 5.12 montre que WSCOM réduit significativement cette quantité lorsque le nombre de machines est faible. Notons au passage qu'avec la génération uniforme des quantités de données sur les graphes aléatoires, quantité de données et nombre de transferts sont directement liés.

WS$_{convex}$ réduit lui aussi la quantité de données transférées mais l'algorithme contraint fortement l'exécution. Comme nous l'avons détaillé section 4.2.2, nous rappelons que WS$_{convex}$ oblige d'exécuter l'ensemble des tâches présentes dans les groupes prédécesseurs avant de pouvoir exécuter des tâches d'un groupe. Ce mécanisme ajoute des synchronisations et augmente virtuellement le chemin critique de l'application.

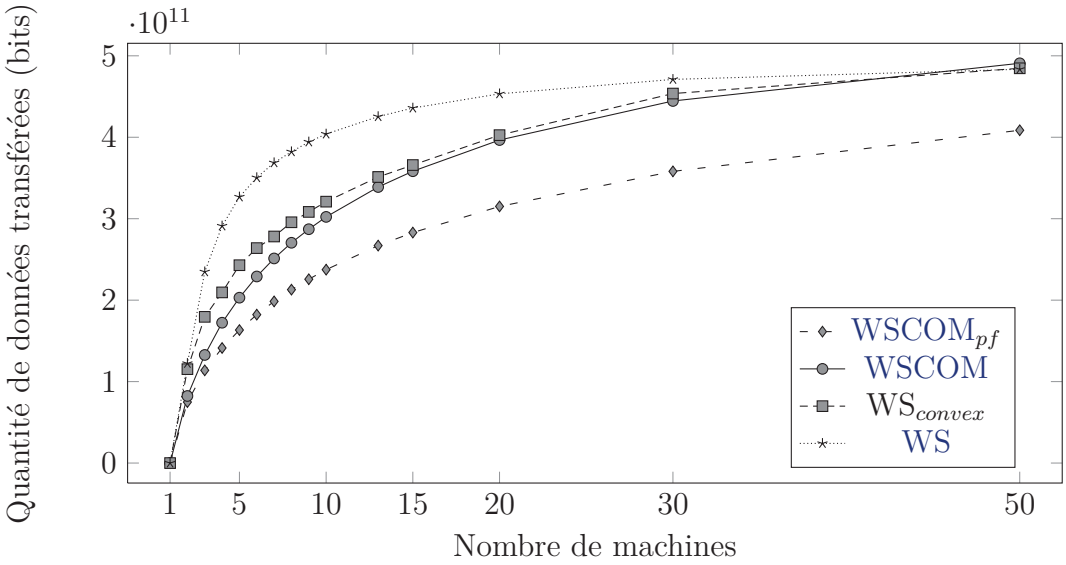


FIGURE 5.12 – Quantité de données transférées durant l'ordonnancement de DAG aléatoires ayant des données volumineuses avec WSCOM, WS_{convex} et WS.

Pour l'heuristique WS_{convex}, l'ordonnancement ne profite pas pleinement du découpage réalisé auparavant sur le DAG. Nous pensons que cette heuristique est sensible au nombre de découpes récursives réalisées. Afin de voir l'impact de ce nombre sur le temps d'exécution, il pourrait être intéressant de faire évoluer ce paramètre.

En plus de réduire les quantités de données communiquées, WSCOM permet d'envoyer les données dès leurs productions. En activant l'envoi des données en avance, WSCOM réduit considérablement le temps d'exécution avec l'heuristique $WSCOM_{pf}$.

Cette comparaison montre que WSCOM a des performances meilleures que les heuristiques classiques du vol de travail. En plus, cet algorithme limite le nombre de communications réalisées lorsque le nombre de machines est faible. Dans le cas contraire, il privilégie l'équilibrage de la charge. Enfin, $WSCOM_{pf}$ permet de montrer que sur les DAG générés aléatoirement, la possibilité d'envoyer les données dès leur production améliore nettement les performances.

c Comparaison avec des ordonnancements hors-ligne

Avant de comparer notre algorithme aux différentes heuristiques hors-ligne, nous commençons par analyser leur comportement sur les DAG générés aléatoirement.

Puis, nous comparons WSCOM, $WSCOM_{pf}$ et la meilleure heuristique hors-ligne lorsqu'il n'y a pas ou peu de congestion du réseau. Ce cadre d'expérience correspond au modèle de communication utilisé par les algorithmes hors-ligne.

Enfin, nous comparons ces algorithmes dans le cas où il y a de la congestion du réseau. Ces expériences nous permettront de mettre en évidence le gain possible dans des conditions plus réalistes.

Comparaison des ordonnancements hors-ligne

La figure 5.13 montre l'évolution du temps d'exécution en fonction du nombre de processeurs pour quelques algorithmes hors-ligne. Les arêtes du DAG ont au plus 2 Gigabits de données. La structure de la plate-forme considérée est une clique. Les différences de temps d'exécution entre les algorithmes sont faibles malgré la quantité de données transférées. Pour obtenir une lecture simplifiée de la courbe, nous avons affiché uniquement certaines heuristiques. L'ensemble de ces heuristiques a été détaillé dans la section 5.2.1, et le fonctionnement de cette famille d'algorithmes a été présenté dans la section 2.1. Dans un soucis de clarté, nous ne présentons pas ici les résultats pour les heuristiques PCT, MinMax, MaxMin, Sufferage et HBMCT. Pour information, PCT obtient des performances équivalente à HEFT, et les autres heuristiques ont des performances semblables à BIL.

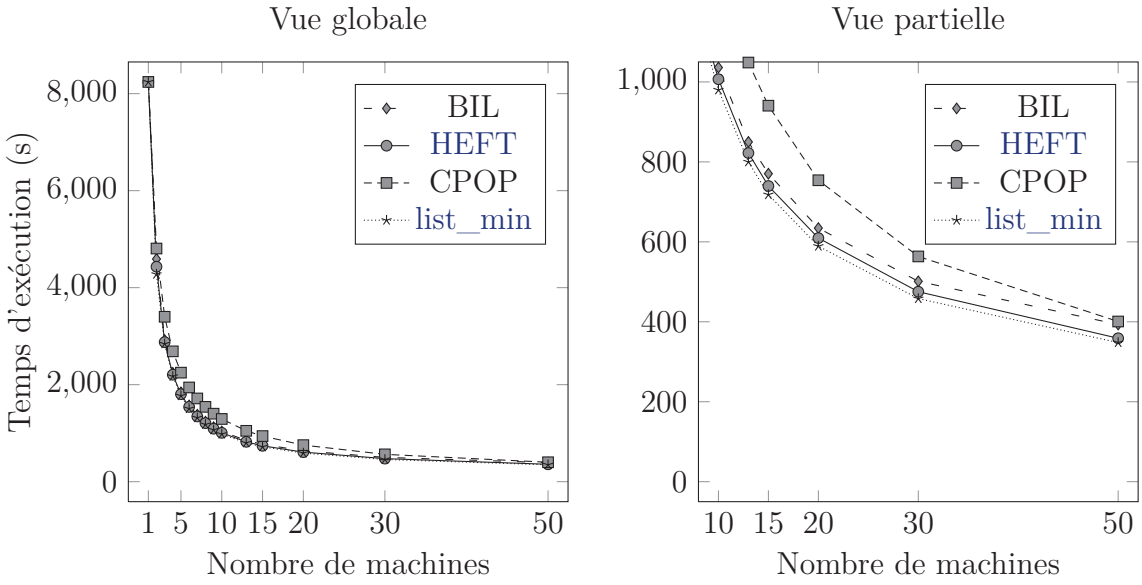


FIGURE 5.13 – Ordonnancement en-ligne de DAG aléatoires ayant des données volumineuses.

Tous les algorithmes hors-ligne ont un comportement similaire sur les DAG générés aléatoirement. Le temps d'exécution tend vers une limite non nulle lorsque le nombre de processeurs augmente. Afin de simplifier les courbes suivantes, nous comparons nos algorithmes à l'heuristique list_min. Pour obtenir cette heuristique, l'algorithme hors-ligne génère l'ordonnancement à partir de chaque algorithme listé précédemment. Parmi les ordonnancements obtenus, il choisi celui ayant le temps d'exécution le plus faible.

Dans toutes les expériences, les algorithmes d'ordonnancement hors-ligne connaissent les temps d'exécution des tâches et de communication exacts. Ils ne prennent seulement pas en compte la congestion du réseau qui n'a pas d'impact sur la plate-forme ayant une structure en clique. Ainsi, les algorithmes hors-ligne peuvent ordonnancer les tâches en sachant exactement le comportement de l'exécution

Nous commençons par détailler les résultats pour des applications travaillant avec des données peu volumineuses.

Comparaison de WSCOM, WSCOM$_{pf}$ et list_min lorsque les temps de communication sont faibles (100 Mb)

Nous considérons dans cette partie des applications générant des transferts de données faibles. Dans ce cas, les résultats sont identiques quelle que soit la plate-forme utilisée : clique ou étoile. Lorsque les quantités de données sont faibles, il n'y a pas de congestion sur les liens réseaux.

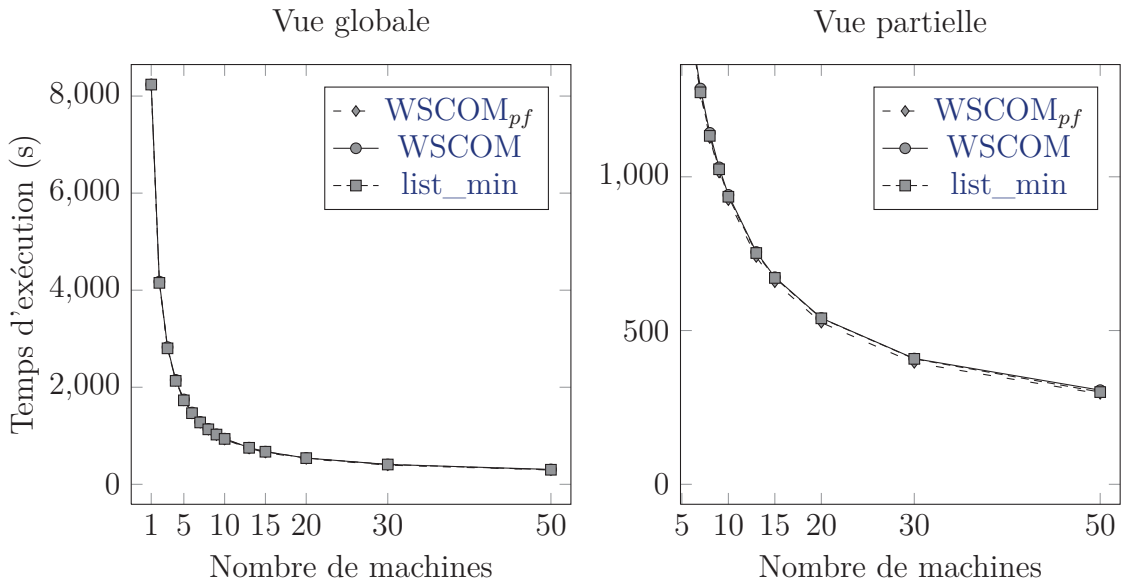


FIGURE 5.14 – Comparaison des heuristiques list_min, WSCOM et WSCOM$_{pf}$ sur DAG aléatoires ayant des données peu volumineuses.

La figure 5.14 montre l'évolution du temps d'exécution en fonction du nombre de processeurs. Les différents algorithmes d'ordonnancement obtiennent des performances similaires. Dans ce cas, le comportement attendu de WSCOM est proche de celui d'un algorithme de liste distribué comme le vol de travail. Il est donc normal que les performances soient proches lorsque les données sont peu volumineuses.

Sur cette figure, nous pouvons aussi constater que WSCOM$_{pf}$ obtient des temps d'exécution faibles lui aussi. Pourtant cet algorithme empêche les vols des tâches de calculs. Ceci peut introduire des lacunes dans l'équilibrage de la charge. Dans le cas de DAG générés aléatoirement, cette limitation a néanmoins peu d'impact sur le temps d'exécution.

Nous pouvons conclure de cette série d'expériences, que malgré e'absence d'information sur les temps d'exécution et du chemin critique, les algorithmes en-ligne obtiennent des performances très proches de celles des algorithmes hors-ligne.

Comparaison de WSCOM, WSCOM$_{pf}$ et list_min avec des temps de communication importants (2 Gb) et sans congestion

Lorsque le volume de communication augmente, la congestion sur les liens partagés commence à apparaître. Pour montrer l'impact de la congestion sur les liens, nous avons

pour ces expériences utilisé les deux types de structure réseau : clique et étoile.

Dans le cas où il n'y a pas de congestion, les algorithmes d'ordonnancement par liste calculent le temps réel des communications. Ils peuvent donc réduire au maximum l'impact des communications. En revanche, sur une plate-forme de type étoile, la congestion a un impact sur les temps de communication et diminue ainsi les performances de ces algorithmes.

La figure 5.15 présente une comparaison de l'heuristique list_min, WSCOM et WSCOM$_{pf}$. Dans cette expérience, la structure de la plate-forme est une clique, et le volume de données maximal est de 2 Gigabits. Dans le cadre de ces expériences, WSCOM

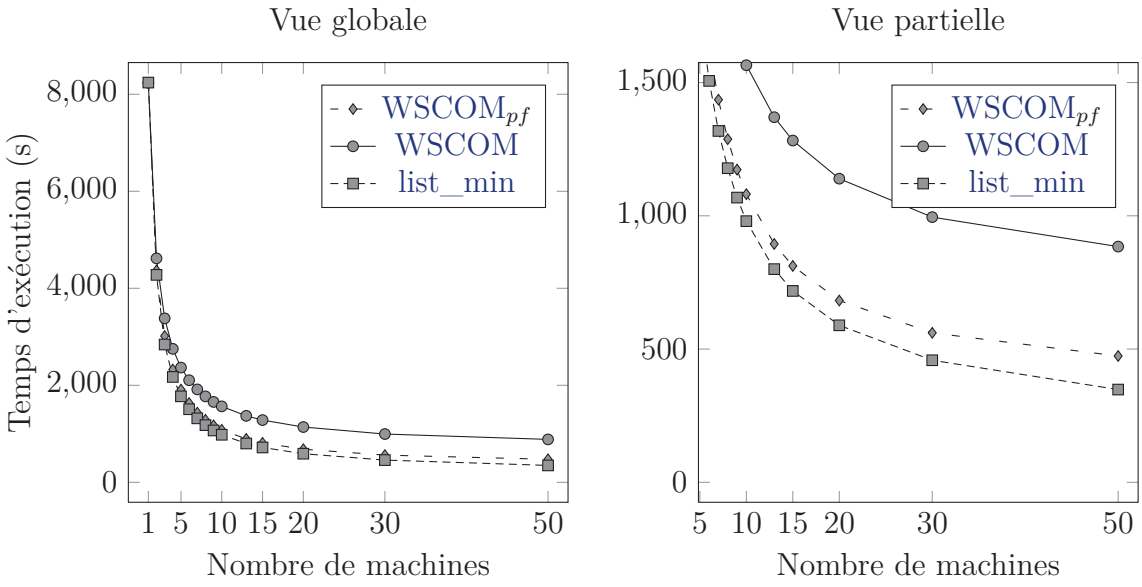


FIGURE 5.15 – Comparaison des heuristiques list_min, wscom et wscom$_{pf}$ sur dag aléatoires ayant des données volumineuses avec une plate-forme sans congestion.

ordonnance les tâches avec un temps d'exécution significativement plus important. Cette perte de performance provient du fait que WSCOM attend le début d'exécution d'une tâche pour envoyer les données. De plus, le processeur demande l'envoi des données les unes après les autres. Il ne peut donc pas recouvrir les communications par l'exécution d'autres tâches.

En revanche, WSCOM$_{pf}$ envoie les données dès leurs productions. Cela lui permet de recouvrir potentiellement certains communications par l'exécution d'autres tâches. Ainsi, il obtient des performances proches de list_min.

Les algorithmes en-ligne n'ont pas de meilleures performances que les algorithme hors-ligne, car ces derniers connaissent exactement ce qu'il va se passer au cours de l'exécution. Dans ces conditions de comparaisons, il est remarquable que WSCOM$_{pf}$ ait des performances aussi proches de celles des algorithmes hors-ligne sans connaître les temps de transferts. Ainsi, ces résultats montrent que l'utilisation de la symétrie du DAG est une méthode intéressante pour réduire la quantité de données transférées par un ordonnancement en-ligne.

Comparaison de WSCOM, WSCOM$_{pf}$ et list_min avec des temps de communication importants (2 Gb) et avec congestion

Dans cette partie, nous nous intéressons aux performances obtenues sur une plate-forme de type grappe de calcul. La figure 5.16 montre l'évolution du temps d'exécution en fonction du nombre de machines sur une telle plate-forme. La quantité de données maximale associée à une arête est de 2 Gigabits. Sur cette plate-forme, la congestion est présente. Les coûts de communications peuvent être plus importants que ceux calculés par les algorithmes hors-ligne.

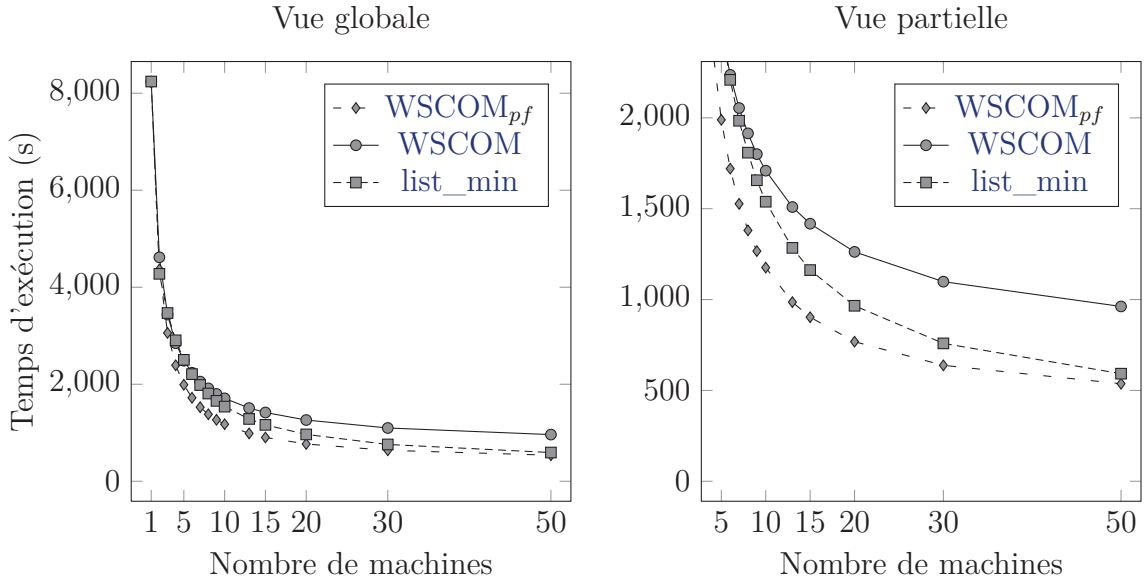


FIGURE 5.16 – Comparaison des heuristiques list_min, wscom et wscom$_{pf}$ sur dag aléatoires ayant des données volumineuses avec une plate-forme de type grappe.

L'apparition de la congestion du réseau explique la baisse de performances des algorithmes hors-ligne.

Il est intéressant de remarquer que WSCOM$_{pf}$ obtient de meilleures performances que les algorithmes hors-ligne. Cette amélioration de performance est significative et surprenante pour un algorithme en-ligne. Nous analysons donc plus en détail ce point.

Pour comprendre ce gain, nous nous intéressons à la quantité de données transferées durant l'exécution. La figure 5.17 montre l'évolution de cette quantité en fonction du nombre de machines.

Cette figure montre que WSCOM et WSCOM$_{pf}$ réalisent un nombre de communications significativement inférieur au nombre obtenu par list_min.

Ces résultats valident notre approche initiale qui a été de considérer le problème d'ordonnancement comme un problème bi-objectif. L'objectif secondaire de la réduction du nombre de communications impacte bien le temps d'exécution.

La différence est significative pour un nombre de machines compris entre 10 et 20. C'est dans cette plage de valeur que WSCOM a le plus de marge pour réduire les communications sans affecter l'équilibrage de la charge. Ainsi, cela explique que la différence de temps d'exécution sur la figure 5.16 est plus importante dans cet intervalle.

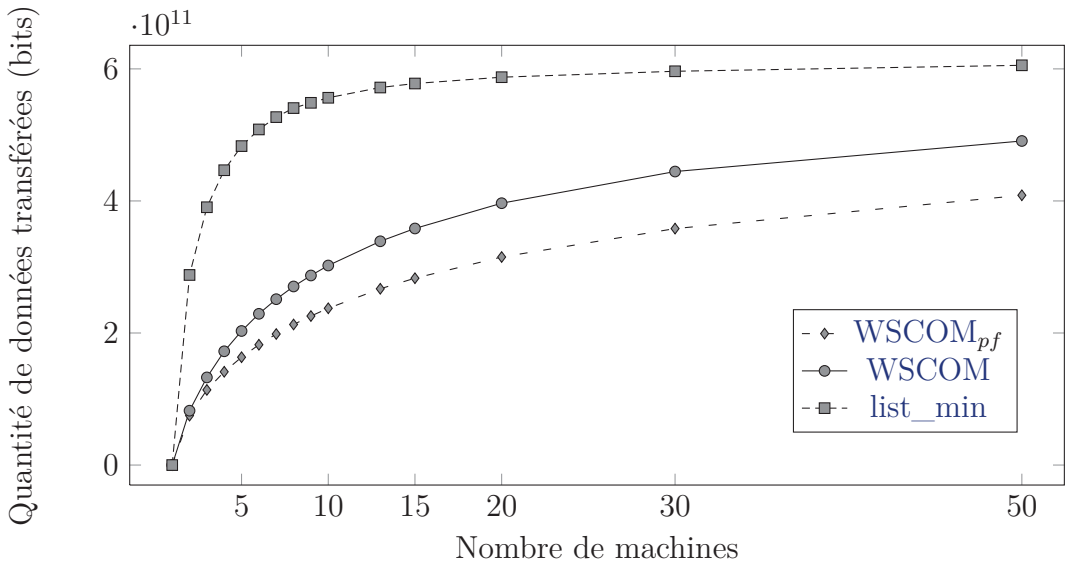


FIGURE 5.17 – Quantité de données transférées par les heuristiques list_min, WSCOM et WSCOM$_{pf}$ en ordonnançant des DAG aléatoires sur une plate-forme de type grappe.

La différence se réduit lorsque le nombre de machines augmente. Nous pensons que la réduction de l'écart est due au fait que WSCOM tente de conserver l'équilibrage de la charge en priorité.

Nous concluons de ce jeu d'expériences que l'utilisation de la symétrie du DAG semble être une solution pertinente pour les DAG considérés. De plus, l'envoi des données dès leur production est une stratégie intéressante et performante. En effet sur les DAG générés aléatoirement, WSCOM permet une réduction du temps d'exécution d'environ 15% par rapport au vol de travail classique. Ce gain de performance provient d'une réduction de la quantité de données transférées.

De plus en permettant à WSCOM d'envoyer les données au plus tôt, WSCOM$_{pf}$ obtient des performances comparables à celles des algorithmes hors-ligne qui ont accès au temps d'exécution et de communication exact sans congestion du réseau.

En fin lorsqu'il y a de la congestion sur le réseau de communication, le temps d'exécution avec l'algorithme WSCOM$_{pf}$ est inférieur à celui obtenu avec les algorithmes hors-ligne.

5.2.3 Ordonnancement de DAG extraits de Makefile

Après avoir comparé le comportement des algorithmes sur des DAG générés aléatoirement, nous analysons les performances obtenues sur des DAG extraits de Makefiles. Ces DAG ont été générés lors de la compilation en séquentiel de l'ensemble des paquets présents dans le gestionnaire de paquets MacPort dont une documentation est présente dans le chapitre 13 du livre [69]. Les temps d'exécution des tâches ont été mesurés directement dans l'outil Make et les volumes de données associé aux arêtes sont déduits de la taille des fichiers. Certains logiciels n'ont pas compilé. Nous avons récupéré de l'ordre de 500 logiciels fonctionnels.

Contrairement aux DAG générés aléatoirement, les DAG obtenus ont des paramètres qui varient fortement :

- le nombre de nœuds,
- le parallélisme de l'application,
- la quantité de données à transférer,
- la structure,
- la quantité de travail.

Ainsi, une modification sur la plate-forme n'influencera pas forcement de la même façon toutes les exécutions.

Sur ces DAG, nous ne pouvons pas augmenter la quantité de données transférées en modifiant les paramètres de génération. Pour modifier le temps de transferts par rapport au temps de calcul, la performance des liens réseau doit être modifiée. Nous changeons dans certaines expériences le débit des liens réseaux. Ce changement n'aura pas le même impact sur tous les DAG. Les DAG transférant peu de données ne seront pas impactés.

Pour avoir une idée du comportement global des différents algorithmes, nous nous intéressons à la moyenne des temps d'exécution. Nous comparons sur la figure 5.18 list_min, WSCOM, WSCOM$_{pf}$ et le meilleur algorithme en-ligne sur deux plates-formes : clique et étoile. Le meilleur algorithme en-ligne est WS_RR pour ces DAG extraits.

La principale différence avec les résultats précédents est la baisse de performance pour l'algorithme WSCOM$_{pf}$.

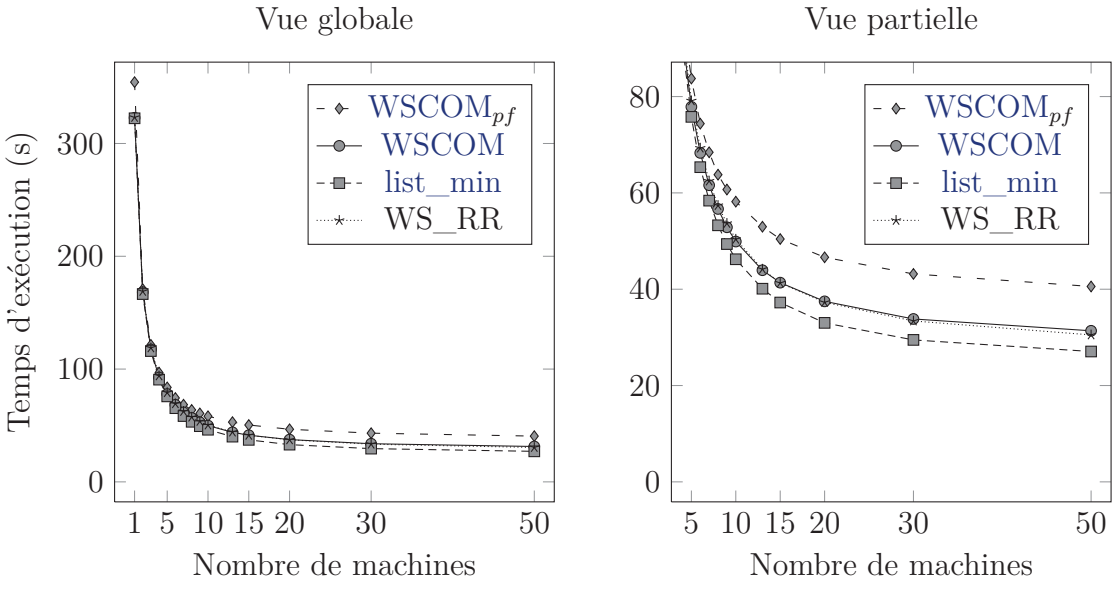


FIGURE 5.18 – Comparaison de list_min, WSCOM, WSCOM$_{pf}$ et WS_RR avec des DAG provenant de Makefile sur une grappe.

Nous commençons donc par analyser cette baisse de performances dans la section a. Puis, la section b présente plus en détails les résultats obtenus avec WSCOM sur différentes plates-formes.

a Exécution avec WSCOM$_{pf}$

La principale différence entre WSCOM$_{pf}$ et WSCOM provient de la restriction des vols aux tâches virtuelles. La figure 5.19 compare le temps d'exécution obtenu avec WSCOM et WSCOM$_{pf}$. Chaque point représente un DAG différent. L'abscisse du point est le temps d'exécution obtenu avec WSCOM, son ordonné avec WSCOM$_{pf}$. Lorsque le point est au dessus de la droite ($f(x) = x$), l'heuristique WSCOM a un temps d'exécution inférieur à celui de WSCOM$_{pf}$. La courbe montre que WSCOM$_{pf}$ améliore les performances sur très peu de DAG et détériore fortement les performances sur une cinquantaine de DAG.

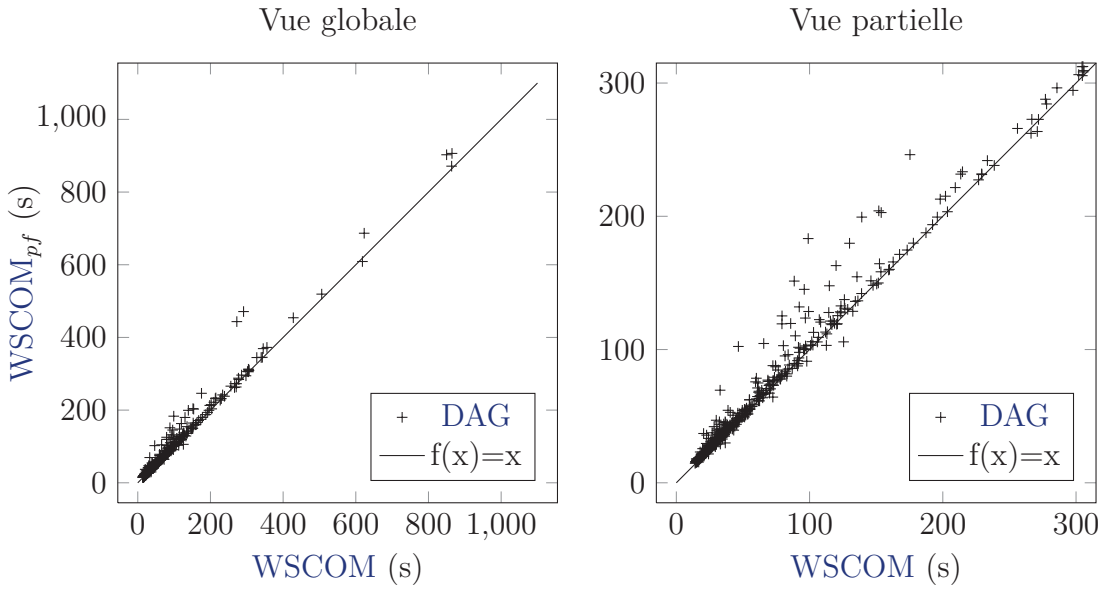


FIGURE 5.19 – Comparaison de WSCOM, WSCOM$_{pf}$ avec des DAG provenant de Makefile sur une grappe.

Dans l'algorithme WSCOM$_{pf}$ le risque est que durant l'exécution, un processeur n'ait plus de travail. Ce processeur vole des tâches virtuelles mais elles ne contiennent pas de tâches prêtes. Ce processeur continue à voler jusqu'au moment ou des tâches deviennent prêtes. Le problème est que l'exécution des tâches virtuelles a ajouté dans sa pile des tâches de calcul. Ces tâches ne pourront plus être volées. Ainsi, ces tâches devront être exécutées uniquement par ce processeur. Afin de mettre en évidence l'impact de cette restriction, nous proposons d'analyse le diagramme de «Gantt» obtenu sur un exemple précis. Ce diagramme est illustré par la figure 5.20. Les temps d'inactivités sont représentés en blanc et l'exécution d'une tâche est représentée par un rectangle gris. Sur ce diagramme il est clair qu'après 17 secondes d'exécution seul le processeur 3 reste actif. Ce processeur a récupéré un ensemble des tâches de calcul durant les faibles temps d'inactivité au début de l'exécution.

Pour conserver l'équilibrage de la charge, il y a deux possibilités : soit permettre de voler les tâches de calcul, soit éviter de voler les tâches virtuelles qui ne mènent pas à des tâches prêtes.

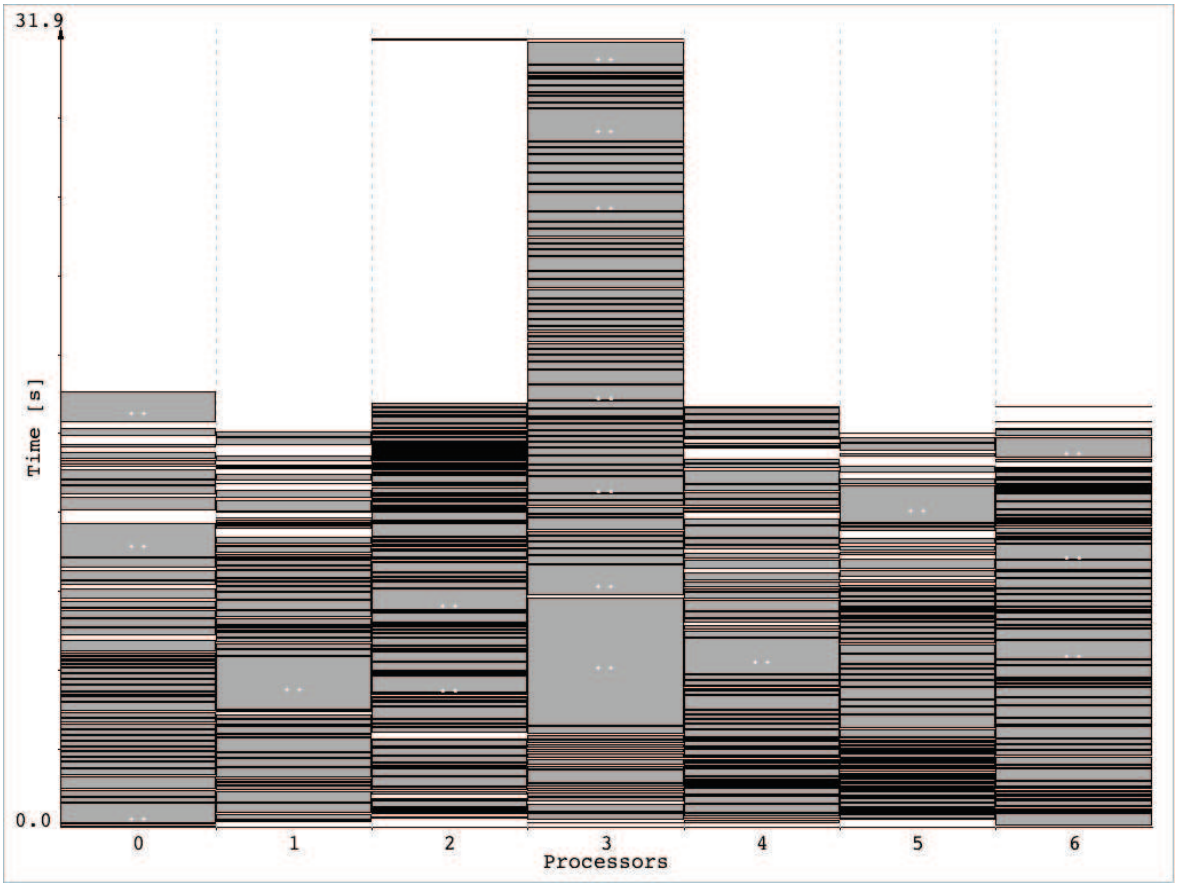


FIGURE 5.20 – Diagramme de Gantt représentant l'exécution d'un DAG avec WSCOM$_{pf}$.

En permettant de voler les tâches de calcul, l'envoi en avance des données risque d'envoyer les données plusieurs fois. Le nombre d'envois d'une même donnée ne sera néanmoins pas supérieur à deux car lorsqu'une tâche de calcul est volée, elle est exécutée directement par le voleur. Il serait ainsi possible d'éviter le déséquilibre au cours de l'exécution tout en limitant la quantité de données transmises. En revanche lors des vols de tâches de calcul, l'algorithme d'ordonnancement ne profite plus de l'information sur la structure du DAG.

Une autre possibilité pour profiter de la structure du DAG même après une phase avec peu de parallélisme est d'éviter de voler les tâches virtuelles ne menant pas à des tâches prêtes. L'idée serait d'ajouter les tâches virtuelles au milieu du DAG après les tâches générant du parallélisme.

b Exécution avec WSCOM

Certains points de l'analyse réalisée pour WSCOM$_{pf}$ restent valables pour WSCOM. Ce dernier est moins impacté par le problème de l'équilibrage de la charge car les vols des tâches de calculs ne sont pas interdits.

Pour analyser plus finement les performances, nous comparons WSCOM à la meilleure heuristique hors-ligne et celle en-ligne pour chaque graphe particulier.

Nous commençons par la comparaison avec list_min sur une plate-forme ayant une structure en forme de clique avec 5 machines. Les résultats sont sensiblement identiques pour d'autre nombres de machines. La figure 5.21 affiche un point par DAG. L'abscisse d'un point désigne le temps obtenu par WSCOM et son ordonnée le temps obtenu avec list_min.

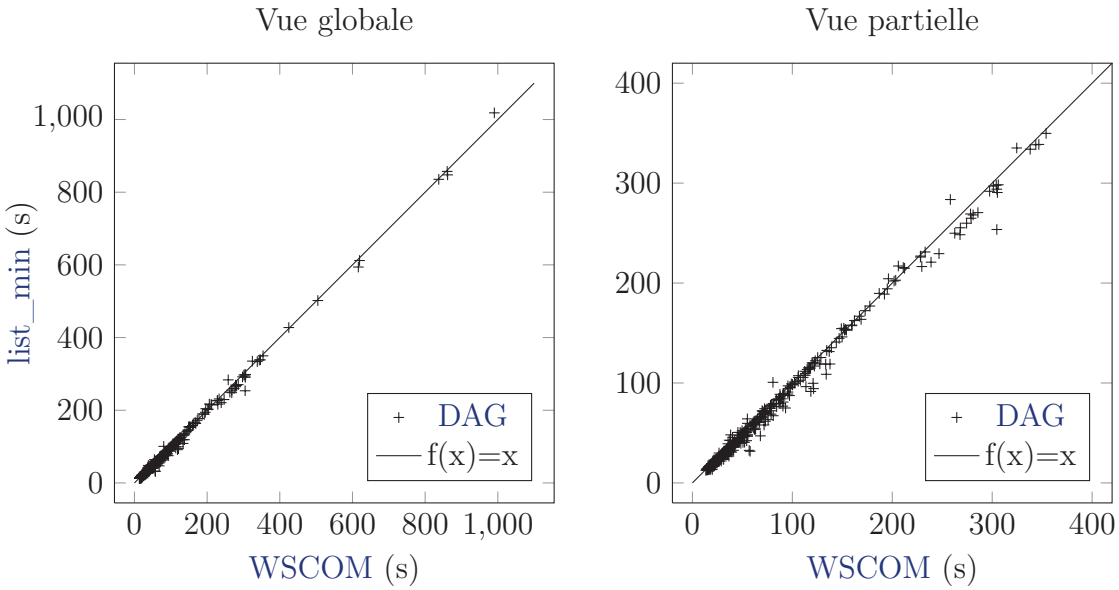


FIGURE 5.21 – Comparaison de list_min, WSCOM avec des DAG provenant de Makefile sur une clique de 5 machines.

WSCOM a des performances proches de celles de list_min. Sur plus des trois quarts des DAG, la perte est inférieure à 20%. De plus, la médiane se situe en dessous de 10%. Le passage à une structure de type grappe de calcul réduit les écarts.

La connaissance des temps d'exécution et de communication permet à l'heuristique list_min d'être moins impactée par le chemin critique et les communications. Enfin, WSCOM n'envoie pas les données au plus tôt, ce qui implique des temps d'inactivité supplémentaires avant l'exécution des tâches.

Malgré ces désavantages, WSCOM obtient un temps d'exécution comparable à celui réalisé par les algorithmes hors-ligne.

Nous nous intéressons maintenant à la comparaison avec les algorithmes en-ligne. L'analyse réalisée est basé sur une plate-forme de type grappe de calcul avec 8 machines. La figure 5.22 compare WSCOM à l'algorithme WS_RR qui est le meilleur algorithme en-ligne dans ce cas. La représentation utilisée est la même que pour la figure 5.21.

Sur les DAG ayant le moins de travail, les deux heuristiques ont des performances équivalentes. Les DAG extraits ont des caractéristiques très variées : la quantité de données traitées, la quantité de travail, le chemin critique, le parallélisme maximal. Les deux algorithmes ont des performances équivalentes sur l'ensemble des DAG. Nous pouvons remarquer également que sur les DAG ayant le plus de travail, WSCOM est sensiblement meilleur.

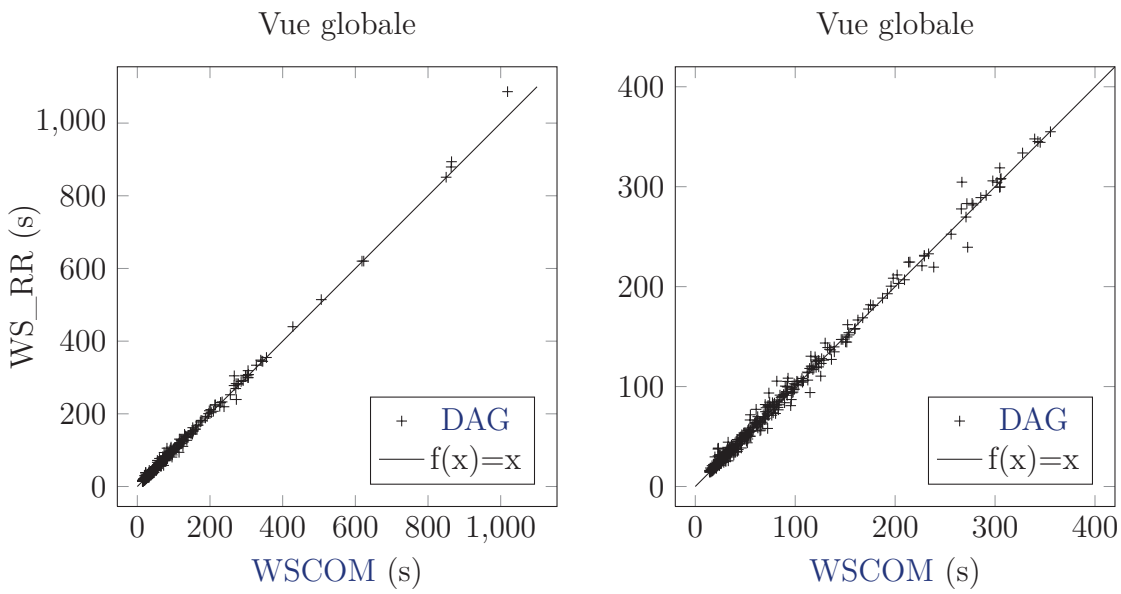


FIGURE 5.22 – Comparaison de WSCOM et WS_RR avec des DAG provenant de Makefile sur une grappe.

Finalement, dans un dernier jeu d'expériences, nous cherchons à quantifier dans quelle mesure WSCOM permet de réduire la dépendance aux ressources du réseau.

WSCOM a été développé pour améliorer les performances du vol de travail lorsqu'il y a suffisamment de parallélisme et des coûts de communications importants. Nous avons donc sélectionné les DAG ayant une quantité de données globale supérieure à 100Mo (quantité de données cumulée ce qui reste donc faible par arête) et un parallélisme supérieur à 5 (la plate-forme considéré par la suite a 5 machines). Le nombre de DAG correspondant est d'environ 100.

Afin de voir l'impact de la bande passante sur les algorithmes, nous comparons sur chaque DAG la bande-passante minimale sur la plate-forme pour obtenir un «speed-up»(Temps séquentiel divisé par le temps sur p machines) minimal. La figure 5.23 compare le vol de travail classique et WSCOM. L'abscisse du point est la bande passante minimale pour obtenir un «speed-up» supérieur à 4 pour WSCOM (le nombre de machines considéré est toujours de 5). L'ordonnée est la bande-passante minimale pour le vol de travail. La taille du point représente le nombre de DAG présents sur cette valeur (de 1 à 8). Pour certains DAG, le mécanisme de vol de travail classique n'atteint pas un «speed-up» supérieur à 4. Les points correspondant à ces applications sont au dessus la droite horizontale tracée sur la figure.

Dans la majorité des cas, WSCOM permet d'utiliser des bandes passantes inférieure au vol de travail classique. WSCOM nécessite jusqu'à 10 fois moins de bande passante pour les DAG où les deux algorithmes obtiennent un «speed-up» supérieur à 4.

Lorsque l'on réitère cette comparaison pour une valeur minimale de «speed-up» plus grande, les points se rapprochent du point (10^3,10^3). Lorsque l'on réitère cette comparaison pour une valeur minimale de «speed-up» plus faible, les points se rapprochent du point (10^0,10^0). Dans tous les cas, la majorité des points se situe au dessus de la droite $f(x) = x$.

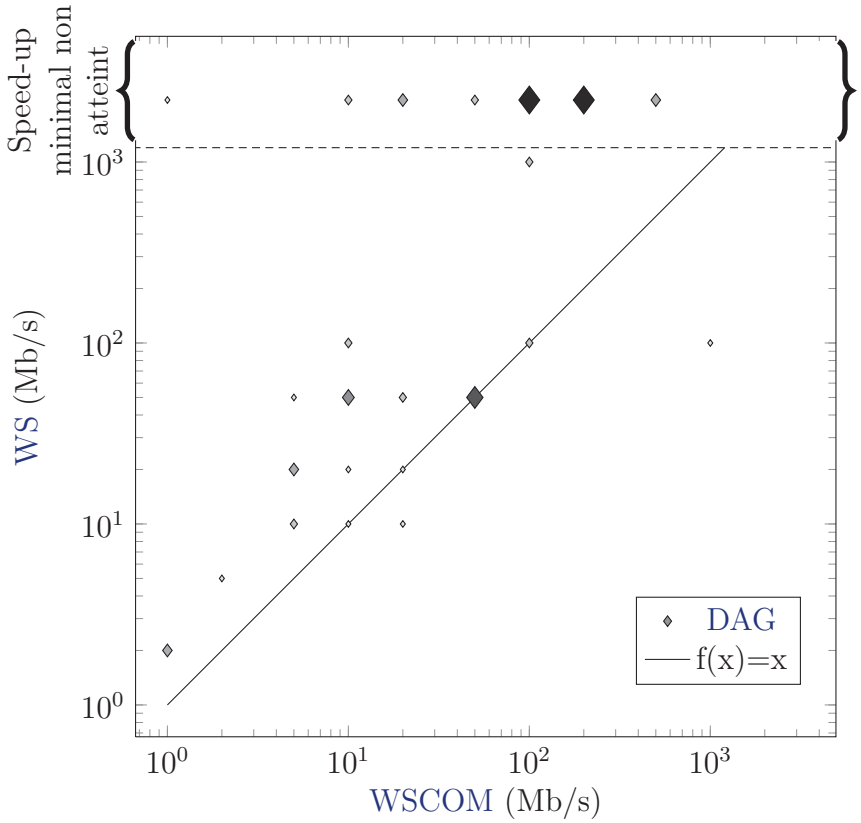


FIGURE 5.23 – Bande passante réseau minimale permettant d'avoir speed-up supérieur à 4 avec 5 machines (vol de travail, WSCOM).

Les DAG pour lesquels cela se passe moins bien que le vol de travail, ont des points de contraction au milieu du DAG et beaucoup de tâches indépendantes. De tels DAG présentent une structure quasi-plate et le placement des tâches non prêtes dans la pile des processeurs impacte dans ce cas légèrement les performances de WSCOM.

5.3 Conclusion

Dans la seconde partie de cette thèse, nous avons analysé la gestion des communications au sein d'algorithmes en-ligne comme le vol de travail. La gestion des communications n'est généralement pas réellement prise en compte.

La gestion des communications intervient à deux niveaux : le volume de données transférées et la réalisation des transferts.

Nous avons proposé plusieurs méthodes permettant de limiter la quantité de transferts effectués en se basant sur la structure du DAG.

La première méthode est WS_{convex} présenté dans la section 4.2.2 (page : 67). Cet algorithme se base sur un algorithme de découpe en partie indépendante. Nous avons analysé Les performances de cet algorithme sur les DAG générés aléatoirement. Cet algorithme n'améliore pas les performances par rapport au vol de travail classique. Cependant, il serait intéressant d'analyser ces performances plus en détail en modifiant le nombre de niveaux de récursion lors de la découpe. Le principal inconvénient de cet algorithme vient du temps nécessaire pour réaliser les modifications sur le DAG, surtout dans le cas où la structure du DAG est complexe comme pour ceux extraits de Makefiles.

La seconde méthode est basée sur l'utilisation de la symétrie du DAG. Dans cette méthode, plusieurs variantes ont été développées : WSCOM, WSCOM_{pf} et WSCOM_{tree}.

Sur les DAG générés aléatoirement, nous avons montré que WSCOM permettait une amélioration significative des performances par rapport aux algorithmes en-ligne. Ce gain est de l'ordre de 20% avec un CCR de $1,5$. En diminuant le CCR, la différence de performance augmente. Dans les expériences classiques du domaine, le CCR descend jusque $0,1$.

De plus, WSCOM_{pf} permet, par l'envoi au plus tôt des données, d'atteindre les meilleures performances et notamment d'obtenir un temps d'exécution plus faible qu'avec les algorithmes hors-ligne, dans le cas d'une plate-forme susceptible de congestion.

Sur ce type de plate-forme, WSCOM_{pf} obtient un gain de performances allant jusqu'à 20% avec un CCR égale à $1,5$.

Sur les DAG extraits de Makefile, les algorithmes étudiés se comportent d'une manière similaire, sauf WSCOM_{pf} qui montre une sensibilité à une réduction du parallélisme au cours de l'exécution. Nous avons proposé différentes pistes d'amélioration envisageables :

- Ajouter les tâches virtuelles au milieu du DAG grâce à une détection des tâches générant du parallélisme,
- Permettre de voler les tâches de calculs quitte à retransmettre une seconde fois les données.

Nous avons également montré que WSCOM permet d'utiliser des plates-formes avec des bandes-passantes plus faibles que le vol de travail classique.

En définitive, les approches choisies se sont avérées efficaces.
- la réduction du nombre de communications réduit également les temps d'exécution ;
- l'utilisation d'une table de hachage distribuée a bien permis une amélioration des performances ;
- l'utilisation des données disponibles sur la topologie du graphe permet d'améliorer judicieusement la localité.

III

Ordonnancement pour une plate-forme hiérarchique

Introduction

Après avoir introduit les algorithmes en-ligne et la gestion des communications au sein des algorithmes hors-ligne, nous avons présenté l'algorithme WSCOM dans le chapitre 4. Il est basé sur le mécanisme de vol de travail permettant de limiter le nombre de communications effectuées au cours de l'exécution. WSCOM ajoute des tâches virtuelles sur le DAG de l'application pour orienter les vols vers des parties indépendantes du DAG. De plus, nous avons détaillé la gestion de l'exécution et des transferts de données.

Cet algorithme permet de réduire l'impact des communications sur le temps d'exécution. En particulier, cette optimisation réduit la quantité de données transférées. De plus, WSCOM permet l'envoi de données dès qu'elles sont produites. Durant l'exécution, WSCOM augmente ainsi le taux de recouvrement des communications par l'exécution de tâches.

Par contre, cet algorithme ignore totalement la structure de la plate-forme. Or sur la plate-forme réelle, les processeurs peuvent être connectés entre eux avec une structure réseau complexe, par exemple une inter-connexion de grappes de calcul. La plate-forme est généralement organisée en groupes de processeurs. Un groupe peut notamment être inclus dans un autre groupe plus grand. Cette structure peut avoir un impact sur :

- le coût des communications en mémoire distribuée,
- le temps d'accès aux données en mémoire partagée.

Dans les deux cas, l'optimisation du temps d'exécution passe par une amélioration de la localité des données pour éviter les communications entre les processeurs qui communiquent lentement.

Dans le chapitre 6, nous présentons deux algorithmes d'ordonnancement par vol de travail. Le premier nommé Probabilist Work-Stealing (PWS) modifie la probabilité de voler certaines machines. Il privilégie le vol des tâches présentes sur des processeurs proches. Le second algorithme Hierarchical Work-Stealing (HWS) utilise deux niveaux de hiérarchie, une pour distribuer le travail entre les groupes, l'autre pour le distribuer au sein d'un groupe.

Puis, le chapitre 7 présente l'analyse des performances de ces algorithmes. Une première analyse théorique montre que HWS réduit le nombre de communications réalisées entre les groupes de machines et donne une borne sur le temps d'exécution. La seconde analyse présente les performances obtenues en pratique avec les algorithmes proposés. Cet partie compare les performances sur des applications transférant des données volumineuses et des applications générant un grand nombre de tâches utilisant peu de données

Optimisation de la localité des données
6

Sommaire du chapitre

Dans ce chapitre, nous nous intéressons à limiter l'impact de la topologie de la plate-forme sur le temps de complétion. Sur l'ensemble de la plate-forme, les coûts de communications entre les processeurs varient en fonction du récepteur et de l'envoyeur.

Sur de nombreuses plates-formes actuelles, il existe des groupes de machines (ou de processeurs) qui communiquent plus rapidement les uns avec les autres qu'avec des machines externes au groupe. Ces différences de performances sont dues aux liens de communications utilisés lors du transfert des données.

Afin de donner une description plus précise d'un groupe de processeurs, nous présentons des regroupements possible de processeurs sur des plates-formes actuelles :

Machines multiprocesseurs Un groupe est l'ensemble des cœurs d'un même processeur.

Machines Non-Uniform Memory Access (NUMA) L'ensemble des processeurs sur une même carte mère forme un groupe.

Grappes de calcul Un groupe correspond à l'ensemble des processeurs d'une machine.

Grilles de calcul Un groupe correspond à une grappe de calcul.

Certaines structures, comme les grilles de calcul, contiennent plusieurs niveaux de hiérarchie (Grappes de calcul, Machines multiprocesseurs). Il est possible de découper la plate-forme en groupes de processeurs plus ou moins grands.

Dans les grappes ou grilles de calcul, certains liens de communication peuvent être partagés avec d'autre utilisateurs. Une utilisation abusive de ceux-ci peut entraîner une augmentation du temps d'exécution de notre application. En réduisant l'utilisation des liens de communications les plus faibles et en améliorant la localité des données, nous espérons pouvoir réduire le temps d'exécution obtenu avec le vol de travail classique.

Nous nous intéressons donc à réduire l'utilisation des liens réseaux partagés et les moins performants pour tenter de réduire le temps d'exécution.

Pour réduire l'utilisation de certains liens de communication, une méthode indirecte est d'améliorer la localité des données. Nous avons introduit dans la section 3.3.1 le principe des algorithmes Parallel Depth First (PDF) [7, 17] et le vol de travail à fenêtre [80]. Ces mécanismes permettent d'améliorer la localité des données par rapport à un ordonnancement avec le vol de travail classique. En revanche, en mémoire distribuée, ces algorithmes nécessitent des synchronisations supplémentaires qui les rendent inefficaces. Dans notre approche, l'objectif est de minimiser l'utilisation des liens de communication les plus lents. Cette optimisation consiste à améliorer la localité des données mais aussi à réduire le nombre de transferts effectués sur ces liens.

Pour prendre en compte la hiérarchie en minimisant les synchronisations, la section 6.1 introduit une première heuristique Probabilist Work-Stealing (PWS) qui privilégie le vol des tâches sur les processeurs les plus proches. Enfin, l'algorithme Hierarchical Work-Stealing (HWS) tente de limiter les transferts de tâches sur les liens les plus lents. Cet algorithme est introduit section 6.2.

6.1 PWS : Vol de travail probabiliste

Nous présentons dans cette section PWS qui tente de minimiser les temps de recherche de travail et de favoriser l'échange de travail entre les processeurs proches. L'objectif est de privilégier l'utilisation des liens les plus performants. Pour effectuer cela, PWS change uniquement le mécanisme de sélection de la victime dans l'algorithme de vol de travail classique.

La probabilité de voler un processeur est choisie en fonction de la distance entre la victime et le voleur : idéalement, plus le temps pour communiquer avec un processeur est important et plus la probabilité de le voler serait faible. En pratique, le fonctionnement de l'algorithme est légèrement différent. La distance entre les processeurs est perçue uniquement par la description en groupe de la plate-forme. Nous précisons son fonctionnement par une description en pseudo-code dans l'algorithme 2.

Algorithme 2 : Probabilist Work-Stealing (PWS)

```
Function Sélection_aléatoire_d'un_processeur() begin
    if Valeur_aléatoire_entre_0_et_1 () < p_roba then
        ;        /* Le nombre aléatoire entre 0 et 1 est comparé à la
        probabilité de voler un processeur externe au groupe */
        return Sélection_d'un_processeur_externe_au_groupe ();
    else
        return Sélection_d'un_processeur_externe_au_groupe ()
```

La modification de la probabilité impacte au démarrage la répartition des tâches et la recherche de travail.

Au début de l'exécution, l'ensemble du travail est généralement contenu dans une tâche placée dans la pile d'un processeur. Le travail va donc se répartir principalement dans le groupe de processeurs ayant la première tâche. Sur les autres sites, il n'y a pas de tâches à voler. Les vols locaux sont rapides et échouent. Des tâches seront obtenues lorsqu'un vol distant sera réalisé dans un groupe possédant du travail.

Au cours de l'exécution, la recherche de travail sera principalement réalisée au sein du groupe. Cela permet d'améliorer la localité. Lorsqu'il y a peu de tâches volables dans le groupe, le nombre de vols échoués augmente. Parmi ces vols, certains seront émis vers les autres groupes (potentiellement avant que tous les processeurs ne soientt inactifs).

Le risque de cette heuristique est d'utiliser une probabilité de voler les machines des autres groupes trop faible et ainsi ne pas répartir équitablement le travail entre les processeurs. En revanche, elle est simple et permet de prendre en compte tous les types de structures (plusieurs niveaux de hiérarchie) car elle se base uniquement sur les temps de communications entre les processeurs.

Le choix de la probabilité peut dépendre de la plate-forme mais aussi de l'application. Le temps d'exécution d'une application travaillant sur des données peu volumineuses sera par exemple, sensible à la répartition du travail. La probabilité ne devra pas être choisie trop faible. En revanche, pour réduire celui d'une application utilisant des données volumineuses, l'algorithme d'ordonnancement doit minimiser l'impact des transferts de données. La probabilité ne devra pas être choisie trop élevée dans ce cas.

6.2 HWS : Vol de travail hiérarchique

Avec PWS, nous avons limité les quantités de données transférées entre deux groupes en réduisant la probabilité de voler une machine présente sur un autre groupe. Lorsqu'un groupe n'a plus de travail, il est possible que plusieurs vols distants soient effectués au même instant. Ainsi, il y a un risque de générer de la congestion sur le réseau. Pour éviter ce risque de congestion, HWS limite le nombre de machines communicant par les liens de communication inter-groupes. Le nombre de vols inter-groupes sera donc considérablement réduit.

Dans ce cas, le risque est d'avoir un important déséquilibre entre les groupes. Pour éviter un tel déséquilibre, nous souhaitons prendre une grande quantité de travail lors des vols entre deux groupes.

Au début de l'exécution, l'ensemble du travail est contenu dans une tâche. L'exécution de cette tâche entraîne la création de plusieurs tâches de tailles inférieures. Puisque les processeurs du même groupe communiquent rapidement entre eux, le travail sera rapidement découpé en plus petits travaux et réparti sur l'ensemble du groupe. Nous élisons une machine par groupe qui se chargera de gérer le travail contenu au sein de son groupe. La machine élue conservera les tâches contenant une grande quantité de travail. Elle fournira du travail à son groupe à partir de ces tâches.

La mécanisme de HWS que nous détaillons dans cette section, a été proposé et analysé théoriquement dans notre article [61] en 2010 à EUROPAR. Récemment, ce

mécanisme a été repris dans l'article [60]. Un des incréments de cet article est une analyse théorique du nombre de défauts de caches sur les plates-formes à mémoire partagée. De plus, il propose une analyse expérimentale en mémoire partagée montrant des gain jusqu'à 20% par rapport au vol de travail classique implanté dans *Cilk*.

Nous présentons maintenant plus en détail le fonctionnement de HWS. Dans un premier temps, nous rappelons la gestion des communications au sein de *Kaapi* dans la section 6.2.1, ce qui justifie les choix réalisés dans cet algorithme. Puis, la section 6.2.2 détaille comment les tâches avec une importante quantité de travail sont différenciées. Ensuite, la gestion de la répartition du travail entre les groupes de machines est présentée section 6.2.3. Enfin, nous précisons les points sensibles de l'implantation dans la section 6.2.4.

6.2.1 Gestion des transferts de données au sein de *Kaapi*

Lors de l'exécution d'une application par vol de travail, les transferts de données peuvent être réalisés à différents moments de l'exécution comme la section 4.1 le précise. Les deux principales méthodes sont :

préemptive Lors d'un vol, toutes les données nécessaires à son exécution sont transférées. Cette méthode permet d'avoir directement les données et de ne pas attendre leurs transferts lors de l'exécution de la tâche. En revanche, les données peuvent être transférées inutilement.

paresseuse Les données sont transférées uniquement lorsqu'elles sont accédées directement. 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 potentiellement deux transferts de données : un au moment du vol et un dès que le résultat est calculé. En réduisant la quantité de vols distants, nous réduirons directement l'utilisation de ces liens et augmenterons potentiellement les performances.

Dans l'algorithme HWS, Nous tentons d'appliquer le procédé d'équilibrage de la charge entre deux processeurs au niveau de deux groupes de machines (*i.e.,* prendre la moitié du travail présent sur le groupe volé).

6.2.2 Tâches globales et locales

Pour conserver l'équilibrage de charge entre les groupes tout en réduisant le nombre de vols, nous proposons de baser HWS sur l'idée suivante : les tâches de tailles réduites ne peuvent être transférées d'un groupe à un autre. Les tâches de taille importante sont réservées pour équilibrer potentiellement la charge entre les groupes.

Nous considérons dans notre cas un ensemble de tâches décrites de manière récursive. Les tâches représentent le travail contenu dans toutes ses descendantes. Celles de profondeur faible contiennent ainsi potentiellement plus de travail. Pour différencier les

tâches, nous nous basons sur une limite fournie par l'utilisateur. Nous définissons de par cette limite deux types de tâches :

Les tâches globales ont une profondeur inférieure au seuil. Elles peuvent être volées entre groupes.

Les tâches locales ont une profondeur supérieure au seuil. Elles 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. La figure 6.1 illustre le «Directed Acyclic Graph» (DAG) de l'application ainsi que la différenciation des tâches locales et globales. Pour éviter la découpe du travail au dessus de la limite au sein d'un groupe et permettre aux autres groupes de trouver facilement des tâches au dessus de la limite, nous proposons de centraliser les tâches globales sur le maître du groupe.

De plus, ces tâches globales réalisent généralement une découpe du travail. Elles représentent donc beaucoup de travail au travers de leurs descendants mais ces tâches en elles-mêmes sont exécutées rapidement. En centralisant ces tâches globales, elles seront exécutées uniquement par les maîtres. Le sur-coût lié à l'exécution de ces tâches est donc généralement faible.

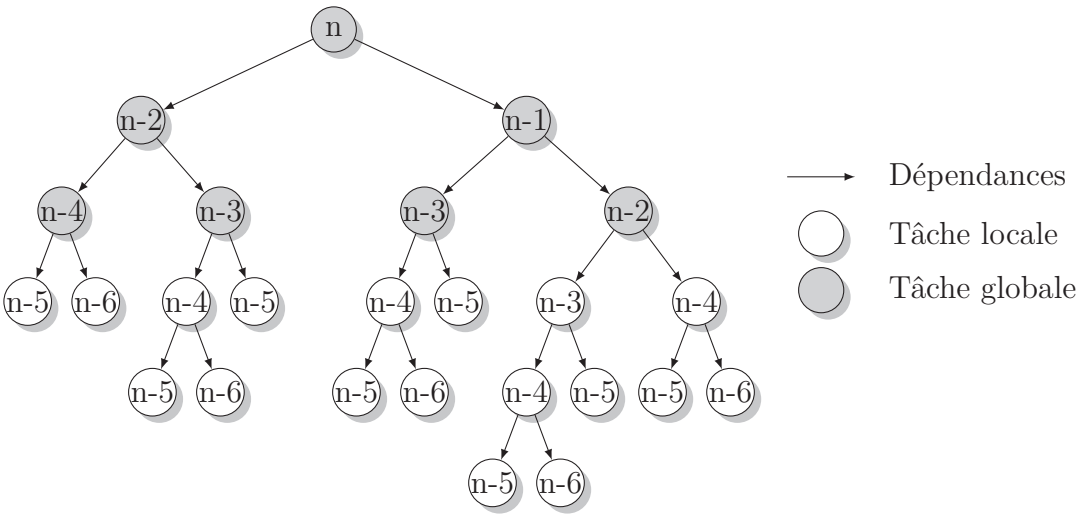


FIGURE 6.1 – Différenciation des tâches locales et globales sur l'application Fibonacci.

6.2.3 Équilibrage de la charge

Dans notre heuristique, les vols entre les groupes sont limités en réduisant le nombre de machines communicantes avec les machines externes au groupe. De plus, les vols distants sont limités aux vols de tâches de taille importante. Nous avons deux niveaux de hiérarchie :

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

Les machines de chaque groupe (esclaves) 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 les tâches fournies par le maître, nous utilisons l'algorithme de vol de travail classique en le restreignant au groupe. Ainsi, lorsqu'un ensemble de tâches est fourni au groupe, l'équilibrage de la charge est effectué efficacement sans connaître la quantité de travail présente. De plus, les données utilisées uniquement par ces tâches ne sont pas communiquées à des machines externes au groupe.

Pour réaliser la distribution du travail entre les groupes, 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. Après avoir ajouté des tâches à sa pile locale, le maître attend que les esclaves exécutent le travail. Celui-ci doit se réactiver plus tard pour fournir des tâches locales à son groupe. Pour ajouter des tâches dans sa pile locale il a deux possibilités :

1. Soit sa pile globale contient travail : il exécute le travail pour en fournir une partie à son groupe.
2. Soit sa pile globale est vide : il envoi une requête de vol auprès d'un autre maître.

Maintenant, il nous reste à déterminer l'instant où le maître placera dans sa pile locale du travail. Pour choisir cet instant, 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 Cluster-aware Load-based Stealing (CLS) [85].

Nous proposons d'utiliser la fréquence des vols au sein du groupe pour réaliser la détection. Lorsque la quantité de travail au sein du groupe diminue, la fréquence des vols augmente. En analysant la fréquence des vols, nous pourrons prendre une décision sans perturber le système. Pour implanter cela, nous ajoutons lors de la réception des vols par le maître un compteur permettant de mesurer la fréquence. Ce paramètre peut être modifié mais il impacte faiblement les performances de l'algorithme en pratique.

Une dernière méthode possible est d'attendre la fin complète du travail local. Elle est décrite en pseudo-code par l'algorithme 3. Cette méthode sera utilisée lors de l'analyse théorique des performances de HWS.

6.2.4 Implantation dans *Kaapi*

Pour analyser en pratique les performances de HWS, nous l'avons implanté dans la bibliothèque *Kaapi*. De cette implantation, nous souhaitons détailler deux points : la différenciation des tâches globales et locales, et la synchronisation entre le maître et les esclaves.

Algorithme 3 : Hierarchical Work-Stealing (HWS)

```
Procedure Maître () begin
    Data : T_initiale ;                 /* La tâche initiale de l'exécution */
    Data : Pile_globale ;                                  /* Pile de Tâches */
    Data : Pile_locale ;                                   /* Pile de Tâches */
    Data : limite ;                     /* Limite fournie par l'utilisateur */
    T_cur = non défini ;
    if le processeur a l'identifient 0 then
        T_cur = T_initiale ;            /* Initialisation d'un des processeurs */
        Hauteur_de_création (T_initiale) = 0 ;
    while des tâches ne sont pas exécutées do
        while T_cur est défini do
            H = Hauteur_de_création (T_cur) + 1 ;
            Tâches = Exécution (T_cur) ;
            ;       /* Exécute la tâche T_cur et retourne les tâches crées
            durant son exécution */
            if H < limite then
                for I ∈ Tâches do
                    Hauteur_de_création (I) = H ;
                    push (Pile_globale,I) ;
            else
                for I ∈ Tâches do
                    Hauteur_de_création (I) = H ;
                    push (Pile_locale,I) ;
                Attendre_la_fin_du_travail_local () ;
            T_cur = Pop(Pile_globale) ;
        P = Sélection_parmi_les_maîtres () ;
        T_cur = Vol_pile_globale (P) ;
Procedure Esclave Data : Pile_locale ;                 /* Pile de Tâches */
begin
    T_cur = non défini ;
    while des tâches ne sont pas exécutées do
        while T_cur est défini do
            Tâches = Exécution (T_cur) ;
            ;    /* retourne les tâches crées par l'exécution de T_cur */
            for I ∈ Tâches do
                push (Pile,I) ;
            T_cur = Pop(Pile) ;         /* Suppression d'une tâche prête */
        P = Sélection_d'un_processeur_au_sein_du_groupe () ;
        T_cur = Vol_pile_locale (P) ;
        push (Pile,T_cur) ;
        T_cur = Pop(Pile)
```

Nous détaillons dans un premier temps la différenciation des tâches globales. Dans *Kaapi*, une tâche est équivalente à une fonction. Lors de son exécution, la fonction est exécutée d'un seul tenant. Aucune instruction présente dans cette fonction ne sera exécutée ultérieurement. Les tâches créées durant l'exécution sont placées dans la pile. Les dépendances entre celles-ci peuvent être quelconques. Ainsi, certaines ne sont pas directement prêtes et dépendent de l'exécution d'autres tâches. Nous ne connaissons donc pas à la création d'une tâche sa profondeur dans le DAG complet de l'application. Nous pouvons uniquement calculer sa profondeur de création. Pour savoir si le maître place cette tâche dans sa pile globale ou non, il se base donc sur la profondeur de création. L'utilisateur nous fournit la profondeur à laquelle les tâches ne contiennent plus suffisamment de travail. La figure 6.2 illustre la différenciation des tâches globales et locales dans le cadre d'une exécution avec *Kaapi*. Cette découpe nous permet de mieux illustrer le détail de l'exécution.

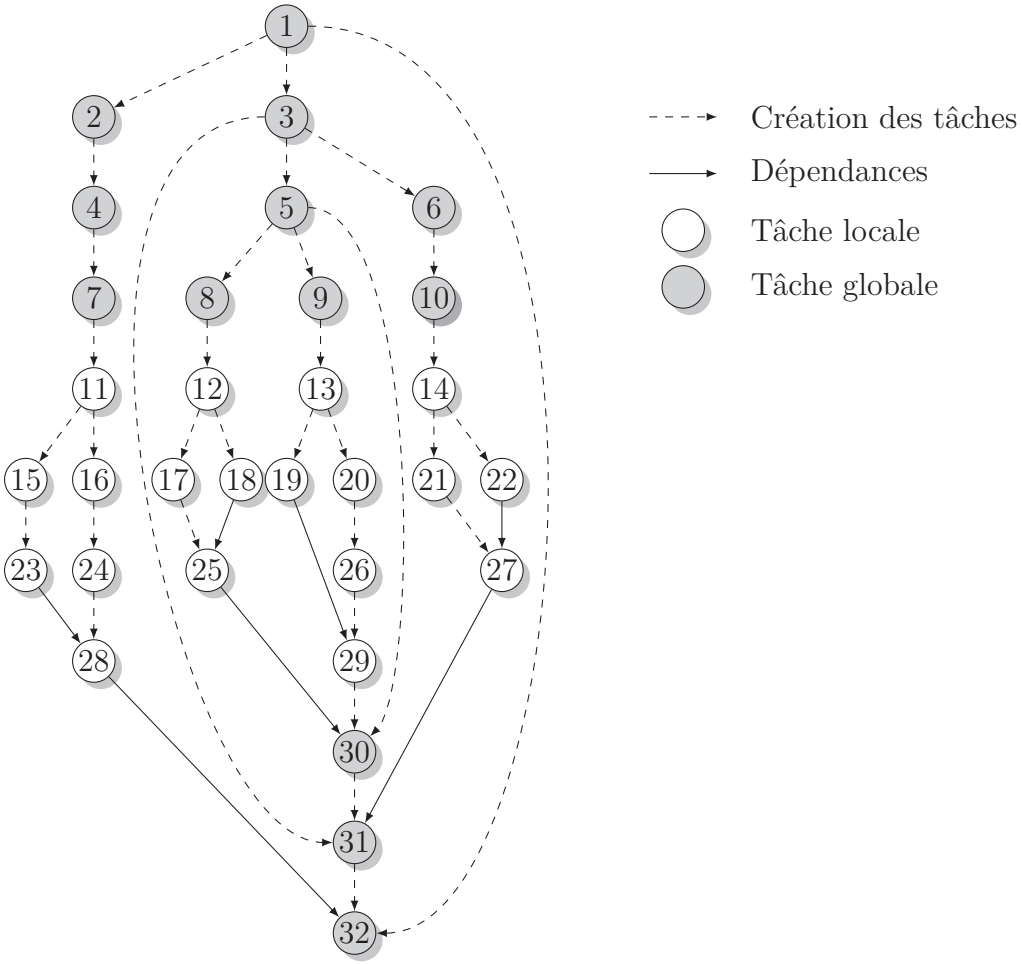


FIGURE 6.2 – Différenciation des tâches locales et globales de par la profondeur de création.

Nous explicitons maintenant les mécanismes de synchronisation des maîtres et des esclaves. Nous rappelons qu'un groupe de machine est lui-même potentiellement distribué.Un esclave peut demander du travail au maître pour plusieurs raisons.

La première concerne la terminaison de l'ensemble du travail fourni par le maître. Dans *Kaapi*, lorsqu'une tâche volée est terminée, le processeur qui l'a exécutée signale à la victime la disponibilité des données produites pour mettre à jour les dépendances. Dans ce cas aucun esclave n'exécute de tâches, nous profitons de ce signal pour donner la main au maître. Le maître obtient la puissance de calcul nécessaire car il est le seul à exécuter une tâche. Les autres processeurs (esclaves) réalisent des vols.

La seconde raison pour réactiver le maître est liée à la détection de manque de travail par l'un des esclaves. Cet esclave a réalisé plusieurs vols sans succès. Il demande donc du travail au maître. Puisque le groupe d'esclaves est distribué, les esclaves présents sur la même machine que le maître sont potentiellement entrain d'exécuter une tâche. Ainsi, lorsque le maître tente de fournir du travail, il peut être perturbé dans l'exécution des tâches. Pour limiter ces perturbations, nous suspendons le premier esclave qui termine l'exécution d'une tâche. Le travail présent dans la pile du processus suspendu peut être volé par les autres ou exécuté lorsque l'esclave sera réactivé.

Analyse de HWS et PWS 7

Sommaire du chapitre

Dans le chapitre précédent, nous avons présenté deux algorithmes Hierarchical Work-Stealing (HWS) et Probabilist Work-Stealing (PWS) s'ajustant à la topologie de la plate-forme.

Dans ce chapitre, nous analysons ces algorithmes afin de mettre en évidence leurs performances. La section 7.1 présente deux analyses théoriques permettant de borner le temps d'exécution et le nombre de vols entre les groupes pour chacun des algorithmes proposés.

Puis, la section 7.2 compare les performances obtenues expérimentalement par ces deux algorithmes avec celles de certains algorithmes existant dans la littérature. Dans cette analyse, nous détaillons le gain sur des applications communiquant des données volumineuses ou générant un grand nombre de tâches.

7.1 Analyse théorique

Nous commençons par analyser théoriquement les performances de l'algorithme PWS dans la section 7.1.1. Cette analyse permet de mettre en évidence l'évolution du nombre de vol entre les groupes en fonction de la probabilité choisie. De plus, le temps d'exécution de PWS reste proche du temps d'exécution avec le vol de travail classique.

Puis, nous présentons dans la section 7.1.2 une analyse de l'algorithme HWS. Pour cet algorithme, nous montrons que le temps d'exécution sans les communications est plus important mais le nombre de vols entre les groupes est quant à lui significativement réduit. Nous espérons qu'avec cette réduction, le temps d'exécution réel sera également réduit.

7.1.1 Analyse théorique de PWS

Nous analysons théoriquement dans cette partie l'algorithme PWS. Pour cette analyse, nous nous basons sur la démonstration du vol de travail classique basée les fonctions Strictly Increasing Per Steal (SIPS) introduites section 3.2.2.

Pour analyser PWS, nous considérons deux niveaux de hiérarchie : le groupe et l'extérieur du groupe. De plus, nous faisons l'hypothèse que la probabilité de voler au sein du groupe est la même pour chacun des groupes.

Ainsi, un processeur qui est dans le groupe i, vole dans son groupe avec une probabilité v. Ce processeur vole avec une probabilité $1 - v$ une machine externe à son groupe. De plus, nous notons m le nombre de processeur au sein du groupe le plus grand ($m = \max_i p_i$).

Nous rappelons les notations utilisées par les fonctions SIPS. Une fonction f_j est associée à chaque processeur j et associe à chaque instant la profondeur minimale des tâches de sa pile. Nous notons F le minimum sur j de toutes les fonctions f_j.

Cette fonction est identique à celle utilisée dans la section 3.2.2 pour analyser le vol de travail classique.

L'analyse de PWS est divisée en deux parties distinctes : la première réalisée dans le lemme 7.1 propose une borne du nombre de vols effectués au cours de l'exécution ; la seconde se base sur ce nombre de vols pour analyser le temps d'exécution, le nombre de vols «locaux» et le nombre de vols «distants».

Lemme 7.1. *Lors de l'ordonnancement d'une application avec l'algorithme PWS, le nombre de vols réalisés durant l'exécution est inférieur à*

$$\left(\frac{p-m}{1-v}\log\left(p-m-1\right)+\frac{m}{1-v}\right)\times D$$

avec $m = \max_i p_i$.

Démonstration. Pour borner le nombre de vols total, nous souhaitons connaitre le nombre de vols nécessaires pour incrémenter F de 1. L'ordre d'exécution en profondeur

ainsi que l'arité bornée à 2 implique que le nombre de tâches de profondeur F est inférieur ou égal à p. Ainsi afin d'augmenter F de 1, chaque processeur ayant une tâche de profondeur égale à F doit être volé.

En bornant le nombre de vols nécessaires pour satisfaire cette condition, nous pouvons alors majorer le nombre de vols total car lorsque F est égale à D, l'exécution est terminée.

Nous nous intéressons donc au nombre minimal de vols pour atteindre l'ensemble des machines présentes sur la plate-forme.

Soit $T(p)$ le nombre de vols nécessaires pour atteindre l'ensemble des p processeurs. Nous avons que $T(p) = \sum_{n=0}^{p-1}(T_{n,p})$ avec $T_{n,p}$ le nombre de vols nécessaires pour voler $n+1$ processeurs différents.

Notons maintenant q_j^n la probabilité qu'il y ait au moins j vols différents pour atteindre un $(n+1)^{\text{ème}}$ processeur (au moins j vols afin de ne pas voler un processeur qui l'a déjà été).

Nous avons directement que q_1^n est égale à 1 (il faut au moins un tirage pour voler un nouveau processeur).

Considérons un vol. Notons k le numéro du groupe du voleur. En supposant que nous avons volé n_1 processeurs au sein du groupe k et n_2 processeurs à l'extérieur du groupe ($n = n_1 + n_2$), la probabilité de voler un processeur qui a déjà été volé est alors égale à : la probabilité de voler un processeur parmi les n_1 déjà volés dans le groupe k (i.e. $v \times \frac{n_1}{p_k}$), plus la probabilité de voler un des n_2 processeurs à l'extérieur de notre groupe k (i.e. $(1-v) \times \frac{n_2}{p-p_k}$).

Ainsi, nous avons par récurrence :

$$q_j^n = q_{j-1}^n \times \left(v \times \frac{n_1}{p_k} + (1-v) \times \frac{n_2}{p-p_k} \right)$$

Cette probabilité dépend directement de k, n_1 et n_2 ce qui est problématique car il est impossible de les connaitre. Nous cherchons donc une borne supérieure sur cette expression.

De part le fonctionnement de PWS, nous savons que pour tout groupe i, la probabilité v de voler au sein du groupe i est supérieure à la probabilité $1-v$ de voler à distance. De plus, nous considérons qu'aucun groupe ne contient plus que la moitié des machines : $p_i < p - p_i$.

Ainsi $\frac{v}{p_i} > \frac{1-v}{p-p_i}$. Par conséquent, pour un n fixé, $\left(v \times \frac{n_1}{p_i} + (1-v) \times \frac{n_2}{p-p_i}\right)$ est maximal pour n_1 maximal et donc p_i maximal.

Nous en déduisons que pour $n \leqslant m$: $q_j^n \leqslant q_{j-1}^n \times \left(v \times \frac{n}{m}\right)$ En revanche, pour $n > m$: $q_j^n \leqslant q_{j-1}^n \times \left(v + (1-v) \times \frac{n-m}{p-m}\right)$.

Nous pouvons maintenant calculer $T(p)$:

$$T(p) = \sum_{n=0}^{p-1} T_{n,p} = \sum_{n=0}^{p-1} \sum_{j=1}^{\infty} q_j^n = \sum_{n=0}^{m} \sum_{j=1}^{\infty} q_j^n + \sum_{n=m+1}^{p-1} \sum_{j=1}^{\infty} q_j^n$$

Nous tentons maintenant de majorer les deux sommes indépendamment.

Nous avons donc pour la première partie de la somme :

$$\sum_{n=1}^{m}\sum_{j=1}^{\infty} q_j^n \leqslant \sum_{n=1}^{m}\sum_{j=1}^{\infty} v \times \frac{n}{m}$$
$$\sum_{n=1}^{m}\sum_{j=1}^{\infty} p_j^n \leqslant \sum_{n=1}^{m} \frac{1}{(1-v)\times\frac{n}{m}}$$

Une borne supérieure de cette somme peut être obtenue en majorant chacun des éléments de la somme. Pour cela, nous replaçons n par m.

$$\sum_{n=1}^{m}\sum_{j=1}^{\infty} q_j^n = \sum_{n=1}^{m} \frac{1}{1-v} = \frac{m}{1-v}$$

Maintenant dans la seconde somme, chaque vol réalisé au sein du groupe est effectué sur un processeur déjà volé. Ainsi, nous avons :

$$\begin{aligned}
\sum_{n=m+1}^{p-1}\sum_{j=1}^{\infty} q_j^n &\leqslant \sum_{n=m+1}^{p-1} \frac{1}{1-\left(1-v-(1-v)\times\frac{n-m}{p-m}\right)} \\
&= \frac{1}{1-v}\sum_{n=m+1}^{p-1} \frac{1}{1-\frac{n-m}{p-m}} \\
&= \frac{1}{1-v}\sum_{n=+1}^{p-1} \frac{p-m}{p-m-(n-m)} \\
&= \frac{p-m}{1-v}\sum_{n=m+1}^{p-1} \frac{1}{p-n} \\
&= \frac{p-m}{1-v}\sum_{n=1}^{p-m-1} \frac{1}{n} \\
&\sim \frac{p-m}{1-v}\log\left(p-m-1\right)
\end{aligned}$$

Ainsi, nous avons que :

$$T(p) \leqslant \frac{p-m}{1-v} O(\log\left(p-m-1\right)) + \frac{m}{1-v}$$

La borne sur $T(p)$ nous fournit le nombre maximal de vols nécessaires pour augmenter à coup sûr de 1 la fonction F qui est égale à la profondeur de la tâche la moins profonde présente sur toute la plate-forme.

Ainsi au cours de l'exécution, il y aura au plus $O\left(\left(\frac{p-m}{1-v}\log\left(p-m-1\right)+\frac{m}{1-v}\right)\times D\right)$ vols sur l'ensemble de la plate-forme car lorsque F est égale à D l'exécution est terminée. □

Théorème 7.2. *Lors de l'ordonnancement d'une application avec l'algorithme PWS, le temps d'exécution est inférieur à*

$$\frac{W}{p} + O\left(\left(\frac{p-m}{p(1-v)}\log(p-m-1) + \frac{m}{p(1-v)}\right) \times D\right).$$

De plus, le nombre de vols globaux est inférieur à :

$$O\left(((p-m)\log(p-m-1)+m) \times D\right).$$

Enfin, le nombre de vols locaux est inférieur à :

$$O\left(\left(\frac{p-m}{1-v}\log(p-m-1) + \frac{m}{1-v}\right) \times vD\right).$$

Démonstration. Durant l'exécution, les processeurs sont soit entrain de voler un processeur soit de réaliser l'exécution d'une tâche. Le temps passé à exécuter des tâches est inférieur à W. Le lemme 7.1 montre que le nombre de vol est quant à lui inférieur à :

$$O\left(\left(\frac{p-m}{(1-v)}\log(p-m-1) + \frac{m}{(1-v)}\right) \times D\right).$$

Nous obtenons ainsi que :

$$T_p \leqslant \frac{W + O\left(\left(\frac{p-m}{(1-v)}\log(p-m-1) + \frac{m}{(1-v)}\right) \times D\right)}{p}$$

De plus, le nombre de vols globaux est égal au nombre total de vols multiplié par la probabilité de voler à l'extérieur du groupe. De manière similaire, nous obtenons que le nombre de vols locaux est égal au nombre de vols total multiplié par la probabilité de voler à l'intérieur du groupe. □

Au final, le temps d'exécution est légèrement plus élevé que pour le vol de travail classique, car l'impact du chemin critique est ici multiplié par un facteur $\log(p)$. De plus, nous sommes capables de quantifier l'impact de la probabilité de vol distant sur le temps de complétion. Notre analyse ne met néanmoins malheureusement pas en évidence de réduction du nombre de vols distants par rapport au vol de travail classique.

7.1.2 Analyse de HWS

L'analyse théorique de HWS s'avère complexe, et a nécessité l'ajout de notre part de différentes restrictions sur le comportement de l'algorithme.

Lors de la présentation de l'heuristique, le maître peut fournir du travail à son groupe de différentes manières (attente de la fin du travail, détection de manque de travail).

Lorsque le maître donne du travail à son groupe, il fournit une des tâches créées juste en dessous de la limite. Durant son exécution, cette tâche génère d'autres tâches

de profondeur plus grande. Nous définissons un bloc de tâches comme l'ensemble des tâches créées par l'intermédiaire d'une tâche fournie par le maître. Dans notre analyse théorique, nous nous limitons au cas où le maître attend la fin de l'exécution du bloc fourni au groupe.

Cette gestion est celle qui introduit le plus de temps d'inactivité entre l'exécution de deux blocs par le même groupe. Ainsi cette restriction ne peut que dégrader les performances de HWS.

Notons enfin qu'avec la version de l'algorithme utilisée pour la preuve, les attentes du maître peuvent provoquer des inter-blocages lors de l'exécution. La figure 7.1 montre une application avec des dépendances entre blocs. Dans une exécution avec deux groupes, le premier groupe exécute le bloc 1. Le second exécute le bloc 3. Aucun de ces deux groupes ne peut finir l'exécution du bloc. Pour débloquer la situation, un des deux groupes doit exécuter le bloc 2. Avec l'algorithme classique, le nombre de vols au sein d'un groupe augmentera, provoquant l'exécution du bloc 2 et évitant le blocage.

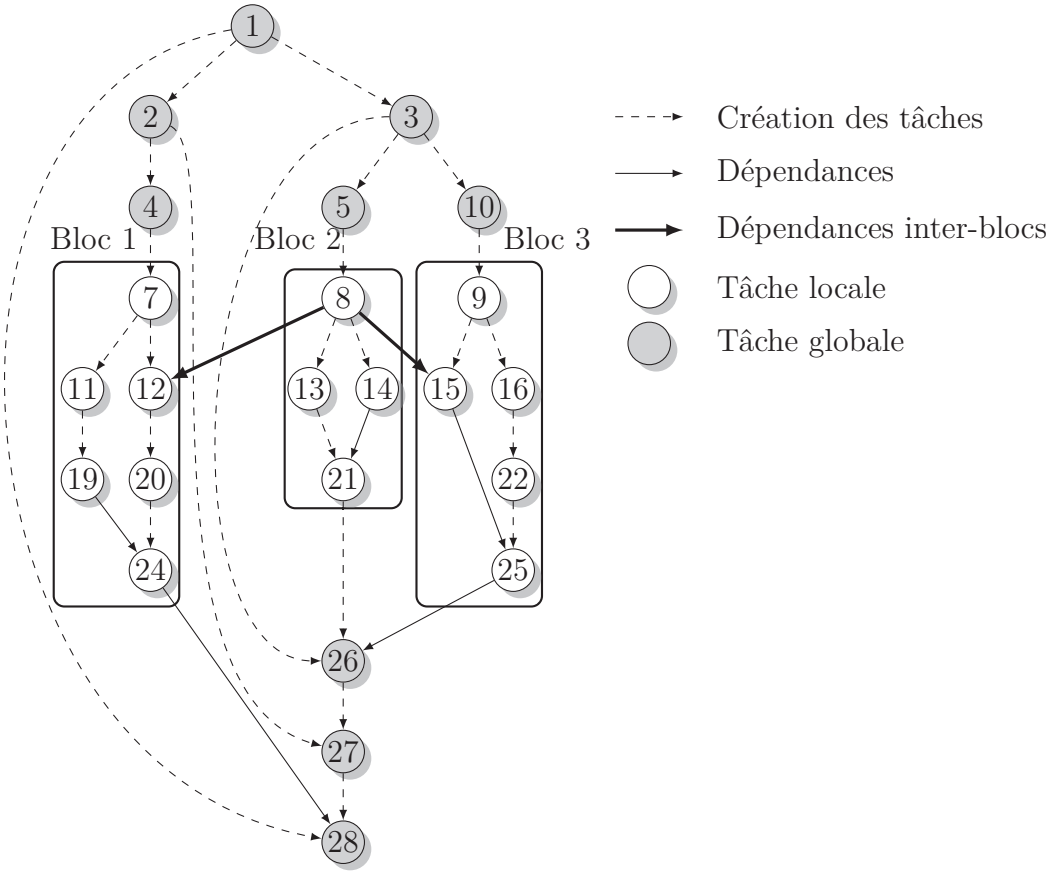


FIGURE 7.1 – Exemple de DAG avec des dépendances inter-blocs.

Pour éviter cette configuration, nous limitons notre démonstration à des applications sans dépendances entre blocs. Cette restriction permet néanmoins d'exécuter des classes de DAG comme «Série Parallèle». Ces DAG permettent de représenter par exemple les application de type «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, nous bornons le temps d'exécution d'un bloc exécuté par un groupe. Cette borne, nous permet alors de déduire le temps d'exécution de l'application globale.

a Modélisation

Nous utilisons des résultats classiques montrés précédemment par Blumofe et Leiserson [9]. Pour utiliser ces résultats, nous devons utiliser la représentation habituelle de l'application. Nous utilisons ici, un DAG G de tâches unitaires dont les arêtes définissent les relation de dépendance et de création entre les tâches. Ce type de DAG ont été détaillés section 1.3

Afin de décrire l'application, nous utiliserons les notations classiques du domaine [8]. Nous rappelons ces notations :

W **:** Le nombre de nœuds dans G.

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

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

Pour la stratégie classique du vol de travail, 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 $\frac{W}{p} + O(D)$.

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 longueur du chemin critique en-dessous de la limite.

L'indice u désigne les ensembles au dessus de la limite («up»), et l'indice d désigne les ensembles en dessous de la limite («down»).

Chaque tâche locale appartient à un bloc, et un bloc ne contient que des tâches locales. L'ensemble des blocs est caractérisé par :

W_i **:** Le nombre de nœuds 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 7.2 montre la représentation d'une application sans dépendance entre les blocs avec une limite fixée à 3 et le regroupement en bloc de tâches.

Par définition des blocs :

$$W_d = \sum_i W_i = W - W_u$$

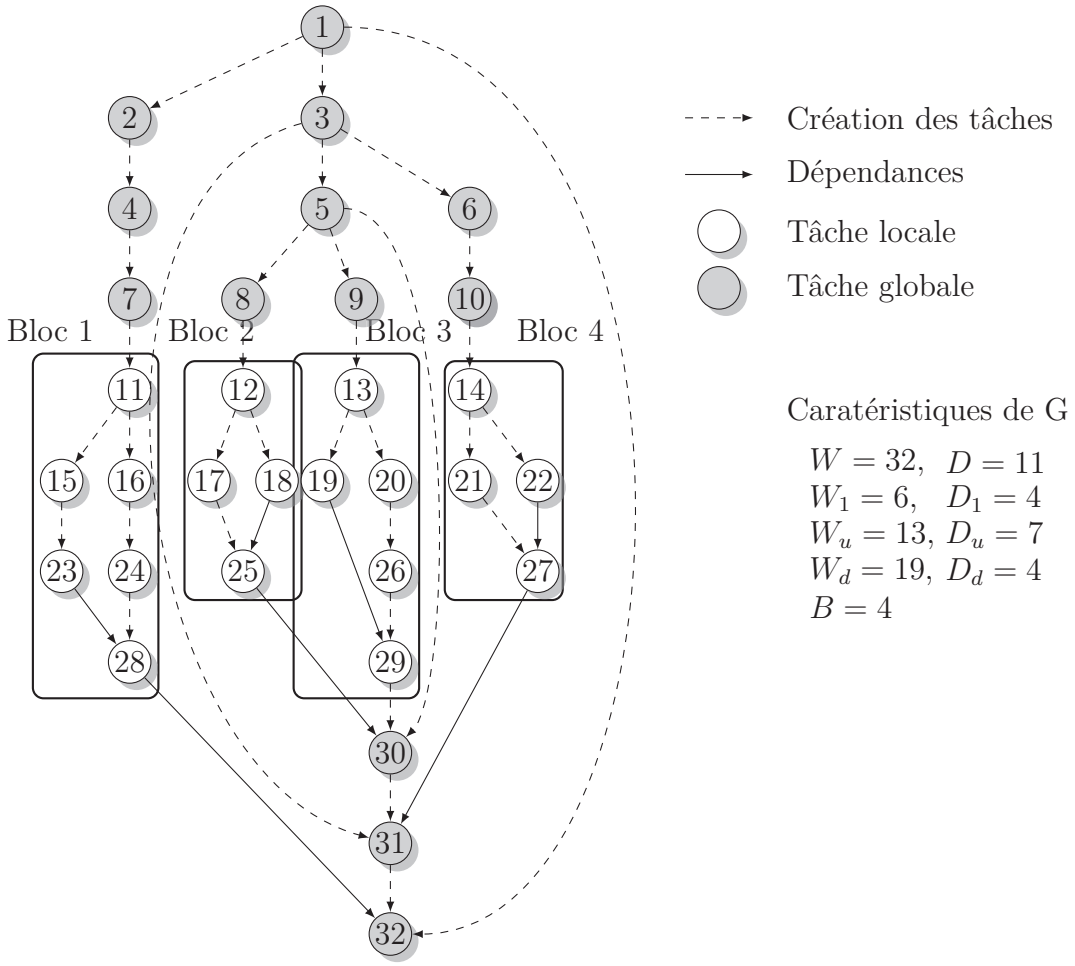


FIGURE 7.2 – Graphe G représentant une application.

$$D_d = \max_i D_i$$

L'application modélisée est exécutée par une plate-forme de p processeurs (unités de calcul) divisés en p_u groupes de machines. Dans cette démonstration, nous considérons un nombre de machines identique par groupe. Ainsi, chaque groupe k de machines contient p_d machines avec $p_d = \frac{p}{p_u}$.

b Preuve théorique

Par souci de clarté nous considérons, dans un premier temps, que le temps d'exécution d'une application avec le vol de travail est inférieur ou égal à $\frac{W}{p} + O(D)$. Nous affinerons les résultats obtenus dans le théorème 7.4 en prenant en compte plus finement les probabilités liées à cette borne.

Théorème 7.3. *Le temps d'exécution de l'application avec l'heuristique HWS est inférieur à* $\frac{W_u}{p_u} + \frac{W_d}{p} + O(D\frac{B}{p_u} + D + \max_i(\frac{W_i}{p_d}))$ *et le nombre de vols réalisés entre les groupes est* $O(p_u \times (D + \max_i(\frac{W_i}{p_d})))$.

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 son groupe. Son temps d'exécution peut donc être borné par $\frac{W}{p} + O(D)$. Le maître étant en attente de la fin de l'exécution, l'exécution du bloc peut être vue par le maître comme l'exécution d'une chaîne de tâches de longueur $\frac{W}{p} + O(D)$.

Pour obtenir une vision d'ensemble de l'exécution, nous construisons un DAG G' à partir de G. G' représente l'application perçue par les maîtres. Toutes les tâches de G créées au-dessus de la limite sont conservées 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 $\frac{W_i}{p_d} + O(D_i)$ dans G'. La figure 7.3 montre un exemple de G'. L'exécution du DAG G' par les maîtres sera équivalente en temps à l'exécution de G sur l'ensemble de la plate-forme. Nous analysons donc l'exécution de G' par les maîtres pour borner le temps d'exécution. Les caractéristiques de G' sont les suivantes :

$$D' \leq D'_u + D'_d = D_u + \max_i \left(O(D_i) + \frac{W_i}{p_d} \right)$$

$$\Rightarrow D' \leq O(D) + \max_i \frac{W_i}{p_d} \tag{7.1}$$

$$W' = W'_u + W'_d = W_u + \sum_i \left(\frac{W_i}{p_d} + O(D_i) \right)$$

$$\Rightarrow W' \leq W_u + \frac{W_d}{p_d} + O(B \times D_d) \tag{7.2}$$

Le mécanisme d'équilibrage de charge entre maîres étant également du vol de travail, le temps d'exécution de l'application est donc borné par $\frac{W_u}{p_u} + \frac{W_d}{p} + O(D\frac{B}{p_u} + D + \max_i \frac{W_i}{p_d})$. De la même façon, le nombre de vols entre les maîtres est borné par $O(p_u \times (D + \max_i \frac{W_i}{p_d}))$, ce qui conclut la preuve de 7.3. □

En réalité néanmoins, le vol de travail étant un algorithme probabiliste, les analyses classiques ne garantissent pas exactement un temps d'exécution de $\frac{W}{p} + O(D)$ mais le garantissent inférieur à $\frac{W}{p} + O(D) + \log(\frac{1}{\epsilon})$ avec une probabilité supérieure à $1 - \epsilon$.

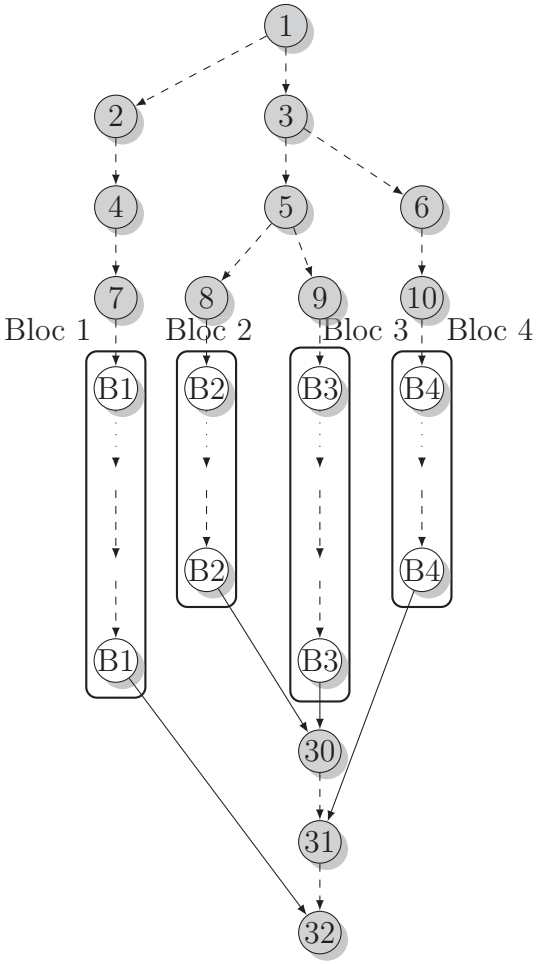
Or, en considérant que le temps d'exécution d'un bloc de tâches est une variable aléatoire, les caractéristiques de G' changent.

Théorème 7.4. *Pour tout $X > 1$, le temps d'exécution de l'application avec l'heuristique HWS est inférieur à :*

$$\frac{W_u}{p_u} + \frac{W_d}{p} + O\left([D + l - \log(\log(X))] \frac{B}{p_u} + D + \max_i(\frac{W_i}{p_d}) \right) + \log\left(\frac{X}{X-1} \right),$$

avec une probabilité supérieure à $\frac{1}{X}$ et l la limite choisie par l'utilisateur. De plus, le nombre de vols réalisés entre les groupes est $O(p_u \times (D + \max_i(\frac{W_i}{p_d})))$.

Démonstration. Nous considérons les DAG G et G' du théorème 7.3. La différence avec les DAG précédent est que les temps d'exécution des blocs de tâches sont inférieurs ou égaux à $\frac{W_i}{p} + O(D) + log(\frac{1}{\epsilon})$ avec une probabilité supérieure à $1 - \epsilon$ [3].



Caractéristiques de G'

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

FIGURE 7.3 – Graphe représentant G', une abstraction de G (Figure 7.2) pour une plate-forme donnée.

Le chemin critique D' est égal à $D_u + \max_i(D'_i) + log(\frac{1}{\epsilon})$ avec une probabilité supérieure à $1 - \epsilon$. Ceci provient du fait que : si un ensemble de variables aléatoires qui sont inférieures à une valeur V avec une probabilité supérieure à r, la valeur maximale de ces variables est inférieure à V avec une probabilité supérieure à r. La longueur de chacun des blocs est inférieure ou égale à $\frac{W_i}{p} + O(D) + log(\frac{1}{\epsilon})$.

Dans la formule, les longueurs des blocs sont ajoutées. La probabilité d'avoir W' inférieur ou égal à $W_u + \frac{W_d}{p_d} + O(B \times D_d) + B \times log(\frac{1}{\epsilon})$ est supérieure à $(1-\epsilon)^B$. Nous posons cette probabilité égale à $\frac{1}{X}$:

$$\epsilon = 1 - \frac{1}{\sqrt[B]{X}}$$

$$\log\left(\frac{1}{\epsilon}\right) = \log\left(\frac{1}{1 - \frac{1}{\sqrt[B]{X}}}\right) = -\log\left(1 - \frac{1}{\sqrt[B]{X}}\right) = -\log\left(1 - e^{-\frac{1}{B}log(X)}\right)$$

Pour X plus grand que 1 et B grand, nous obtenons par développement limité que :

$$1 - e^{-\frac{1}{B}\log(X)} \approx \frac{1}{B} \times \log(X) + O\left(\frac{(\log(X))^2}{B^2}\right)$$

En remplaçant cela dans la formule précédente :

$$log(\frac{1}{\epsilon}) \approx \log(B) - \log(\log(X)) + O(\log(B))$$

Nous avons que $\log(\log(X))$ tend vers l'infini quand X tend vers 1. Pour avoir une probabilité supérieure à 90%, $\log(\log(X))$ est inférieur à 3. De plus, B est une valeur qui est inférieure à 2^l avec la limite l choisie par l'utilisateur. Ainsi, $\log(B)$ est inférieur ou égal à l ($l < D$). Ainsi :

$$W' = W_u + \frac{W_d}{p_d} + O(B \times D_d) + B \times (-\log(\log(X)) + O(l))$$

Finalement :

$$T_p \leq \frac{W_u}{p_u} + \frac{W_d}{p} + O\left([D + l - \log(\log(X))]\frac{B}{p_u} + D + \max_i(\frac{W_i}{p_d})\right) + \log\left(\frac{X}{X-1}\right) \quad (7.3)$$

La probabilité que le temps d'exécution soit inférieur à l'équation 7.3 est supérieure ou égale à $\frac{1}{X}$. □

c Analyse des résultats

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

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 s'ils résultent du jeu de calcul mathématique.

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

$\frac{W_u}{p_u}$ **:** Le travail réservé au dessus de la limite doit être exécuté par les maîtres. Notre algorithme 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 est une borne inférieure du temps d'exécution, nous ne pouvons pas avoir un temps inférieur à cela.

D **:** L'impact classique du chemin critique.

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

$D\frac{B}{p_u}$ **:** 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 groupe 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 sur le problème réel. Enfin, la quantité de vols «longues distances» réalisés se trouve fortement réduite, passant de $O((p - \min_i(p_i)) \times D)$ (au pire cas, les vols sont effectués par le groupe de machines le plus petit) pour l'algorithme classique à $O(g \times (D + \max_i(\frac{W_i}{p_d})))$.

7.2 Analyse expérimentale

L'analyse théorique de HWS permet de mettre en avant une réduction du nombre de tâches transférées entre deux groupes de machines. Puisque les liens de communications sont généralement moins performants, nous pensons que cette réduction permettra en pratique de minimiser l'impact des communications sur le temps d'exécution. En revanche, pour PWS, l'analyse théorique que nous avons réalisée ne montre pas de réduction du nombre de transferts par rapport au vol de travail classique. Mais intuitivement, le changement de probabilité devrait impacter le nombre de transferts dans la plupart des cas.

Ainsi, nous analysons les performances obtenues par ces algorithmes en pratique. Nous commençons par montrer l'efficacité de nos approches avec un environnement d'exécution de petite taille dans la section 7.2.1. Puis dans la section 7.2.2, nous analysons les performances sur une plate-forme à grande échelle.

7.2.1 Environnement contrôlé

Nous comparons dans cette section les algorithmes PWS et HWS avec les algorithmes classiques de la littérature : le vol de travail classique (noté Work-Stealing (WS)), l'algorithme de vol de travail implanté dans *Kaapi* (noté KWS), et enfin l'algorithme Cluster-aware Random Stealing (CRS) introduit par *Satin*. Ces algorithmes ont été

présentés dans la section 3.3.2. Nos expériences montrent avec des exemples pratiques que nos algorithmes réduisent le temps d'exécution et la quantité de communications.

Ces expériences ont été réalisées pour comparer les algorithmes existant à nos algorithmes PWS et HWS, mais aussi pour montrer les perturbations réseaux subies sur plusieurs types d'applications. Les expériences sont réalisées sur une nombre réduit de processeurs, ce qui nous permet une compréhension très fine des comportements des différents algorithmes. Elles sont un premier pas pour réaliser de futures expériences à grande échelle.

Pour réaliser ces expériences, nous avons utilisé la plate-forme d'expérimentation Grid'5000. La machine à mémoire partagée avec le plus d'unité de calcul a deux processeur Intel Xeon E5420 composé chacun de quatre cœurs fonctionnant à 2.5 GHz.

Cette plate-forme permet d'avoir un temps de référence sans transfert de données sur huit unités de calculs. Ce temps pourra être comparé à l'exécution sur huit unités de calcul séparées équitablement sur deux machines.

Nous utilisons le réseau *InfiniBand* [58] comme support de communications entre les machines. Pour les exécutions utilisant ce réseau, nous avons prêté une attention particulière à éviter les perturbations en choisissant des machines connectées sur le même commutateur.

Cette plate-forme composée de deux machines connectées par un réseau *InfiniBand* est hiérarchique car les quatre unités de calcul présentes sur la même machine communiquent plus rapidement entre elles.

Ces expériences nous permettront de détailler le comportement des algorithmes sans perturbations. Afin d'analyser finiment ces résultats et de pouvoir comparer les performances des algorithmes, les expériences ont été reproduites cent fois pour chaque point. Les intervalles de confiances sont calculé à 95%. Ils sont affichés sur certaines courbes uniquement dans le but d'améliorer la lisibilité de certaines courbes.

Nous proposons une analyse de l'ensemble des algorithmes d'ordonnancement par vol de travail sur deux applications différentes.

La première application compte le nombre de solutions dans le problème des N-reines. Dans ce problème, l'objectif est de placer une reine par ligne sur l'échiquier sans qu'aucune soit ni sur la même ligne, ni sur la même colonne, ni sur la même diagonale. La figure 7.4 montre une solution avec un échiquier de taille 5×5. Cette application crée de nombreuses tâches mais elles transfèrent peu de données.

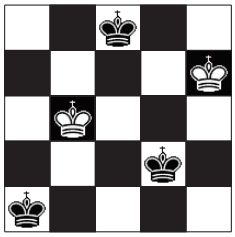


FIGURE 7.4 – Une solution du problème des N-Reines (5×5).

La seconde est une implantation de l'algorithme de tri-fusion. Dans cette application, nous trions un tableau de doubles contenant quatre gigaoctets de données. Nous avons choisi cet exemple car chaque tâche contient un grande quantité de données à transférer.

Avec ces deux applications, nous mettons en avant la prise en compte de la hiérarchie pour les communications avec le tri-fusion, et le sur-coût potentiel lié à la gestion des tâches dans l'application des N-reines.

Les sections a et b présentent respectivement les expériences sur l'application solvant le problème des N-reines et l'application de tri-fusion.

a N-Reines

Dans ce lot d'expériences, le problème des N-reines est résolu par une approche récursive. En premier, le programme crée une tâche pour chaque position possible de la première reine sur la première ligne. En fonction de ce choix, ces tâches créeront d'autres tâches pour chaque placement possible de la seconde reine et ainsi de suite. Lorsque cinq reines ont été placées, l'algorithme calcule en séquentiel le résultat. À la fin de l'exécution, le programme fournit le nombre de placements différents permettant de résoudre le problème.

Dans notre bibliothèque *Kaapi*, lors d'un vol, le processeur prend une des tâches les plus anciennes. Pour cette application, un gain de performance est possible en volant la moitié des tâches. Pour combler ce manque, nous avons réalisé une découpe binaire récursive des tâches lorsque leur nombre était supérieur à 2.

Les données associées a chaque tâche représentent l'échiquier qui est codé sur N^2 bits. Ainsi, les données à transférer sont peu volumineuses. En revanche, le nombre de tâches est grand. L'instance résolue ici est celle de taille dix-huit. Il y a un peu plus d'un million et demi de tâches. Dans ces conditions, les heuristiques doivent permettre d'équilibrer la charge avec un faible sur-coût.

La figure 7.5 montre le temps d'exécution pour chacune des heuristiques utilisant huit unités de calcul réparties sur deux machines. De plus, deux temps de références sont donnés :

– le temps d'exécution sur quatre unités de calcul se situant sur une seule machine.
– le temps d'exécution sur huit unités de calcul se situant sur une seule machine.

Il est possible d'observer que les heuristiques ont un temps d'exécution proche les unes des autres. Ainsi, nous constatons que les heuristiques hiérarchiques permettent un équilibrage de la charge efficace lorsqu'il y a peu de données et de nombreuses tâches.

La figure 7.6 montre l'évolution du temps d'exécution en fonction de la probabilité de voler une machine externe au groupe dans PWS. Le temps d'exécution augmente légèrement lorsque la probabilité est trop faible dans cet exemple. Cette augmentation est due à un manque de travail au sein d'un des deux groupes.

Pour HWS, l'utilisateur fournit une limite permettant de différencier les tâches locales et globales. La figure 7.7 montre l'évolution du temps d'exécution en fonction de limite choisie. En augmentant la limite l, le nombre de tâches réalisées par le maître augmente en 2^l. Cette figure montre que le temps d'exécution est stable pour une limite inférieure à 10 (environs 1000 tâches). En augmentant le nombre de maîtres, la limite

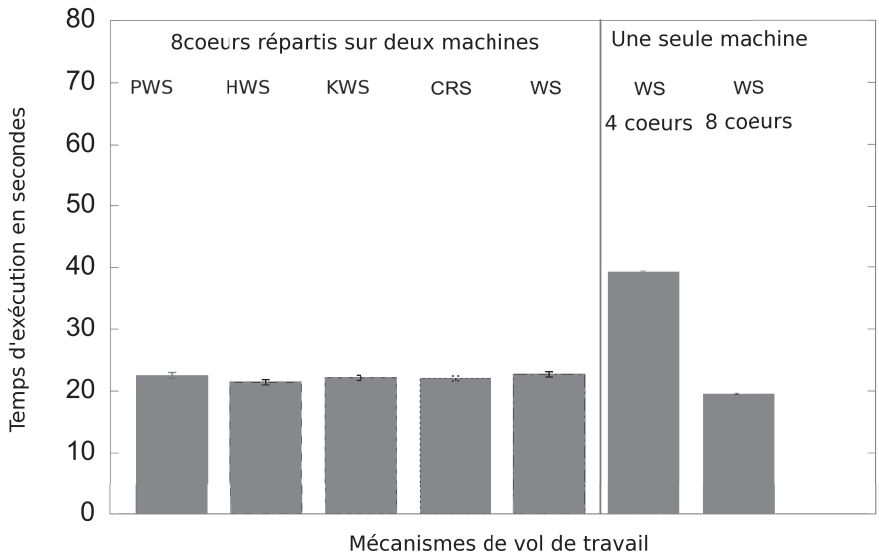


FIGURE 7.5 – Temps d'exécution pour résoudre le problème des N-Reines avec différents mécanismes de vol de travail en mémoire partagé ou distribuée.

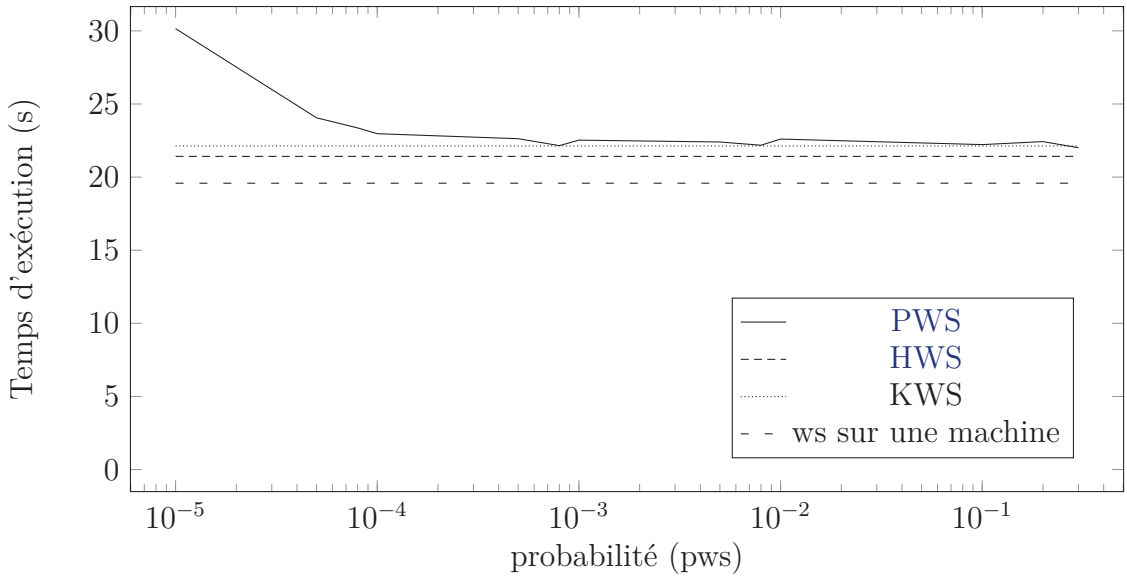


FIGURE 7.6 – Évolution du temps d'exécution sur 8 unités de calculs sur le problème des N-Reines en fonction de la probabilité choisie pour l'heuristique PWS.

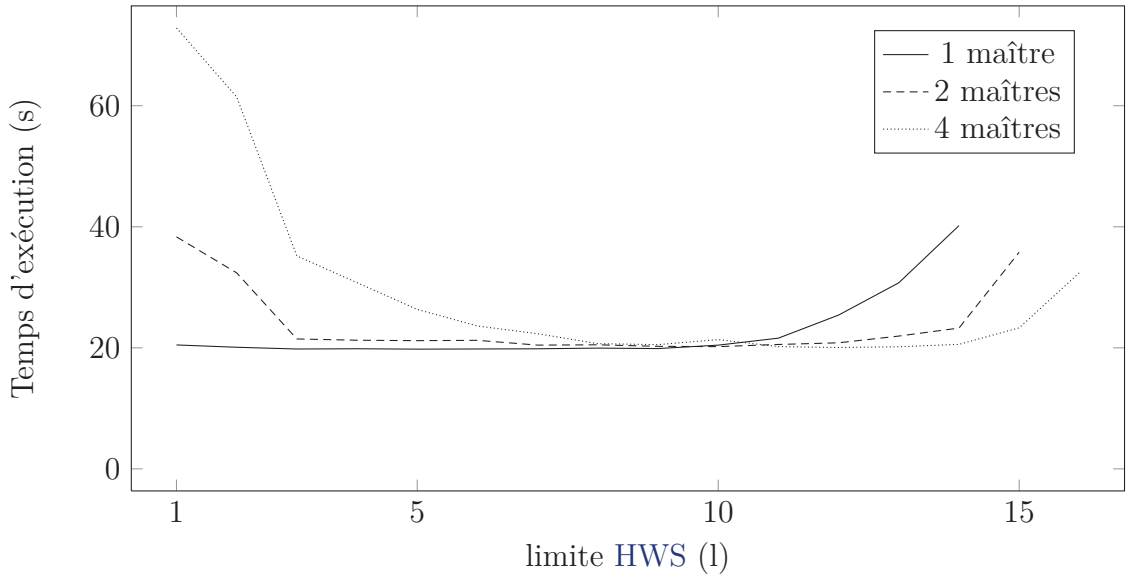


FIGURE 7.7 – Évolution du temps d'exécution sur 8 unités de calculs en fonction de la limite pour l'heuristique HWS.

peut être choisie plus grande, car l'exécution de ces tâches est répartie entre les maîtres. En revanche, lorsque la limite est faible, le nombre de tâches à répartir entre les maîtres est trop faible. Le temps d'exécution est donc plus important. La limite de HWS doit ainsi être choisie pour fournir suffisamment de travail aux maîtres.

b Tri-Fusion

Nous analysons le temps d'exécution de chaque algorithme basé sur le vol de travail pour trier un tableau de quatre gigaoctets sur une plate-forme distribuée.

L'implantation du tri-fusion utilise une découpe récursive du tableau. À chaque étape, l'algorithme crée deux tâches réalisant le tri de la moitié des données. En plus de ces deux tâches, une tâche de fusion est créée. La fusion est effectuée en place avec l'algorithme implanté dans la bibliothèque *Standard Template Library (STL)* [78]. Lorsque les données ont une taille inférieure à quatre kilo-octets, le travail est effectué en séquentiel avec l'algorithme de tri présent dans la *STL*.

L'allocation du tableau sur chacune des machines est effectué en début d'exécution. Lors des transferts de données, les éléments sont copiés dans le tableau pré-alloué. Le sur-coût de cette implantation de l'algorithme avec *Kaapi* sur une unité de calcul est inférieur à trois pour cent en comparaison au temps obtenu avec l'algorithme de la *STL*. Pour chacune des mesures, l'intervalle de confiance affiché est calculé sur une centaine d'expériences.

La figure 7.8 illustre une comparaison entre les algorithmes PWS, HWS et certaines politiques de vol de travail existantes.

Cette figure met en évidence que le vol de travail classique (WS) et l'algorithme de *Satin* (CRS) obtiennent un temps d'exécution plus important en utilisant deux

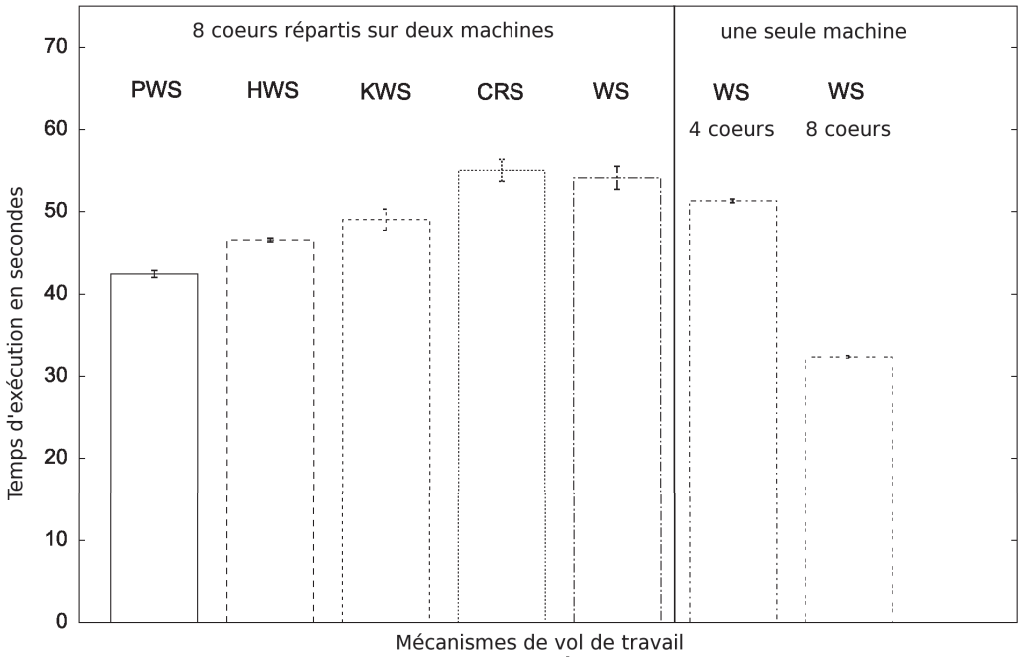


FIGURE 7.8 – Temps d'exécution pour trier un tableau de 4 Giga-octets avec différentes mécanismes de vol de travail en mémoire partagée ou distribuée.

processeurs distribués sur deux machines que le temps d'exécution sur un processeur. Ces algorithmes n'arrivent pas à tirer parti de la puissance de calcul supplémentaire.

L'allongement du temps d'exécution est dû aux transferts de données effectuées. Ces deux algorithmes transfèrent plus de huit gigaoctets de données durant l'exécution. Cette quantité de données est importante mais provient de l'implantation dans *Kaapi* qui transfère les données au moment du vol et non au moment de l'accès.

Cela montre aussi que le nombre de tâches transférées entre les deux machines est important. Il est donc difficile de prévoir quel processeur exécutera quelle tâche. Aussi, si les données étaient transférées lors de l'accès, il y aurait de nombreux temps d'attente.

Pour l'algorithme CRS, ces résultats sont en adéquation avec ceux présentés dans [85]. Les expériences effectuées sur CRS montrent que l'algorithme fournit des performances équivalentes à celles du vol de travail classique lorsque l'application a beaucoup de données comme dans la multiplication de matrices.

L'algorithme de *Kaapi* permet de trier le tableau en un temps inférieur à celui sur un processeur dans soixante pour cent des exécutions. Dans cet algorithme, le processeur tente de voler un des processeurs sur la même machine. Si ce vol échoue, il vole un processeur sur une machine distante. Ainsi, la quantité de données transférées est réduite à un peu moins de six gigaoctets.

Enfin, nos deux algorithmes PWS et HWS permettent de profiter de la puissance supplémentaire disponible qui représente ici la moitié de la puissance totale. Les quantités de données transférées sont inférieures à quatre gigaoctet. Ceci est équivalent à transférer la moitié du tableau sur l'autre machine puis à rapatrier les résultats. La limite utilisée

pour HWS est égale à deux. Pour PWS, nous avons choisi la probabilité qui minimise le temps d'exécution. Sur ces expériences, ces deux algorithmes permettent d'obtenir un temps d'exécution significativement plus faible.

Pour détailler le choix de la probabilité dans PWS, la figure 7.9 montre l'évolution du temps d'exécution en fonction de la probabilité de voler un processeur sur l'autre machine. Lorsque la probabilité est égale à un demi, le temps d'exécution est le même que celui du vol de travail classique. En diminuant la probabilité, le temps d'exécution diminue. Il est équivalent à celui de HWS lorsque la probabilité est inférieure à un dixième. De plus, ce temps reste stable lorsque la probabilité diminue. Ainsi, l'utilisateur peut choisir aisément la probabilité pour cette application.

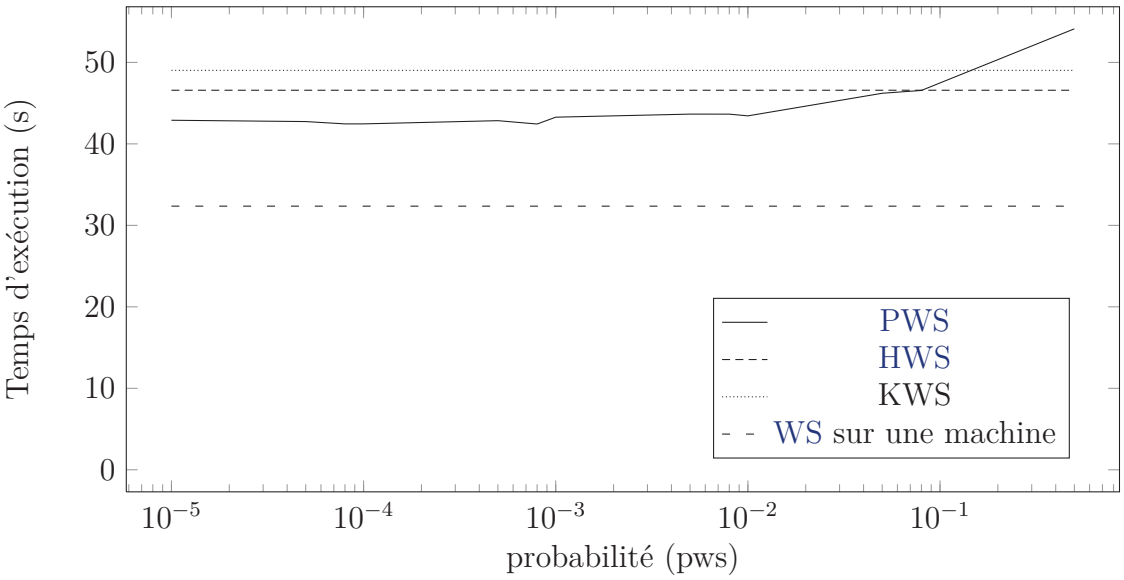


FIGURE 7.9 – Évolution du temps d'exécution sur 8 unités de calculs pour trier un tableau de 4 Giga-octets en fonction de la probabilité choisie pour l'heuristique PWS.

En comparant l'évolution du temps d'exécution pour cette application et celle obtenue avec l'application des N-reines, nous remarquons que les deux évolutions sont opposées. L'utilisateur devra choisir cette probabilité en fonction de l'application. Lorsque l'application transfère peu de données, la probabilité ne devra pas être choisie trop faible. Mais l'intervalle de sélection de la probabilité est assez large dans les deux cas.

Pour HWS, l'application de tri-fusion découpe le travail en deux de manière identique au problème des N-reines. En revanche, la quantité de travail laissée aux maîtres est plus importante car la fusion des tableaux est plus gourmande en calcul que la somme réalisée dans les N-reines. Ainsi, le temps d'exécution augmente légèrement plus vite.

7.2.2 Plate-forme Non-Uniform Memory Access (NUMA)

Après avoir réalisé des expériences sur une petite plate-forme distribuée, nous proposons de tester nos algorithmes sur une plate-forme plus grande. Nous avons donc

réalisé des expériences sur une plate-forme NUMA ayant un nombre de cœurs important. Sur ce type de machine, les processeurs ont accès à différents bancs mémoire. En fonction du processeur et du banc mémoire ayant la donnée, le temps d'accès varie.

Cette plate-forme est composée de 48 cœurs répartis sur quatre processeurs de 12 cœurs. Chaque processeur est formé de deux sous groupes de 6 processeurs. Les processeurs sont des AMD Opteron fonctionnant à une fréquence de 2.2 GHz. Il y 8 bancs mémoires de 32 Go chacun. Un banc mémoire est relié de manière privilégié avec un sous groupe de 6 cœurs.

Pour ces expériences, nous avons utilisé l'application tri-fusion décrite dans la section 7.2.1. Comme nous l'avons détaillé précédemment dans *Kaapi*, les transferts de données sont effectué au moment du vol. Ainsi lors du vol d'une tâche, l'ensemble des données accédées par le sous-DAG sont transférées. De plus à la fin de l'exécution, les données sont renvoyées au processeur volé initialement même si cela n'est pas nécessaire.

Pour HWS entre les maîtres, nous souhaiterions que les données soient envoyées en même temps que le maître effectue la découpe. Dès que les données du premier bloc sont présentes, le maître fourni le travail à ces esclaves. Pour éviter l'envoi direct de la moitié du tableau, nous avons utilisé une découpe initiale en 8.

Dans ces expériences, nous distinguons deux types d'exécution :
– au sein d'un processus (un seul espace mémoire).
– au sein de deux processus (deux espaces mémoire).
Pour les expériences au sein d'un processus, il est nécessaire de répartir les données entre les différents bancs mémoires pour y accéder plus rapidement. Il existe différentes méthodes de placement des données sur plate-forme NUMA comme celles détaillées dans [65]. Généralement, elles ne tiennent pas compte d'informations provenant de l'application. Nous avons utilisé le placement «interleave» fourni par la librairie numactl [46]. Le placement est réalisé avec un algorithme «Round-Robin».

En revanche, lors de l'utilisation de deux processus, chaque espace mémoire est placé sur le banc mémoire le plus proche du processeur utilisé. Les communications sont réalisées au travers d'une socket réseau comme si les deux processus étaient sur des machines distantes. Cela rajoute un sur-coût dans le transferts des données. Nous utilisons deux processus différents pour pouvoir fournir le découpage en groupes aux heuristiques hiérarchiques de vol de travail.

La première série d'expériences a été réalisée sur 12 cœurs. La plate-forme est découpée en deux groupe de 6 cœurs. Les 6 cœurs appartiennent au même sous groupe au sein d'un processeur. Les deux groupes sont répartis sur deux processeurs différents.

La figure 7.10 compare l'exécution de l'application tri-fusion sur différentes heuristiques. Une de ces heuristiques utilise le vol de travail classique au sein d'un processus utilisant 12 unité de calculs avec les placements «round-robin» des données. Les autres heuristiques (HWS, PWS, KWS, WS, CRS) utilisent deux processus utilisant chacun 6 unité de calculs.

La seconde série fonctionne de la même façon mais avec 24 cœurs au total. Cet comparaison est illustrée par la figure 7.11.

Sur ces deux expériences, les deux algorithmes hiérarchiques HWS et PWS obtiennent des performances meilleures que celle du vol de travail classique au sein d'un processus.

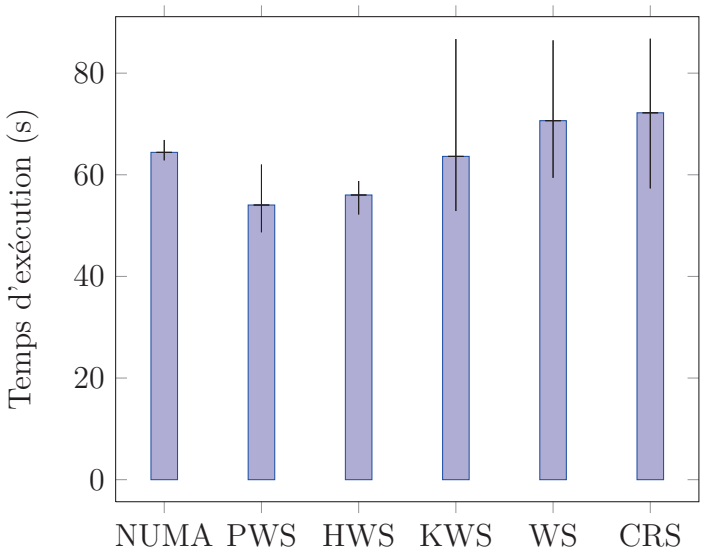


FIGURE 7.10 – Évolution du temps d'exécution sur 12 unités de calculs divisé en deux groupes (barre d'erreur : la valeur minimale et maximale sur 30 exécutions).

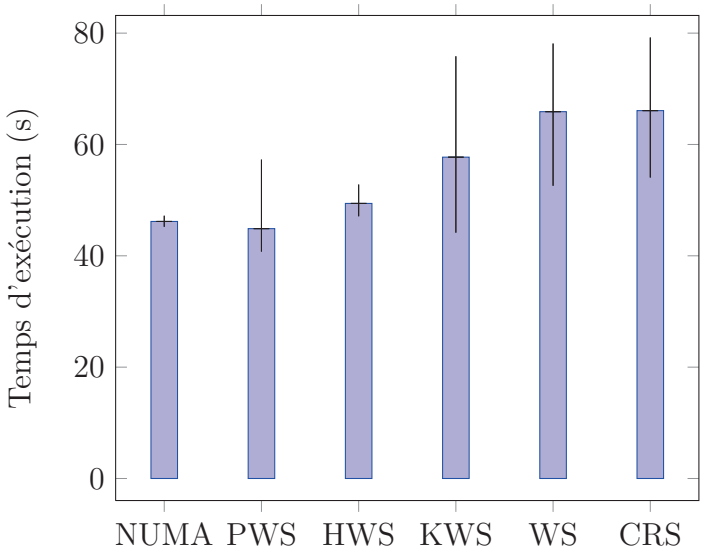


FIGURE 7.11 – Évolution du temps d'exécution sur 24 unités de calculs divisé en deux groupes (barre d'erreur : la valeur minimale et maximale sur 30 exécutions).

Pourtant, ces algorithmes utilisent des processus communiquant au travers de «sockets» réseaux. Il serait possible d'améliorer leurs performances sur une telle architecture en introduisant une gestion de la hiérarchie à l'intérieur d'un processus. Cela éviterait aussi les problèmes liés à la gestion des transferts de données dans ce cas.

Malgré cela, les algorithmes HWS et PWS améliorent les performances par rapport à l'exécution sur un processus. Ces expériences montrent que les algorithmes hiérarchiques ne sont pas limités à une utilisation de machines connectées par des liens réseaux. Ainsi, cela confirme les résultats obtenus sur deux machines de quatre cœurs présentés dans la section 7.2.1 mais avec une puissance de calcul plus importante.

7.3 Conclusion

Dans cette partie, nous avons introduit deux nouveaux algorithmes de vol de travail : PWS et HWS. Ces algorithmes nous permettent de prendre en compte la hiérarchie d'une plate-forme pour minimiser le temps d'exécution.

Nous avons détaillé une analyse théorique pour chacun d'entre eux. L'analyse de HWS met en avant la réduction du nombre de transferts entre les groupes de machines. L'analyse PWS met en évidence que le temps d'exécution est proche de celui obtenu par le vol de travail classique. Notre analyse ne met maheureusement pas en évidence de réduction du nombre de vols distants. Nous pensons que ceci est lié à la manière dont est déterminée la borne supérieure sur les probabilités de vols «inutiles» et non à l'algorithme en lui même. Une analyse plus fine pourrait ainsi éventuellement permettre de mettre en évidence le gain de performance visible en pratique.

Nous avons également validé expérimentalement nos algorithmes par une campagne de tests réels. Dans certaines configurations, nous sommes ainsi en mesure de réduire le temps d'exécution de plus de 20 pour cent par rapport au vol de travail classique.

Pour aller plus loin avec ces algorithmes, nous détaillons les points qu'il pourrait être intéressant d'approfondir.

Le principale avantage de PWS est que sa mise en œuvre est simple. La seule modification par rapport au vol de travail est sur le mécanisme de sélection de la victime. De plus dans les expériences réalisées, nous avons pu constater que le choix de la probabilité peut être réalisé dans une plage assez conséquente.

Pour HWS, l'implantation est quant à elle plus complexe. Elle nécessite de changer l'ordre d'exécution des tâches et le choix de la tâche volée.

Nous avons vu au travers de la section 7.2.2, que HWS peut s'adapter à différents niveaux de la hiérarchie (grappe de calcul, machine multiprocesseurs). Cet algorithme pourrait avoir d'autres avantages qui n'ont pas encore été explorés.

Par exemple, le vol de travail fonctionne bien sur CPU lorsque les tailles des tâches sont petites. En revanche, certaines architectures de processeurs comme les GPU nécessitent des tâches de tailles importantes pour profiter de leur puissance de calcul. Pour ordonnancer des tâches sur GPU et CPU, la taille des tâches réalisant le calcul doit être adaptée à l'architecture. En plaçant un maître par type d'architecture, nous

pouvons penser à faire varier le niveau de découpe dans chacun des groupes en fonction de l'architecture.

Une seconde utilisation de cet algorithme est sur une plate-forme hétérogène composée d'une grille de calcul et de machines reliées par internet avec une connexion haut débit. Lors d'expériences à grande échelle, nous avons remarqué qu'en utilisant deux grappes de calcul de grid'5000 et des machines de la plate-forme dsllab [30], le temps d'exécution est plus important en ajoutant des machines de dsllab même sur une application intensive en calcul comme les N-reines. Les processeurs perdent du temps en communiquant avec les machines de dsllab. En utilisant HWS, les maîtres fourniront moins de travail aux machines de dsllab car elles nécessitent plus de temps pour exécuter un bloc de tâches. Ainsi, les machines de la grille de calcul ne perdront pas de temps à essayer de voler des machines connectées par internet.

Sur ce type de plate-forme, le placement des maîtres aura certainement une influence sur les temps de communications. Dans cet exemple, il serait plus intéressant de placer le maître qui gère les machines de dsllab, sur la grille. Ainsi, il communiquera plus rapidement avec les autres maîtres.

Enfin pour l'algorithme HWS, la preuve fonctionne en utilisant l'analyse d'un mécanisme d'équilibrage de la charge pour un groupe d'esclave et une autre analyse pour les maîtres. Dans cette analyse, il n'est pas obligatoire de conserver les mêmes algorithmes de répartition de la charge pour les maîtres et les esclaves. Les mécanisme de répartition utilisés doivent seulement avoir une borne sur le temps d'exécution pour utiliser la même démarche d'analyse. Il serait certainement intéressant de regarder la composition d'autres algorithmes afin de trouver une méthode permettant de supprimer le choix de la limite par l'utilisateur.

Conclusion

Tout au long de cette thèse, nous nous sommes intéressés au problème d'ordonnancement avec communications. La prise en compte des communications devient inévitable avec l'architecture des prochains processeurs ; Au sein d'une puce de calcul, nous retrouvons un réseau de communications complexe. Afin de préparer la programmation de telles architectures, nous avons étudié deux problèmes distincts d'ordonnancement en-ligne avec communications. Cet ordonnancement en-ligne nous impose une connaissance imparfaite des conditions d'exécution. En pratique, ce manque d'informations affaiblit fortement les performances des différents algorithmes envisageables.

Nous avons cherché à travers notre travail à tirer parti de la moindre information disponible pour réduire l'impact du réseau au maximum. Notre travail se divise en deux parties proposant deux manières d'aborder la problématique initiale. La première tente d'utiliser de l'information provenant de l'application ; La seconde quant à elle tente d'utiliser de l'information provenant de la plate-forme.

Dans la partie II, nous avons ainsi considéré une classe d'applications particulières pour lesquelles la structure du «Directed Acyclic Graph» (DAG) de tâches se trouve connue avant l'exécution. Cette classe d'application provient d'un problème pratique réel : L'exécution d'une application de type Makefile sur plate-forme distribuée. Pour ce problème, nous avons proposé différents algorithmes permettant, à travers l'utilisation de tâches virtuelles, d'augmenter la localité des accès aux données et ainsi de réduire la pression sur les ressources réseau. Nous avons montré au travers d'une analyse théorique, que notre principal algorithme, Work-Stealing with COMmunication (WSCOM), réalise au pire cas un nombre de transferts de données proportionnels au nombre de processeurs (p) multiplié par le chemin critique (D) pour les DAG «Fork-Join». Ce nombre ($O(pD)$) est relativement proche de la borne inférieure de $O(p\sqrt{D})$ présenté section 5.1.1 (page : 80).

Nous avons réalisé un ensemble d'expériences validant les algorithmes proposés au travers de simulations. Nous avons mis en évidence que la réduction du nombre de communications obtenue se traduit effectivement par une amélioration des temps de complétion, notamment en présence de congestion. Enfin, nous avons montré que l'utilisation de WSCOM permet réellement d'étendre le champs d'application du vol de travail vers des plates-formes de faible bande-passante.

D'une manière transversale, nous avons, dans la seconde partie III, considéré que la topologie de la plate-forme est connue des algorithmes d'ordonnancement. Cette description fournit uniquement un regroupement des processeurs qui communiquent efficacement entre eux. Bien que cette information ne soit pas suffisante à elle seule pour prédire les temps de communication (les volumes de données étant inconnus), elle nous a permis de développer deux algorithmes : Probabilist Work-Stealing (PWS) et Hierarchical Work-Stealing (HWS). En privilégiant l'utilisation des liens locaux,

ceux-ci permettent une fois encore de réduire la consommation de ressources réseau. L'analyse théorique de PWS montre que le temps de complétion reste raisonnable. Pour HWS, une analyse initiale nous permet de voir que, dans le cas restreint considéré, le nombre de vols distants est fortement réduit. Les expériences réalisées montrent que ces algorithmes permettent effectivement un gain de performances sur des applications intensives en communication.

Au niveau logiciel, nos travaux ont permis le développement de *Outils de parallèlisation d'applications décrites par un Makefile sur plates-formes distribuées (DSMake)*, un outil distribué pour l'exécution de Makefiles, basé sur WSCOM. À l'heure actuelle, cet outil reste néanmoins dépendant d'un système de fichiers distribué pour la réalisation des transferts de données. En parallèle à ces développements, PWS et HWS ont été intégrés à la bibliothèques *Kaapi*.

Les perspectives de ces travaux sont de plusieurs ordres.

Les perspectives à court terme serait de compléter l'implantation de *DSMake* pour mettre en œuvre les mécanismes de prefetching nécessaires à WSCOM_{pf}. Il serait alors possible de réaliser une campagne d'expériences réelles validant les résultats obtenus par simulations. De plus, cette implantation nous permettrait de mettre en évidence que les mécanismes de vol de travail actuels peuvent profiter d'un gain non négligeable en réalisant l'envoi des données au plus tôt.

Toujours dans le registre des expériences, il serait sans doute intéressant de réaliser des expériences sur HWS/PWS à plus grande échelle. Nous avons déjà évoqué quelques pistes en ce sens dans la section 7.3.

Dans un registre plus théorique, il serait envisageable d'améliorer les analyses théoriques de HWS et PWS. En effet, l'analyse actuelle de HWS est restreinte à des groupes homogènes de machines ; tandis que l'analyse de PWS pourrait peut-être être améliorée pour montrer une réduction du nombre de vols distants.

Nous nous sommes intéressés dans nos travaux à la parallélisation d'applications décrites par un Makefile. Généralement, cette méthode de programmation est utilisée pour la compilation de programmes. Lors de l'installation d'un programme, il y a généralement un ensemble d'autres applications à installer. Ainsi, nous pourrions nous intéresser à la parallélisation de la compilation d'un ensemble de programmes. La compilation d'un programme peut être modélisée : soit comme une tâche malléable, soit comme un ensemble de tâches avec dépendances. Dans ce contexte, il serait intéressant de savoir comment obtenir un ordonnancement de vol de travail pour des tâches malléables et de comparer avec l'ordonnancement du DAG complet par vol de travail hiérarchique.

Enfin, une évolution à plus moyen terme serait de considérer des applications sur des architectures de réseaux sur puces existantes à l'heure actuelle. Nous pouvons citer par exemple les architectures développées par Tilera [87] qui propose une centaine de cœurs disposés en grille bidimensionnelle. Sur une telle architecture, aucune décomposition hiérarchique ne s'impose d'elle même. Il serait certainement possible de constituer alors des groupes virtuels pour HWS, éventuellement en fonction de l'application considérée. PWS constitue sans doute également un algorithme intéressant pour ce type de plate-forme.

Au final, nous espérons avoir montré, à travers nos travaux, que l'absence d'informations lors de l'ordonnancement d'applications distribuées ne pose pas forcément des contraintes aussi insurmontables que ce que l'on pourrait imaginer de prime abord. Ces travaux ont été l'occasion d'essayer de lier pratique et théorie, ce qui s'inscrit pleinement dans la dynamique de mon équipe.

Algorithmes en pseudo-code A

Sommaire du chapitre

Dans cette annexe, nous proposons une description en pseudo-code des algorithmes les plus utilisés au sein de la thèse. Les algorithmes présentés sont divisés en trois parties : ordonnancements par liste, agrégations de tâches et ordonnancements en-ligne.

A.1 Ordonnancements par liste

L'analyse complète des algorithmes d'ordonnancement par liste a été réalisée dans la section 2.1 (page : 23). Nous détaillons, dans un premier temps, le principe en pseudo-code commun aux différents ordonnancements par liste. Puis, les sections suivantes détaillent l'implantation d'algorithmes en se basant sur l'algorithme générique.

A.1.1 Algorithme Générique d'ordonnancement par liste

Algorithme 4 : Ordonnancement par liste

```
Data : E_t : Ensemble de tâches
Result : O_t : Ordonnancement final des tâches
begin
    while des tâches ne sont pas ordonnancées do
        T=Prochaine_tâche (E_t,O_t)
        (p,t)=Assignation (T,O_t)
        if une tâche fini avant la date t then
            Mise_à_jour (t)
                        /* Mise à jour des tâches prêtes et de l'horloge
            virtuelle */
        else

            Ordonnance (T,P,t,O_t)
```

A.1.2 Earliest Task First (ETF)

Pour détailler Earliest Task First (ETF), nous nous basons sur l'algorithme 4. Nous décrivons pour ETF uniquement les fonctions principales qui sont : Prochaine_tâche et Assignation.

Algorithme 5 : ETF

Function `Prochaine_tâche` (E_t,O_t)
begin
    min $= \infty$
    $T =$ non défini
    for $i \in$ `Tâches_prêtes` (E_t) **do**
        $d =$ `Date_de_début_d'exécution_minimale` (i)
            `/* Date à partir de laquelle la tâche` i `peut être exécutée par au moins un processeur */`
        if $min > d$ **then**
            min $= d$
            $T = i$
    return T

Function `Assignation` (T,O_t)
begin
    min $= \infty$
    $P =$ non défini
    for $i \in$ *Processeurs* **do**
        $d =$ `Date_de_début_d'exécution` (T,i)
            `/* Date d'exécution de la tâche` T `sur le processeur` i `*/`
        if $min > d$ **then**
            min $= d$
            $P = i$
    return (P,min)

A.1.3 Heterogeneous Earliest Finish Time (HEFT)

Pour détailler Heterogeneous Earliest Finish Time (HEFT), nous nous basons sur l'algorithme 4. Nous décrivons pour l'algorithme HEFT uniquement les fonctions principales qui sont : Prochaine_tâche et Assignation.

Algorithme 6 : HEFT

```
Function Prochaine_tâche (E_t,O_t)
begin
    max = 0
    T = non défini
    for i ∈ Tâches_prêtes (E_t) do
        d = Hauteur (i)
              /* Calcul de la hauteur de la tâche i en considérant les
        temps de calculs et les temps de communications */
        if max < d then
            max = d
            T = i
    return T
Function Assignation (T,O_t)
begin
    min = ∞
    P = non défini
    for i ∈ Processeurs do
        d = Date_de_fin_d'exécution (T,i)
         /* Date de fin d'exécution de la tâche T sur le processeur i
        */
        if min > d then
            min = d
            start_min = Date_de_début_d'exécution (T,i)
            P = i
    return (P, start_min)
```

A.2 Algorithmes d'agrégation

Les algorithmes d'agrégation ont un fonctionnement différent des algorithmes de liste. Durant l'exécution, un processeur peut être inactif et une tâche prête sans que celle-ci lui soit assignée. La description de ces algorithmes a été réalisée dans la section 2.2 (page : 28). Nous détaillons ici quelques algorithmes pseudo-code qui ont été utilisées dans le manuscrit.

A.2.1 Algorithme récursif de décomposition en agrégats convexes

Algorithme 7 : Décomposition récursive

```
Function Décompose (G(V,E)) begin
    a = Sélection_aléatoire_d'une_tâche (G)
    G' = Suppression_successeurs_et_prédécesseurs (a,G)
    b = Sélection_aléatoire_d'une_tâche (G')
    (A,B) = Formation_des_agrégats (a,b,G)
    S = Génération_agrégat_successeur (A,B,G)
    P = Génération_agrégat_prédécesseur (A,B,G)
    return (P,A,B,S)
Function Décomposition_récursive (niveau, G(V,E))
Data : la variable seuil est la limite de récursion fournie par l'utilisateur

begin
    if niveau>seuil then
        Identification_agrégat(G) ;        /* Identifie les agrégats afin de
        reconstruire les regroupements */
        return G ;
    max = 0
    for i =0 ; i < 10 ; i++ do
        (P_tmp, A_tmp, B_tmp, S_tmp) = Décompose (G)
        if max<size_min(A,B) then
            (P,A,B,S)=(P_tmp, A_tmp, B_tmp, S_tmp)
    Décomposition_récursive (niveau+1, P)
    Décomposition_récursive (niveau+1, A)
    Décomposition_récursive (niveau+1, B)
    Décomposition_récursive (niveau+1, S)
    return G ;
```

A.2.2 Edge Zeroing (EZ)

Algorithme 8 : EZ

```
Data : G(V, E) : Graphe de tâches
begin
    Liste = Tri_des_arêtes_en_fonction_des_coûts_de_communications (G)
        /* Tri les arêtes en fonction de leurs coûts de communication
    (du plus grand au plus petit) */
    C = Chemin_critique (G)
    L = Pop (Liste)
    while L est défini do
        ancien_graphe = G
        Agrégation_des_tâches (L,G)   /* Agrégation des tâches reliées
        par l'arête L */
        Ajout_d'arêtes (G, ancien_graphe)
                /* Ajoute une arête fictive entre les tâches d'un même
        agrégat qui n'ont pas de relations de précédences. Les arêtes
        ajoutées sont orientées de la tâche la plus haute à la moins
        haute dans l'ancien graphe. */
        C_n = Chemin_critique (G)
        if C ≤ C_n then
            G = ancien_graphe
        else
            C = C_n
        L = Pop (Liste)
```

A.2.3 Dominant Sequence Clustering (DSC)

Algorithme 9 : DSC

```
Procedure DSC (G) begin
    N = Noeuds (G)
    n = Tâche_de_priorité_maximale (N)
    while n est défini do
        profondeur = Profondeur (n)
        Agrégat = non défini
        for A ∈ Agrégats_prédécesseurs(n) do
            if profondeur > Profondeur_avec_agrégation (A,n) then
                profondeur = Profondeur_avec_agrégation (A,n)
                Agrégat = A
        if Agrégat est défini then
            G = Agrégation (Agrégat,n)
        Enlever (n,N)
        n = Tâche_de_priorité_maximale (N)
Function Tâche_de_priorité_maximale (N) begin
    max = 0
    T = non défini
    for i ∈ N do
        if Prédécesseurs_ordonnancés (i) et
                max < Hauteur (i) + Profondeur (i) then
            max = Hauteur (i) + Profondeur (i)
            T = i
    return T
```

A.3 Vol de travail

Le vol de travail est un algorithme central dans cette thèse. Le fonctionnement classique du vol de travail est présenté dans la section 3.2 (page : 41). Une description en pseudo-code est fourni par l'algorithme 10.

Algorithme 10 : Vol de travail

```
Data : T_initiale                          /* La tâche initiale de l'exécution */
Data : Pile_id     /* Pile de tâches associée au processeur courant de
       Tâche */
T_cur = non défini
if le processeur a l'identifient 0 then
    T_cur = T_initiale                     /* Initialisation d'un des processeurs */
while des tâches ne sont pas exécutées do
    while T_cur est défini do
        Tâches = Exécution (T_cur)  /* Retourne les tâches activées (donc
        prêtes) par l'exécution de T_cur */
        for i ∈ Tâches do
            push (Pile_id,i)
        T_cur = Pop(Pile_id)                /* Suppression d'une tâche prête */
    P = Sélection_aléatoire_d'un_processeur ()
    T_cur = Vol (P)
Function Vol (P) begin
    T= non défini
    T=head(Pile_P)   /* Prend la tâche la plus ancienne sur la pile du
    processeur P */
    return T
```

Bibliographie

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM. ➪ *2 citations pages 43 and 51*

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM. ➪ *cité page 43*

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2) :115–144, 2001. ➪ *4 citations pages 43, 44, 45, and 133*

[4] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *Algorithmica*, pages 379–386, 1990. ➪ *cité page 13*

[5] Michael A. Bender and Cynthia A. Phillips. Scheduling dags on asynchronous processors. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 35–45, New York, NY, USA, 2007. ACM. ➪ *cité page 18*

[6] Micheal A. Bender and Micheal O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory of Computing Systems*, 35 :289–304, 2002. ➪ *2 citations pages 18 and 44*

[7] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *In Proc. 19th ACM-SIAM Sympos. Discrete Algorithms*, pages 501–510, 2008. ➪ *2 citations pages 51 and 116*

[8] Robert D. Blumofe. *Executing multithreaded programs efficiently.* PhD thesis, Massachusetts Institute Of Technology, Massachusetts, USA, 1995. ➪ *2 citations pages 41 and 131*

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science FOCS*, pages 356–368, 1994. ➪ *4 citations pages 2, 42, 44, and 131*

[10] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46 :720–748, September 1999. ➪ *cité page 44*

[11] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, New York, NY, USA, 1998. ➪ *cité page 13*

[12] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008. ➪ *cité page 89*

[13] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ➪ *cité page 1*

[14] Douglas Campbell and Chris Grevstad. A tutorial for make. In *Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective*, ACM '85, pages 374–380, New York, NY, USA, 1985. ACM. ➪ *cité page 61*

[15] Louis-Claude Canon, Olivier Dubuisson, Jens Gustedt, and Emmanuel Jeannot. Defining and controlling the heterogeneity of a cluster : The wrekavoc tool. *J. Syst. Softw.*, 83 :786–802, May 2010. ➪ *cité page 88*

[16] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000 : a large scale reconfigurable controlable and monitorable grid platform. In *SC*, 2005. ➪ *cité page 88*

[17] Guy Blelloch Carnegie, Guy E. Blelloch, and Phillip B. Gibbons. Effectively sharing a cache among threads. In *in SPAA '04 : Proceedings of the sixteenth annual ACM symposium on Parallelism in*, pages 235–244, 2004. ➪ *2 citations pages 51 and 116*

[18] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008. ➪ *cité page 88*

[19] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ➪ *cité page 1*

[20] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society. ➪ *2 citations pages 41 and 47*

[21] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC, pages 151–158, New York, NY, USA, 1971. ACM. ➪ *cité page 10*

[22] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *Proceedings of 3rd International ICST Conference on Simulation Tools and Techniques*, Malaga Espagne, mar 2010. ICST. ➪ *cité page 90*

[23] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp : towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM. ➪ *cité page 22*

[24] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7 :279–301, October 1989. ➪ *cité page 41*

[25] A De Boelelaan, Rob V. van Nieuwpoort, Jason Maassen, Wrzesinska Gosia, Thilo Kielmann, and Henri E. Adaptive load-balancing for divide-and-conquer grid applications. *Journal of Supercomputing*, 2004. ➪ *cité page 53*

[26] Robert P. Dick, David L. Rhodes, and Wayne Wolf. Tgff : task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, CODES/CASHE '98, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society. ➪ *cité page 90*

[27] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. ACM. ➪ *2 citations pages 44 and 65*

[28] Bruno Donassolo, Henri Casanova, Arnaud Legrand, and Pedro Velho. Fast and scalable simulation of volunteer computing systems using simgrid. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 605–612, New York, NY, USA, 2010. ACM. ➪ *cité page 88*

[29] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003. ➪ *cité page 88*

[30] Gilles Fedak, Jean-Patrick Gelas, Thomas Hérault, Victor Iniesta, Derrick Kondo, Laurent Lefèvre, Paul Malecot, Lucas Nussbaum, Ala Rezmerita, and Olivier Richard. DSL-Lab : a Low-Power Lightweight Platform to Experiment on Domestic Broadband Internet. In *International Symposium on Parallel and Distributed Computing (ISPDC 2010)*, Istanbul, Turkey, July 2010. ➪ *cité page 146*

[31] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ➪ *4 citations pages 1, 41, 47, and 89*

[32] Michael R. Garey and David S. Johnson. "Strong" NP-completeness results : motivation, examples, and implications. *J. Assoc. Comput. Mach.*, 25(3) :499–508, 1978. ➪ *2 citations pages 11 and 14*

[33] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ➪ *2 citations pages 10 and 11*

[34] Nicolas Gast and Gaujal Bruno. A mean field model of work stealing in large-scale systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 13–24, New York, NY, USA, 2010. ACM. ➪ *2 citations pages 44 and 65*

[35] Bruno Gaujal, Guillaume Huard, Eric Thierry, and Denis Trystram. Convex Scheduling for Grid Computing. In *WASC, 1st Workshop on Algorithms for Scheduling and Communication*, Bertinoro, Italy, 2004. ➪ *cité page 67*

[36] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi : A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In *PASCO '07 : Proceedings of the 2007 international workshop on*

Parallel symbolic computation, pages 15–23, New York, NY, USA, 2007. ACM. ➪ *3 citations pages 1, 41, and 47*

[37] Thierry Gautier, Jean-Louis Roch, and Frédéric Wagner. Fine grain distributed implementation of a dataflow language with provable performances. In *ICCS*, 2007. ➪ *cité page 48*

[38] Teofilo F. Gonzalez, Oscar H. Ibarra, and Sartaj Sahni. Bounds for lpt schedules on uniform processors. *Siam Journal on Computing*, 6 :155–166, 1977. ➪ *cité page 12*

[39] Gaël Gorgo and Jean-Marc Vincent. Perfect sampling of load sharing policies in large scale distributed systems. In *Proceedings of the 17th international conference on Analytical and stochastic modeling techniques and applications*, ASMTA'10, pages 174–188. Springer-Verlag, 2010. ➪ *cité page 53*

[40] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2) :416–429, 1969 . ➪ *7 citations pages 12, 13, 14, 15, 17, 23, and 24*

[41] T. Hagras and J. Janeček. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Comput.*, 31 :653–670, July 2005. ➪ *cité page 26*

[42] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM. ➪ *cité page 44*

[43] Willy Herroelen and Roel Leus. Robust and reactive project scheduling : a review and classification of procedures. *International Journal of Production Research*, 42(8) :1599–1620, 2004. ➪ *cité page 40*

[44] Han J.A. Hoogeveen, jan K. Lenstra, and Bart Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, 16(3) :129 – 137, 1994. ➪ *cité page 22*

[45] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18 :244–257, April 1989. ➪ *3 citations pages 24, 25, and 26*

[46] Andi Kleen. A numa api for linux. Technical report, SUSE Labs, april 2005. ➪ *cité page 143*

[47] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7 :506–521, 1996. ➪ *cité page 31*

[48] Jonathan K. Lee and Jens Palsberg. Featherweight x10 : A core calculus for async-finish parallelism. In *PPoPP*, 2010. ➪ *cité page 47*

[49] Renaud Lepère and Denis Trystram. A new clustering algorithm for large communication delays. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, Washington, DC, USA, 2002. IEEE Computer Society. ➪ *3 citations pages 29, 30, and 67*

[50] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *HCW '99*, Washington, DC, USA, 1999. IEEE Computer Society. ➪ *2 citations pages 25 and 92*

[51] Muthucumaru Maheswaran and Howard Jay Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, HCW '98, pages 57–, Washington, DC, USA, 1998. IEEE Computer Society. ➪ *cité page 92*

[52] Carolyn McCreary and Helen Gill. Automatic determination of grain size for efficient parallel processing. *Commun. ACM*, 32 :1073–1078, September 1989. ➪ *cité page 29*

[53] David M. Nicol and Joel H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37 :1073–1087, September 1988. ➪ *cité page 40*

[54] Lucas Nussbaum and Olivier Richard. Lightweight emulation to study peer-to-peer systems. *Concurrency and Computation : Practice and Experience*, 20(6) :735–749, 2008. ➪ *cité page 88*

[55] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 573–577. Springer-Verlag, London, UK, 1996. ➪ *cité page 92*

[56] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ➪ *cité page 1*

[57] Jonathan E. Pecero-Sánchez and Denis Trystram. A new Genetic Convex Clustering algorithm for parallel time minimization with large communication delays. In *International Conference Parallel Computing, ParCo'2005*, Malaga, Spain, sep 2005. ➪ *cité page 30*

[58] Odysseas I. Pentakalos. An introduction to the infiniband architecture. In *Int. CMG Conference*, pages 425–432, 2002. ➪ *cité page 137*

[59] Swann Perarnau and Guillaume Huard. Krash : Reproducible cpu load generation on many-core machines. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010. ➪ *cité page 88*

[60] Chen Quan, Huang Zhiyi, Guo Minyi, and Zhou Jingyu. Cab : Cache aware bi-tier task-stealing in multi-socket multi-core architecture. *In the International Conference on Parallel Processing*, September 2011. ➪ *cité page 118*

[61] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th international Euro-Par conference on Parallel processing : Part I*, Euro-Par'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag. ➪ *cité page 117*

[62] Jean-Noël Quintin and Swann Perarnau. Sensibilité des algorithmes d'ordonnancement. In *French RENPAR Conference*, 2011. ➪ *cité page 90*

[63] Andrei Radulescu, Arjan J. C. van Gemund, and Hai-Xiang Lin. Llb : A fast and effective scheduling algorithm for distributed-memory systems. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, pages 525–530, Washington, DC, USA, 1999. IEEE Computer Society. ➪ *cité page 34*

[64] Andrei Radulescu and Arjan J. C. Van Gemund. Fast and effective task scheduling in heterogeneous systems. In *Proceedings of the 9th Heterogeneous Compu-*

ting Workshop, HCW '00, Washington, DC, USA, 2000. IEEE Computer Society. ➪ *cité page 26*

[65] Christiane Pousa Ribeiro, Jean-Francois Mehaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '09, pages 59–66, Washington, DC, USA, 2009. IEEE Computer Society. ➪ *cité page 143*

[66] Sakellariou Rizos and Zhao Henan. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004. ➪ *cité page 92*

[67] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *IPDPS*, 2008. ➪ *2 citations pages 41 and 47*

[68] Jean-Louis Roch. On the number of steal operations in a greedy schedule. Technical report, Personal Communication. ➪ *4 citations pages 43, 45, 46, and 47*

[69] Ernest Rothman, Brian Jepson, and Rich Rosen. *Mac OS X For Unix Geeks*. O'Reilly Media, Inc., 2008. ➪ *cité page 101*

[70] Sartaj Sahni. Scheduling master-slave multiprocessor systems. *IEEE Trans. Comput.*, 45 :1195–1199, October 1996. ➪ *cité page 38*

[71] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989. ➪ *cité page 31*

[72] Pranab Kumar Sen and Pranab Kumar Sen. Invariance principles for the coupon collector's problem : a martingale approach. *Ann. Statist*, pages 372–380, 1979. ➪ *cité page 47*

[73] Agarwal Shivali, Barik Rajkishore, Bonachea Dan, Sarkar Vivek, Shyamasundar Rudrapatna K., and Yelick Katherine. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA*, 2007. ➪ *cité page 44*

[74] Gilbert C. Sih and Edward A. Lee. Dynamic-level scheduling for heterogeneous processor networks. In *SPDP*, pages 42–49, 1990. ➪ *cité page 31*

[75] Oliver Sinnen. *Task Scheduling for Parallel Systems ; electronic version*. Wiley, Newark, 2007. ➪ *cité page 22*

[76] W. Richard Stevens. *UNIX Network Programming : Networking APIs : Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1997. ➪ *cité page 50*

[77] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001. ➪ *cité page 73*

[78] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997. ➪ *cité page 140*

[79] Yang Tao and Gerasoulis Apostolos. Pyrros : static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 428–437, New York, USA, 1992. ACM. ➪ *cité page 31*

[80] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, and Bruno Raffin. A Work Stealing Algorithm for Parallel Loops on Shared Cache Multicores. In *4th Workshop on Highly Parallel Processing on a Chip (HPPC)*, aug 2010. ➪ *2 citations pages 51 and 116*

[81] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation*, volume 6507 of *Lecture Notes in Computer Science*, pages 291–302. Springer Berlin / Heidelberg, 2010. ➪ *cité page 45*

[82] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5) :379–394, 2002. ➪ *cité page 90*

[83] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13 :260–274, March 2002. ➪ *3 citations pages 25, 26, and 92*

[84] John D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69, 1994. ➪ *cité page 44*

[85] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP : Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA, 2001. ACM. ➪ *4 citations pages 53, 54, 120, and 141*

[86] Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Satin : Simple and efficient java-based grid programming. *Scientific International Journal for Parallel and Distributed Computing*, 6(3) :19–32, September 2005. ➪ *2 citations pages 41 and 47*

[87] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27 :15–31, September 2007. ➪ *cité page 148*

[88] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. Seti@home massively distributed computing for seti. *Computing in Science and Engg.*, 3 :78–83, January 2001. ➪ *cité page 17*

[89] Min-You Wu and Daniel D. Gajski. Hypertool : A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1 :330–343, 1990. ➪ *cité page 31*

[90] Tao Yang. *Scheduling and code generation for parallel architectures*. PhD thesis, New Brunswick, New Jersey, USA, 1993. UMI Order No. GAX93-33468. ➪ *cité page 33*

[91] Tao Yang and Apostolos Gerasoulis. Dsc : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5 :951–967, 1994. ➪ *2 citations pages 31 and 67*

[92] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34 :44–49, September 2005. ➪ *cité page 19*

[93] Henan Zhao and Rizos Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 189–194. Springer Berlin / Heidelberg, 2003. ➪ *cité page 26*

Table des figures

Table des matières

Glossaire pour les logiciels

Cilk Bibliothèque classique de vol de travail. 1, 2, 47, 48, 59, 62, 72, 118

DSMake Outils de parallèlisation d'applications décrites par un Makefile sur plates-formes distribuées. 2, 3, 148

GGEN Graph-GENerator. 90

InfiniBand Réseau haute performance présent dans de nombreuses plates-formes. 50, 52, 137

Kaapi Bibliothèque de vol de travail pour plates-formes distribuées. 1, 3, 4, 47–50, 53, 115, 118, 120, 122, 123, 136, 138, 140, 141, 143, 148, 167, 173

LogP Modèle de communication entre deux machines. 22

MPI Message Passing Interface. 1, 88

OpenMP API pour la programmation parallèle en mémoire partagée. 1

PYRROS Logiciel implantant l'algorithme DSC et des méthodes de repliement. 31, 34

Satin Bibliothèque de vol de travail en java pour plates-formes distribuées. 47, 53, 120, 136, 140

SimGrid Ensemble d'outils permettant la simulation de plates-formes hétérogènes et distribuées. 88–90

STL Standard Template Library. 140

TBB Threading Building Blocks. 47, 48

XWS X10 Work Stealing. 47

Glossaire pour les algorithmes d'ordonnancement

CHS Cluster-aware Hierarchical Stealing. 53, 54

CLS Cluster-aware Load-based Stealing. 53, 54, 120

CRS Cluster-aware Random Stealing. 54, 136, 140, 141, 143

DCP Dynamic critical-path. 31

DL Dynamic-level. 31

DSC Dominant Sequence Clustering. 3, 31–35, 67, 151, 157, 167, 174

ETF Earliest Task First. 25, 26, 151, 153, 174

EZ Edge Zeroing. 31–33, 151, 156, 174

HEFT Heterogeneous Earliest Finish Time. 25, 26, 28, 92, 97, 151, 154, 174

HWS Hierarchical Work-Stealing. 3, 4, 113, 116–118, 120, 121, 125, 126, 129, 130, 132, 133, 136–143, 145–148, 169, 173

list_min Ordonnancement hors-ligne prenant le meilleur ordonnancement par liste pour le DAG considéré. 97–102, 105, 168, 169

LPT Largest Processing Time. 12–14, 24, 26, 167

MD Mobility directed. 31

PDF Parallel Depth First. 51, 52, 116

PWS Probabilist Work-Stealing. 3, 4, 113, 116, 117, 125–127, 129, 136–143, 145, 147, 148, 169, 173

Sufferage Algorithme d'ordonnancement par liste. 25, 27, 92, 97

vol de travail Algorithme d'ordonnancement en-ligne. 2, 3, 15, 18, 21, 37, 38, 41–48, 50–52, 59, 61–66, 70, 72, 74, 76, 79, 89, 91, 92, 94–96, 98, 106–109, 113, 115, 116, 118, 120, 126, 129, 131–133, 136, 137, 139–143, 145, 147, 158, 167–169, 172

WS Vol de travail classique. 91, 92, 94–96, 102, 107, 136, 140, 142, 143, 168, 169

WSCOM Work-Stealing with COMmunication. 3, 4, 59, 65, 69, 70, 72, 74, 76, 77, 79, 80, 86–89, 91–96, 98–109, 113, 147, 148, 168, 169, 173

Mots-Clefs

arête lien orienté entre deux noeuds représentant une relation de précédence entre deux tâches. 19, 31, 32, 48, 49, 61, 70, 71, 73, 74, 90, 91, 97, 100, 101, 106, 131, 156, 167

CLANS Agrégats de tâches dont celles-ci ont le même ensemble de prédécesseurs et de successeurs. 3, 29, 30, 167

congestion État d'un réseau au sein du quel des données envoyées par différentes machines transitent par un même lien saturé. 25–28, 40, 59, 88–91, 94, 96–100, 108, 117, 147, 168

convexe Le découpage d'un DAG est convexe si le graphe constitué par les agrégats est un DAG. 29–31, 67, 76, 91, 167

D Longueur du chemin le plus long au sein du DAG. 21, 24, 25, 45–47, 80–87, 91, 126–129, 131–136, 147

DAG Graphe Dirigé Acyclique. 19, 21, 26, 29–32, 34, 39, 40, 42, 45, 47–49, 59, 61–80, 84–99, 101–108, 113, 119, 122, 130, 131, 133, 143, 147, 167–169, 172, 173

DHT Table de hachage distribuée. 73, 75, 89, 92, 93, 168

en-ligne Ce dit pour un algorithme qui calcule une solution à un problème d'ordonnancement au cours de l'exécution. 2, 3, 7, 8, 12–15, 18, 23, 35, 37–40, 45, 59, 61, 79–81, 84, 86, 88, 91–94, 97–100, 102, 104, 105, 108, 113, 147, 151, 167, 168, 171, 173

FIFO Premier entré, premier sorti. 43

Fork-Join Structure de DAG particulière. 4, 47, 62, 63, 72, 80, 86, 87, 147, 168

hauteur hauteur d'une tâche est la longueur du chemin le plus long partant de celle-ci. 21, 26, 32, 34, 81, 82, 84, 154

heuristique Algorithme dont les performances ne sont pas analysés théoriquement. 22, 24–27, 29–32, 51, 53, 54, 59, 67, 87, 92, 94, 96–101, 103–105, 116, 117, 119, 129, 131–133, 138–140, 142, 143, 168, 169

hors-ligne Ce dit d'un algorithme qui calcule une solution à un problème d'ordonnancement avant l'exécution. 3, 7, 8, 12, 13, 15, 18, 22, 23, 25, 37, 39, 59, 76, 79, 84, 88, 89, 91, 92, 96–100, 104, 105, 108, 113, 171–173

LIFO Dernier entré, premier sorti. 42, 43

Makefile Fichier pour la commande Make décrivant les instructions à exécuter et leurs dépendances. 3, 61, 63–65, 70, 79, 91, 92, 101–103, 105, 106, 108, 147, 148, 167–169, 173

Maître-travailleurs Modèle de programmation parallèle. 38

NUMA Accès Mémoire non-uniforme. 50, 51, 115, 125, 142, 143, 173

profondeur profondeur d'une tâche est la longueur du chemin le plus long entre une source et cette dernière. 21, 32, 33, 42, 44, 46–48, 54, 66, 74, 81, 84, 91, 118–120, 122, 126–128, 130, 169

SIPS Strictly Increasing Per Steal. 45, 46, 126

Spawn Mot clef de Cilk permettant de désigner un ensemble d'instructions qui peuvent être exécutées indépendamment des autres instructions. 2, 47

Sync Mot clef de Cilk permettant d'exécuter les instructions qui suivent dès que les parties indépendantes préalablement désignées avec un «spwan» sont terminées. 2, 47, 48, 62

thread Processus léger. 1

W Quantité de travail présente sur l'ensemble du DAG. 15, 16, 19, 21, 25, 45–47, 80, 81, 84–86, 91, 129, 131–133

Résumé.

La course à l'augmentation de la puissance de calcul qui se déroule depuis de nombreuses années entre les différents producteurs de matériel a depuis quelques années changé de visage : nous assistons en effet désormais à une véritable démocratisation des machines parallèles avec une complexification sans cesse croissante de la structure des processeurs. À terme, il est tout à fait envisageable de voir apparaître pour le grand public des architectures pleinement hétérogènes composées d'un ensemble de cœurs reliés par un réseau sur puce.

La parallélisation et l'exécution parallèle d'applications sur les machines à venir soulèvent ainsi de nombreux problèmes. Parmi ceux-ci, nous nous intéressons ici au problème de l'ordonnancement d'un ensemble de tâches sur un ensemble de cœurs, c'est à dire le choix de l'affectation du travail à réaliser sur les ressources disponibles.

Parmi les méthodes existantes, on distingue deux types d'algorithmes : en-ligne et hors-ligne. Les algorithmes en-ligne comme le *vol de travail* présentent l'avantage de fonctionner en l'absence d'informations sur le matériel ou la durée des tâches mais ne permettent généralement pas une gestion efficace des communications. Dans cette thèse, nous nous intéressons à l'ordonnancement de tâches en-ligne sur des plates-formes complexes pour lesquelles le réseau peut, par des problèmes de congestion, limiter les performances. Plus précisément, nous proposons de nouveaux algorithmes d'ordonnancement en-ligne, basés sur le vol de travail, ciblant deux configurations différentes.

D'une part, nous considérons des applications pour lesquelles le graphe de dépendance est connu à priori. L'utilisation de cette information nous permet ainsi de limiter les quantités de données transférées et d'obtenir des performances supérieures aux meilleurs algorithmes hors-ligne connus. D'autre part, nous étudions les optimisations possibles lorsque l'algorithme d'ordonnancement connaît la topologie de la plate-forme. Encore une fois, nous montrons qu'il est possible de tirer parti de cette information pour réaliser un gain non-négligeable en performance.

Nos travaux permettent ainsi d'étendre le champ d'application des algorithmes d'ordonnancement vers des architectures plus complexes et permettront peut-être une meilleure utilisation des machines de demain.

Mots-clés : algorithmes parallèles, ordonnancements, vol de travail, topologie de la plate-forme, communications.

Abstract.

The race towards more processing power between all different hardware manufacturers has in recent years faced deep changes. We see nowadays a huge development in the use of parallel machines with more and more cores and increasingly complex architectures. It seems now clear that we will witness in a near future the development of cheap Network On Chip computers.

Executing parallel applications on such machines raises several problems. Amongst them we take in this work interest in the problem of scheduling a set of tasks on a set of computing resources. Between all existing methods we can generally distinguish on-line or off-line algorithms. On-line algorithms like *work-stealing* present the advantage to work without informations on hardware or tasks durations but do not generally achieve an efficient control of communications. In this book we take interest in on-line tasks scheduling on complex platforms where networking can impact (through congestion) performance. More precisely, we propose several new scheduling algorithms based on work-stealing targeting two different configurations.

In a first study, we consider applications whose dependency graph is known in advance. By taking advantage of this information we manage to limit the amount of data transfered and thus to achieve high performance and even outperform the best known off-line algorithms. Concurrently to that, we also study possible optimisations in the case where knowledge of platform topology is available. We show again that it is possible to use this information to enhance performance.

Our work allows therefore to extend the application field of scheduling algorithms towards more complex architectures and we hope will allow a better use of tomorrow's machine.

Keywords: parallel algorithms, work-stealing, scheduling, platform topology, communications.