ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'école doctorale de MathIF

présentée et soutenue publiquement le 11 décembre 2003

par Monsieur Arnaud LEGRAND

Algorithmique parallèle hétérogène et techniques d'ordonnancement : approches statiques et dynamiques

Directeurs de thèse :	Monsieur	Olivier	BEAUMONT
	Monsieur	Yves	ROBERT
Après avis de :	Monsieur	Jacques	BAHI, Membre/Rapporteur
	Madame	Claire	HANEN, Membre/Rapporteur

Devant la commission d'examen formée de :

Monsieur	Jacques	BAHI, Membre/Rapporteur
Monsieur	Olivier	BEAUMONT, Membre/Directeur de Thèse
Monsieur	Michel	COSNARD, Membre/Président du Jury
Madame	Claire	HANEN, Membre/Rapporteur
Madame	Brigitte	PLATEAU, Membre
Monsieur	Yves	ROBERT, Membre/Directeur de Thèse

L'ennui naquit un jour de l'uniformité.
Antoine Houdard de La Motte (1672-1731).

God only knows what I'd be without you.
The Beach Boys (Petsound).

Merci !

Factorisons. J'ai une chance incroyable. Tous les gens que je connais sont gentils. Je ne dis pas sympas. Ils sont *vraiment* gentils. À tel point que, sachant que le reste du monde n'est pas comme cela, je me demande parfois ce qui fait que j'ai eu la chance de me retrouver là et de les rencontrer. À défaut d'être originaux, ces remerciements sont sincères.

Tout d'abord, Merci à toi, Yves, pour m'avoir «débauché» et fait réaliser que ce que je cherchais était tout simplement à coté de moi, pour m'avoir appris que *le mieux est souvent l'ennemi du bien*, pour ta patience et pour m'avoir toujours laissé beaucoup de liberté.
Olivier, Merci à toi sans qui cette thèse serait sûrement bien creuse, merci à toi d'avoir une fois ou deux attendu mon retour, le lendemain, pour trouver avec moi la solution du problème qui nous taraudait depuis le milieu de l'après-midi... et merci pour ton délicieux bloc de fois-gras (Miam !).
Bref, Merci à tous les deux pour m'avoir accompagné durant ces trois années dont on aimerait qu'elles ne se terminent jamais tant elles étaient agréables.

Merci à toi, Henri, mon «troisième directeur de thèse», parce que ta façon d'aborder les choses m'a sans aucun doute profondément marqué et car j'ai comme l'impression que je n'ai pas fini d'entendre parler de SimGrid.

Merci, à vous, Jacques, Claire, Michel et Brigitte pour m'avoir fait l'honneur d'accepter d'être membres du jury de cette thèse. En particulier, Merci à toi, Claire, pour m'avoir gentiment fait remarquer que certains passages de la première version de cette thèse étaient vraiment incompréhensibles et m'avoir aidé à les clarifier.

Merci à toi, Martin, pour nos discussions enflammées sur la folie du monde qui nous entoure et sur le sens de nos recherches.

Merci à vous, Olivier, Guillaume et Vincent pour votre gentillesse, votre amitié, votre disponibilité, votre infinie patience, et vos paranoïas crépusculaires qui nous ont permis de passer quelques bonnes soirées, du moins je l'espère. Ne changez pas.

Merci à vous, les «petits derniers» qui passerez par là vous-aussi bientôt et qui avez fait briller mes yeux avec des girafes pleines d'une substance mielleuse : Abdou, Hélène, Antoine et Loris.

Merci à vous, Frédéric et Hervé, qui l'un et l'autre étaient toujours prêts à me débloquer quand il s'agissait de parler de graphes, d'hyper-graphes ou de séparateurs.

Merci à vous, Anne-Pascale, Corinne, Isabelle et Sylvie, pour votre bonne humeur et vos sourires qui ont un rôle indéniable dans l'ambiance de ce laboratoire. Merci à vous,

Eddy, Frédéric(s), Jean-Yves, Fabrice, Alain, ... Bref, Merci à tous les membres du LIP qui animent ce laboratoire et permettent de travailler dans d'aussi bonnes conditions.

Merci également à messieurs Knuth et Lamport sans qui ce document serait bien plus difficile à lire qu'il ne l'est déjà. Merci aux acteurs du logiciel libre sans qui tout ceci serait beaucoup moins drôle voire tout simplement impossible.

Enfin, Merci à mes beaux-parents de m'avoir laissé emmener leur fille. Merci à mes parents pour avoir fait de moi ce que je suis. Plus le temps passe, plus je me dis que je suis vraiment né au bon endroit. Merci à mon frère, parce que je l'aime, tout simplement. Et Merci à ma femme et mes enfants car ils sont ma seule raison d'être et que le reste n'est qu'amusement.

Table des matières

Chapitre 1

Introduction

Les besoins en puissance de calcul vont en s'accroissant dans une multitude de domaines (simulation et modélisation, traitement du signal, d'images, fouille de données, télé-immersion, etc.) et le parallélisme est une tentative de réponse toujours d'actualité. Partant du vieil adage selon lequel «l'union fait la force», dès les années 60, les premiers super-calculateurs ont fait leur apparition et ont connu leur heure de gloire jusque dans les années 90. L'effondrement des sociétés commercialisant les super-calculateurs s'est réalisé du fait de l'avènement d'architectures de type grappes de stations, bien moins coûteuses. Cette évolution n'a été possible que grâce à des efforts constants en terme de conception et de développement logiciel (notamment avec des bibliothèques comme ScaLAPACK, MPI ou PVM et le développement de la programmation asynchrone et des processus légers) et à la démocratisation des avancées technologiques présentes dans les machines parallèles propriétaires classiques (réseau haut-débit, processeur super-scalaires, unités arithmétiques spécialisées, architectures multiprocesseurs, etc.).

Les plates-formes de calcul parallèle ne se résument donc plus aux super-calculateurs monolithiques de Cray, IBM ou SGI. Les réseaux hétérogènes de stations (les machines parallèles du «pauvre» par excellence) sont monnaie courante dans les universités ou les entreprises. Faire coopérer un grand nombre de processeurs du commerce, agencés en grappes et reliés par un réseau plus ou moins rapide, s'avère nettement moins onéreux mais aussi nettement plus délicat. Ajoutons à cela que la tendance actuelle en matière de calcul distribué est à l'interconnexion de machines parallèles classiques et de grappes, réparties à l'échelle d'un continent, par des liens rapides afin d'agréger leur puissance de calcul (la fameuse *computing grid* décrite dans le livre de Foster et Kesselman [55]). Il apparaît donc clairement que l'hétérogénéité est une caractéristique essentielle des plates-formes de calcul d'aujourd'hui et de demain. La conception d'algorithmes adaptés – ou l'adaptation d'algorithmes existants – à ces nouveaux environnements parallèles est donc nécessaire et a motivé les travaux présentés dans cette thèse.

Noyaux d'algèbre linéaire dense

Poursuivant les travaux engagés par notre groupe, nous nous sommes d'abord intéressés à l'extension de techniques d'équilibrage de charge statiques permettant de prendre en compte l'hétérogénéité des ressources de calcul dans le cadre de noyaux d'algèbre linéaire dense. Ces algorithmes ayant été longuement étudiés dans le passé et étant bien maîtrisés dans un cadre homogène, il est naturel d'essayer de mesurer la complexité introduite par l'hétérogénéité de la plate-forme. Nous avons donc étudié et proposé de nouveaux schémas de distribution adaptés au produit de matrice *et* à la décomposition LU. Le problème de l'équilibrage de charge pour la factorisation LU est plus difficile que pour le produit de matrice car la portion de matrice sur laquelle on travaille diminue à chaque étape de l'algorithme.

Outre l'hétérogénéité des plates-formes actuelles, on trouve également comme caractéristique fréquente l'instabilité de leurs performances. En effet, sur une plate-forme homogène, les étapes de synchronisation imposées par certaines opérations de communication (la diffusion pipelinée par exemple), n'ont pas un impact énorme puisque si l'équilibrage de charge est bien effectué, tous les processeurs finissent leurs calculs et sont prêts à communiquer en même temps. Sur une plate-forme hétérogène les choses ne sont pas aussi simples : les mécanismes de pipeline classiques sont ralentis par les machines et les liens les plus lents, et la vitesse des machines est plus difficile à évaluer. Obtenir un bon équilibrage de charge qui soit durable dans le temps devient une gageure. Il est donc indispensable de pouvoir rééquilibrer la charge en cours de calcul. Une redistribution complète des données pour passer d'une distribution optimale à une autre n'étant généralement pas envisageable en raison de la quantité de communications que cela engendrerait (elle pourrait être très supérieure à la quantité totale de communications générée par l'algorithme), nous avons proposé des mécanismes de redistributions légères pour le produit de matrice, fondés sur les distributions statiques, pouvant être effectués entre chaque étape de calcul et permettant de rééquilibrer la charge de façon optimale.

Ces travaux sont présentés dans le chapitre 2 et ont fait l'objet de publications dans [4, 6, 7, 8, 16, 18, 19].

Ordonnancement de tâches indépendantes

La paradigme maître-esclave consiste en l'exécution de tâches indépendantes par un ensemble de processeurs, appelés *esclaves*, sous la houlette d'un processeur particulier, le *maître*. Cette technique est utilisée depuis longtemps en raison de sa stabilité et de la simplicité de sa mise en œuvre. Les versions les plus simples présentées dans la littérature ne sont cependant pas efficaces sur une plate-forme où les processeurs n'ont pas tous la même puissance de calcul et où les temps de communication du maître vers les esclaves ne sont pas uniformes. Il est nécessaire de prendre en compte cette hétérogénéité pour utiliser au mieux la puissance de calcul disponible.

Nous nous sommes donc intéressés à l'extension de ce paradigme aux plates-formes hétérogènes. Nous appuyant sur des modélisations déjà existantes, nous avons proposé

un algorithme polynomial qui donne une solution optimale au problème de l'allocation de tâches indépendantes de caractéristiques identiques lorsqu'une seule communication est nécessaire avant le traitement des tâches sur les différents processeurs. Lorsqu'une communication avant et une communication après le traitement des tâches sont nécessaires, la situation est plus complexe. Le problème est NP-complet mais il existe un algorithme d'approximation polynomial. Enfin, nous avons montré que quand une communication est nécessaire avant le traitement de chacune des tâches, le problème redevient polynomial. Ces problèmes qui se résolvaient trivialement dans un cadre homogène sont nettement plus compliqués dans un cadre hétérogène. Néanmoins, si la difficulté de ces problèmes provient en grande partie du caractère hétérogène de la plate-forme, elle provient également du fait que la quantité de travail à effectuer est finie et discrète et que l'objectif que l'on cherche à atteindre consiste à traiter les données en un temps minimal (minimisation de *makespan*).

Ces travaux sont présentés dans le chapitre 3 et ont fait l'objet de publications dans [3, 15, 17].

Ordonnancement en régime permanent

Les trois chapitres suivants montrent que la métrique de minimisation du *makespan* n'est pas forcément adaptée à de telles applications et à de telles plates-formes. En effet, en raison du temps de déploiement et de la difficulté de mise en œuvre sur de telles plates-formes, il n'est rentable de déployer que de grosses applications. Ces dernières exhibent souvent une certaine régularité dont il est possible de tirer parti. Notamment, en s'intéressant, non pas au *makespan* mais en se plaçant en régime permanent (situation qui se produit nécessairement dès que le nombre de tâches à traiter devient important), les problèmes du chapitre précédent sont beaucoup plus simples à traiter et il est possible d'en déduire des ordonnancement dont le temps d'exécution est asymptotiquement optimal. Le régime permanent est calculé en formulant un programme linéaire qui permet de prendre très simplement en compte la présence de plusieurs maîtres, l'hétérogénéité des capacités de calcul et de communications des différents processeurs, l'affinité des tâches avec certains types de machines et la topologie de la plate-forme. Nous donnons également une description polynomiale de l'ordonnancement permettant d'atteindre ce régime permanent, et nous en montrons l'optimalité asymptotique.

Dans certaines situations, le calcul du régime permanent optimal peut être calculé directement sans passer par la résolution d'un programme linéaire en utilisant une stratégie orientée bande-passante. Ce type de résultat peut alors être simplement utilisé pour concevoir des ordonnancements non centralisés robustes tirant partie d'informations statiques.

Le chapitre 4 introduit différents modèles de plate-forme et présente des résultats d'optimalité asymptotique pour l'ordonnancement de tâches indépendantes de caractéristiques identiques. Ces travaux ont fait l'objet de publications dans [12, 13, 20].

Le chapitre 5 présente l'extension de ces résultats au cas où les tâches à répartir recèlent du parallélisme interne. Ils ont été publiés dans [1, 10].

Le chapitre 6 présente l'adaptation des techniques précédentes au modèle des tâches divisibles. Dans ce modèle, les tâches sont indépendantes et infiniment fractionnables. À la différence des travaux précédents la granularité est donc aussi fine qu'on le souhaite. Le succès rencontré par ce modèle provient du fait que des algorithmes optimaux et des formes closes existent pour les instances simples de ce modèle. Nous étendons un certain nombre de résultats au cadre hétérogène et proposons des stratégies asymptotiquement optimales. Ces travaux ont fait l'objet de publications dans [2, 11].

Modélisation et simulation de plates-formes hétérogènes

S'il est difficile de garantir les performances d'une heuristique, il est également difficile de valider expérimentalement son efficacité. Cette étape est pourtant souvent nécessaire pour pouvoir comparer objectivement ses algorithmes à ceux proposés dans la littérature. S'il était possible de recourir à des expériences grandeur nature quand on se plaçait dans un cadre homogène, ce n'est plus le cas dans un cadre hétérogène. En effet, dans le cas d'une plate-forme de calcul hétérogène et distribuée, de telles expériences sont très délicates à mener en raison de son instabilité latente. Il est impossible de garantir que l'état d'une plate-forme de calcul qui n'est pas entièrement dédiée à l'expérimentation va rester le même entre deux expériences, ce qui empêche donc toute comparaison rigoureuse. On utilise donc des simulations afin d'assurer la reproductibilité des expériences, toute la difficulté étant alors d'arriver à simuler un tel environnement de façon réaliste.

En effet, la plupart des résultats d'ordonnancement que l'on peut trouver dans la littérature sont obtenus grâce à des hypothèses assez restrictives et à des modèles assez simples : les réseaux d'interconnexion sont souvent très simplistes en regard de la réalité ; les ressources de calcul et de communication sont souvent supposées ne pas avoir de variation de performances ; il est toujours possible d'obtenir des prédictions parfaites des différences de vitesses des ressources. Ces hypothèses simplificatrices sont nécessaires à la compréhension de certains phénomènes mais ne sont jamais vérifiées en pratique et les heuristiques qui en découlent sont rarement confrontées à la réalité.

Pour parvenir à une comparaison honnête des algorithmes, une approche efficace consiste à effectuer des simulations utilisant des traces, c'est-à-dire utilisant des enregistrements de différents paramètres d'une plate-forme réelle pour obtenir un comportement réaliste. Le chapitre 7 présente quelques réflexions sur le thème de la modélisation et de la simulation ainsi que le logiciel qui en a découlé. SimGrid est un logiciel que nous avons développé en collaboration avec Henri Casanova de l'Université de Californie, San Diego. C'est un simulateur modulaire écrit en C et permettant de simuler une application distribuée où les décisions d'ordonnancement peuvent être prises par différentes entités. La force de ce simulateur réside dans sa capacité à importer et simuler aisément des plates-formes réalistes (de la grille de metacomputing au réseau de stations de travail). Ces travaux ont fait l'objet de publications dans [9, 21]

Chapitre 2

Équilibrage de charge pour l'algèbre linéaire

2.1 Introduction

La parallélisation des algorithmes d'algèbre linéaire est étudiée depuis plus d'une vingtaine d'années en raison de l'utilité de ces derniers dans des domaines aussi variés que la mécanique du solide, des fluides, les simulations météorologiques, la chimie, etc. L'efficacité de leur mise en œuvre est donc cruciale pour un large champ d'applications. Cependant, l'architecture des machines a évolué et si bon nombre de principes restent encore valables (*pipelining*, recouvrement calcul/communication, etc.), ces évolutions rendent les algorithmes parallèles classiques d'algèbre linéaire inadaptés aux nouvelles plates-formes de calcul. Les hypothèses relatives aux architectures servant de base à ces algorithmes ne sont plus réalistes et les problèmes liés à l'algorithmique hétérogène sont encore mal maîtrisés.

Nous nous intéressons dans ce chapitre à l'extension de techniques d'équilibrage de charge statiques permettant de prendre en compte l'hétérogénéité des ressources de calcul, notamment dans le cadre de noyaux d'algèbre linéaire dense. Dans la section 2.2, nous rappelons les résultats connus sur la difficulté de la distribution des données pour le produit de matrices. Dans la section 2.2.4, nous présentons également une validation expérimentale de l'efficacité des schémas de distribution présentés en section 2.2. Dans la section 2.3, nous présentons rapidement les schémas de distribution classiques pour les factorisations LU puis nous expliquons comment prendre en compte l'hétérogénéité des ressources de calcul. Nous présentons une distribution bidimensionnelle (asymptotiquement optimale) des données assurant un bon équilibrage de charge tout au long du calcul. Le problème de l'équilibrage de charge pour la factorisation LU est plus difficile que pour le produit de matrice car la portion de matrice sur laquelle on travaille diminue à chaque étape de l'algorithme. Dans la section 2.3.5, nous comparons à l'aide de simulations les distributions proposées dans la section 2.3. Enfin, dans la section 2.4, nous présentons une technique de rééquilibrage de charge pour le produit matriciel, puis nous concluons

en section 2.5.

2.2 Algorithme parallèle du produit matriciel

Dans cette section, nous présentons dans un premier temps l'algorithme classique du produit matriciel sur une grille homogène. Ensuite nous montrons comment adapter cet algorithme à des plates-formes de calculs hétérogènes et enfin, nous présentons des résultats de complexité relatifs au problème de l'équilibrage de charge du produit matriciel sur une plate-forme hétérogène.

2.2.1 Produit matriciel sur une grille homogène

Si on dispose d'une grille composée de $p \times p$ processeurs identiques, l'algorithme de produit matriciel mis en œuvre dans la librairie ScaLAPACK est une version par blocs de la *double-diffusion*[1] décrite dans [1, 56, 73] et qui se déroule de la façon suivante.

Supposons que l'on souhaite effectuer le produit de deux matrices carrées de taille n. Dans ce cas, les trois matrices sont disposées de la même façon sur la grille : le processeur $P_{i,j}$ est donc responsable des sous-matrices $A_{i,j}$, $B_{i,j}$ et $C_{i,j}$.

Le produit se déroule en n étapes. À l'étape k, la $k^{ème}$ colonne de A est diffusée horizontalement et la $k^{ème}$ ligne de B est diffusée verticalement. Chaque processeur détient donc un fragment de colonne de A et un fragment de ligne de B de tailles n/p et met à jour la partie de C dont il est responsable à l'aide de ces vecteurs.

Cet algorithme est celui qui est mis en œuvre dans la bibliothèque ScaLAPACK car il est extensible, efficace et ne suppose pas de permutation initiale des données (à la différence de l'algorithme de Cannon [73] par exemple).

Bien évidemment, ScaLAPACK utilise une version par blocs de cet algorithme : chaque coefficient de la description est remplacé par un bloc $r \times r$ de matrice, la valeur optimale de r dépendant du rapport entre les temps de calcul et de communication. Enfin, il est important de remarquer que, même si la distribution utilisée est de type grille, il n'est absolument pas nécessaire de disposer de processeurs interconnectés en grille : le produit est effectué sur une grille *logique* de processeurs.

On notera par la suite τ le temps de communication d'un élément entre deux machines, et donc τr^2 représentera le temps nécessaire à la communication d'un bloc de taille r. On notera également $t_c r^3$ le temps nécessaire au produit de deux blocs carrés de taille r.

On ne s'intéressera dans ce chapitre qu'à deux cas idéalisés de réseau : celui où toutes les communications s'effectuent séquentiellement (noté avec un $\sum$) et celui où, au contraire, le réseau permet de réaliser toutes les communications en parallèle (noté

[1] ScaLAPACK préconise l'utilisation d'une grille plutôt qu'une ligne de processeurs pour des raisons d'extensibilité [20].

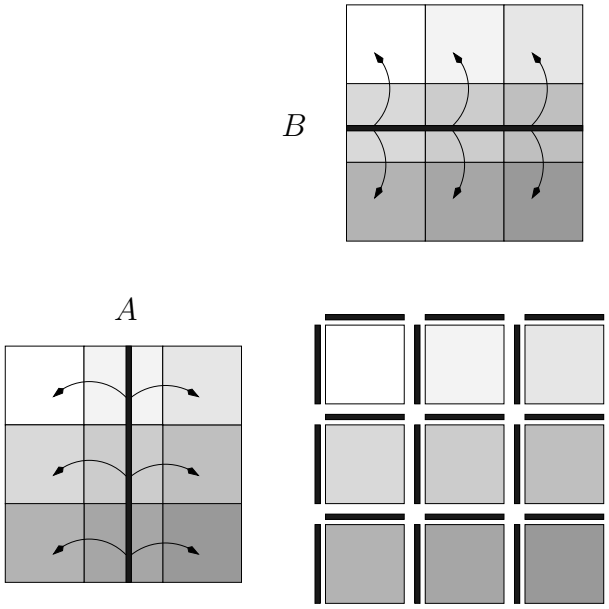


FIG. 2.1 – Algorithme du produit matriciel parallèle sur une grille 3×3

avec un max). Dans le premier cas, le temps nécessaire au calcul du produit de deux matrices de tailles n en utilisant l'algorithme précédent est :

$$\begin{aligned} C_{\Sigma} &= \sum_{k=1}^{n} \left[\underbrace{t_c \left(\frac{n}{p}\right)^2}_{\text{calculs}} + \underbrace{\sum_{i=1}^{p} \sum_{j \neq k} \tau \left(\frac{n}{p}\right)}_{\text{communications horizontales}} + \underbrace{\sum_{j=1}^{p} \sum_{i \neq k} \tau \left(\frac{n}{p}\right)}_{\text{communications verticales}} \right], \\ &= \sum_{k=1}^{n} \left[t_c \left(\frac{n}{p}\right)^2 + \tau(p-1)n + \tau(p-1)n \right], \\ &= t_c \frac{n^3}{p^2} + 2(p-1)n^2\tau. \end{aligned} \tag{2.1}$$

Si le réseau permet de réaliser les communications en parallèle, le temps de calcul est le suivant :

$$\begin{aligned} C_{\max} &= \sum_{k=1}^{n} \left[\underbrace{t_c \left(\frac{n}{p}\right)^2}_{\text{calculs}} + \underbrace{\max_{i=1}^{p} \max_{j \neq k} \tau \left(\frac{n}{p}\right)}_{\text{communications horizontales}} + \underbrace{\max_{j=1}^{p} \max_{i \neq k} \tau \left(\frac{n}{p}\right)}_{\text{communications verticales}} \right], \\ &= \sum_{k=1}^{n} \left[t_c \left(\frac{n}{p}\right)^2 + \tau \frac{n}{p} + \tau \frac{n}{p} \right], \\ &= t_c \frac{n^3}{p^2} + 2\frac{n^2}{p}\tau. \end{aligned} \tag{2.2}$$

Dans les deux cas, l'équilibrage de charge est parfait car chaque processeur est responsable d'une surface qui est exactement proportionnelle à sa vitesse et donc le temps de calcul de chaque étape est égal à $t_c \left(\frac{n}{p}\right)^2$.

2.2.2 Produit matriciel sur une plate-forme hétérogène

Si on dispose de p processeurs identiques de temps de cycle t_c et si on note s_i la fraction de surface dont est responsable le processeur i, h_i sa hauteur et w_i sa largeur, on peut remarquer que l'expression (2.1) s'écrit plus généralement :

$$C_{\sum} = \left(\max_{i \in [\![1,p]\!]} t_c s_i \right) n^3 + \left(\sum_{i=1}^{p} \tau(h_i + w_i) \right) n^2 \tag{2.3}$$

Le premier terme ($\max_{i \in [\![1,p]\!]} t_c s_i$) représente l'efficacité de l'équilibrage de charge. Une bonne distribution doit donc minimiser ce facteur. Le second terme ($\sum_{i=1}^{p}(h_i + w_i)$) représente la quantité de communications à chaque étape. En effet, chaque processeur a besoin à chaque étape de nh_i données de A et nw_i données de B. La sommation de ces termes est due au réseau d'interconnexion qui ne permet pas de réaliser de communications en parallèle. Dans le cas contraire, la forme générale du coût de l'algorithme est :

$$C_{\max} = \left(\max_{i \in [\![1,p]\!]} t_c s_i \right) n^3 + \left(\max_{i \in [\![1,p]\!]} \tau(h_i + w_i) \right) n^2 \tag{2.4}$$

Si la vitesse des processeurs est homogène, le premier terme est minimisé en attribuant à chaque processeur la même quantité de travail ($s_i = c^{te}$) et le second en les organisant en grille bidimensionnelle. Dans le cas où les processeurs ne sont pas de même puissance, il suffit de remplacer t_c par un $t_c^{(i)}$ dépendant du processeur mais nous verrons dans la section 2.2.3 que le problème de minimisation est alors beaucoup plus complexe.

2.2.3 Problèmes d'optimisation et complexité

Comme nous l'avons remarqué dans la section 2.2.2, dans le cas où les processeurs sont tous identiques, les surfaces allouées à chaque processeur doivent être égales afin d'obtenir un équilibrage de charge parfait. Supposons que nos p processeurs aient des temps de calcul respectifs $(t_c^{(i)})_{i=1\ldots p}$ et soient responsables de surfaces $(s_i)_{i=1\ldots p}$. Alors, à chaque étape, le processeur i effectue ses mises à jour pendant un temps $t_c^{(i)} s_i$. Il faut donc $t_c^{(1)} s_1 = t_c^{(2)} s_2 = \cdots = t_c^{(p)} s_p$ pour minimiser $\max_i t_c^{(i)} s_i$, c'est-à-dire avoir un équilibrage optimal à chaque étape. Les vitesses relatives des processeurs étant fixées, on peut déterminer les surfaces qui doivent être attribuées à chaque processeur (on notera par la suite s_i la vitesse normalisée du processeur i et $t_i = 1/s_i$ son temps de cycle «normalisé»). Étant donné qu'il est toujours possible de satisfaire la condition précédente en attribuant à chaque processeur un nombre de colonnes proportionnel à sa vitesse, il convient de minimiser le second terme de la complexité calculée en section 2.2.2, c'est-à-dire la quantité

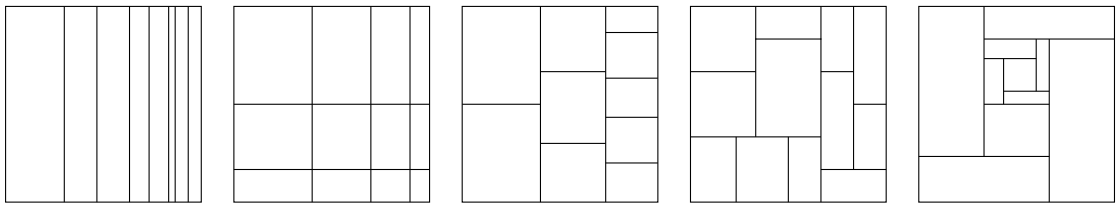


FIG. 2.2 – Taxonomie des partitions du carré unité

de communications. C'est donc en respectant ces contraintes de surfaces que l'on doit minimiser les communications.

À chaque étape, un processeur reçoit une quantité de données proportionnelle à la largeur et à la longueur de la zone dont il est responsable (un fragment de ligne de blocs et un fragment de colonne de blocs). On peut donc se ramener au problème de la partition du carré unité en p rectangles de surfaces fixées. Un certain nombre de types de partition sont envisageables (voir figure 2.2). Pour chacun de ces types de partitions, on envisagera deux types (extrêmes) de réseaux de communication et donc deux fonctions objectifs distinctes. Si le réseau ne permet pas d'effectuer de communications en parallèle, la fonction objectif à minimiser est $\hat{C} = \sum_{i=1}^{p}(h_i + w_i)$ et dans le cas contraire, la fonction objectif est $\hat{M} = \max_{i=1}^{p}(h_i + w_i)$. Nous nous sommes donc ramenés aux problèmes d'optimisations suivants en fonction du réseau sous-jacent :

Définition 2.1 (Hétérogène 1D). *Étant donné p réels positifs $s_1, \ldots, s_p$ (représentant les vitesses normalisées des processeurs) tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition «unidimensionnelle» du carré unité en p rectangles R_i de surfaces s_i.*

Ce problème est trivial puisqu'il suffit d'allouer à chaque processeur une zone de hauteur 1 et de largeur s_i. On notera qu'ici, l'hétérogénéité n'a aucune influence sur le schéma de communications.

Définition 2.2 (Hétérogène 2D). *Étant donné p^2 réels positifs $s_1, \ldots, s_{p^2}$ (représentant les vitesses normalisées des processeurs) tels que $\sum_{i=1}^{p^2} s_i = 1$, trouver une partition «en grille» du carré unité en p rectangles R_i de surfaces s_i, c'est-à-dire $r_1, \ldots, r_p, c_1, \ldots, c_p$ tels que $\sum_{i=1}^{p} r_i = \sum_{j=1}^{p} c_j = 1$ et une bijection f de $[\![1,p]\!] \times [\![1,p]\!]$ sur $[\![1,p^2]\!]$ minimisant $\max_{(i,j)\in[\![1,p]\!]\times[\![1,p]\!]} r_i t_c^{(f(i,j))} c_j$*

Ce problème apparaît naturellement lorsque l'on cherche à étendre la distribution en grille présentée à la section 2.2.1 à un cadre hétérogène.

Définition 2.3 (Péri-Sum). *Étant donné p réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition du carré unité en p rectangles R_i de surfaces s_i et de dimensions $h_i \times w_i$ telle que $\hat{C} = \sum_{i=1}^{p}(h_i + w_i)$ soit minimum.*

Ce problème est celui qui apparaît quand toutes les communications sont séquentielles. Le coût des communications à chaque étape est donc proportionnel à la somme des demi-périmètres des zones allouées aux processeurs. On ne s'impose aucune contrainte sur le type de partition. Péri-Sum est NP-complet par réduction à 2-Partition mais il existe une heuristique polynomiale garantie à un facteur 5/4 dont les solutions sont récursives ([12]).

Définition 2.4 (Col-Péri-Sum). *Étant donné p réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition du carré unité en p rectangles R_i de surfaces s_i et de dimensions $h_i \times w_i$ organisée en colonnes telle que $\hat{C} = \sum_{i=1}^{p}(h_i + w_i)$ soit minimum.*

Les hypothèses effectuées sur le réseau d'interconnexion sont les mêmes que pour Péri-Sum mais on impose aux solutions d'être organisées en colonnes. Col-Péri-Sum se résout en temps polynomial grâce à une méthode de programmation dynamique et ses solutions sont expérimentalement proches de l'optimum absolu de Péri-Sum [12].

Définition 2.5 (Péri-Max). *Étant donné p réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition du carré unité en p rectangles R_i de surfaces s_i et de dimensions $h_i \times w_i$ telle que $\hat{M} = \max_{i=1}^{p}(h_i + w_i)$ soit minimum.*

Ici, on s'intéresse au cas où toutes les communications point-à-point peuvent se faire en parallèle pourvu qu'elles ne mettent pas en jeu les mêmes machines. Le coût des communications à chaque étape est donc approximativement proportionnel au maximum des demi-périmètres des zones allouées aux processeurs. On ne s'impose aucune contrainte sur le type de partition. Péri-Max est NP-complet par réduction à une variante de 2-Partition mais il existe une heuristique polynomiale garantie à un facteur $2/\sqrt{3}$ qui fournit des solutions organisées en colonnes [11].

Définition 2.6 (Col-Péri-Max). *Étant donné p réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition du carré unité en p rectangles R_i de surfaces s_i et de dimensions $h_i \times w_i$ organisée en colonnes telle que $\hat{M} = \max_{i=1}^{p}(h_i + w_i)$ soit minimum.*

Les hypothèses effectuées sur le réseau d'interconnexion sont les mêmes que pour Péri-Max mais on impose aux solutions d'être organisées en colonnes. Col-Péri-Max est NP-complet mais il existe une heuristique polynomiale garantie à un facteur $2/\sqrt{3}$ (la même que pour Péri-Max).

Définition 2.7. *Une distribution vérifiant que les $h_i t_i w_i$ sont égaux entre eux et valent tous 1 sera dite* parfaitement équilibrée.

Les solutions de Hétérogène 2D sont parfaitement équilibrées si et seulement s'il existe un arrangement des vitesses relatives en une matrice de rang 1. En revanche, les solutions de Hétérogène 1D, Péri-Sum, Col-Péri-Sum, Péri-Max ou de Col-Péri-Max sont toujours parfaitement équilibrées. Nous verrons plus tard qu'un équilibrage parfait pour le produit de matrice permet d'obtenir un équilibrage asymptotiquement optimal pour LU.

	1D	2D	En colonne	Récursive	Quelconque
$\sum$	Polynomial	NP-complet [10]	Polynomial [12]	Pas de résultat connu.	NP-complet. Heuristique garantie à un facteur 5/4. [11]
max			NP-complet [10] Heuristique garantie à un facteur $2/\sqrt{3}$.	Pas de résultat connu.	NP-complet. Heuristique garantie à un facteur $2/\sqrt{3}$. [11]

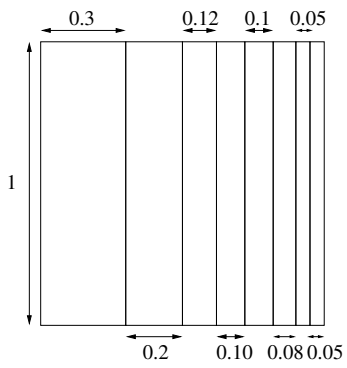
FIG. 2.3 – Résultats de complexité

Il existe donc un certain nombre d'heuristiques «naturelles» pour approcher Péri-Sum et Péri-Max. Cependant, il est assez difficile de les garantir. Deux heuristiques efficaces en colonnes (assez simples à mettre en œuvre) pour Péri-Sum et Péri-Max sont proposées dans [12]. Une heuristique récursive pour Péri-Sum est proposée dans [11] et garantie à un facteur 5/4. Les résultats pour tous ces problèmes sont résumés dans le tableau de la figure 2.3.

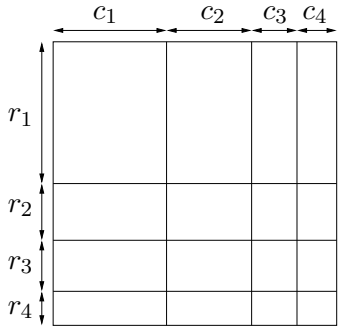
Un certain nombre de travaux portaient déjà sur le problème de la mise en œuvre efficace du produit de matrices sur des plates-formes hétérogènes mais aucun, mis à part ceux que nous venons de présenter, n'a à ce jour fourni de résultats de complexité ni de garantie sur les algorithmes proposés. Par exemple, Kalinov et Lastovetky [70] regroupent certains processeurs (arbitrairement) en colonnes, équilibrent dans un premier temps la charge entre les différentes colonnes de processeurs puis, dans un second temps, la charge dans chaque colonne (voir figure 2.4(c)). Une autre approche est proposée par Crandall et Quinn dans [41]. Ils proposent un algorithme récursif permettant de partitionner le carré unité en p rectangles d'aire $s_1, \ldots, s_p$ tout en "minimisant" le coût des communications (la somme des demi-périmètres). Cette heuristique n'est cependant pas garantie et peut même amener à de moins bons résultats que la solution de Col-Péri-Sum (voir figure 2.4(d)). Enfin, dans [69], Kaddoura, Ranka et Wang améliorent l'algorithme de Crandall et de Quinn [41] et proposent quelques variations mais n'apportent toujours pas de garanties sur les performances de leurs algorithmes.

2.2.4 Validation expérimentale

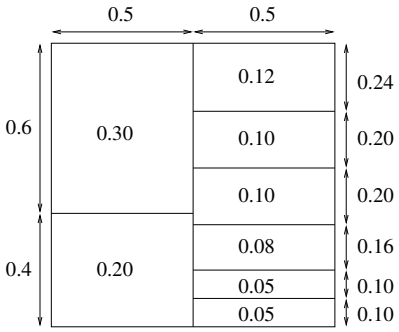
Nous avons mis en œuvre et comparé 5 heuristiques de distribution en utilisant la bibliothèque de communications MPICH [63]. L'objectif de ces expériences est principalement de démontrer le gain potentiel apporté par ces types de distributions. Le protocole



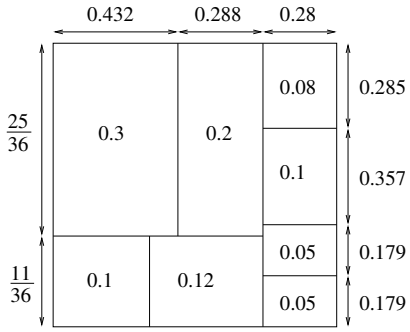
(a) Hétérogène 1D : $\hat{C} = 7$



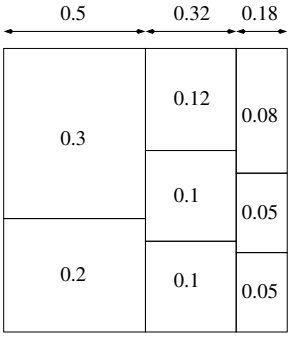
(b) Hétérogène 2D



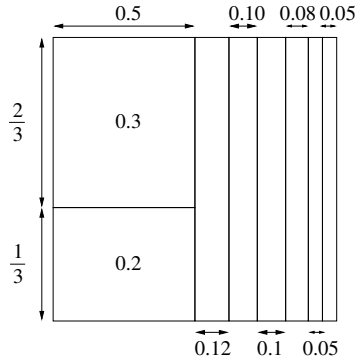
(c) Kalinov : $\hat{C} = 6$



(d) Heuristique récursive de Crandall et Quinn : $\hat{C} = 5.56$



(e) Col-Péri-Sum : $\hat{C} = 5.5$



(f) Col-Péri-Max : $\hat{M} = 1.133$

FIG. 2.4 – Illustration de divers problèmes d'optimisation et heuristiques

expérimental est le suivant : chaque processeur évalue sa vitesse en calculant un produit de matrices de tailles raisonnables ; ces évaluations sont rassemblées sur un processeur qui en déduit la distribution adaptée à l'heuristique que l'on cherche à évaluer ; finalement, la distribution est fournie à chaque processeur et plusieurs produits sont enfin calculés. C'est le temps moyen de ces derniers produits qui est présenté dans les mesures suivantes.

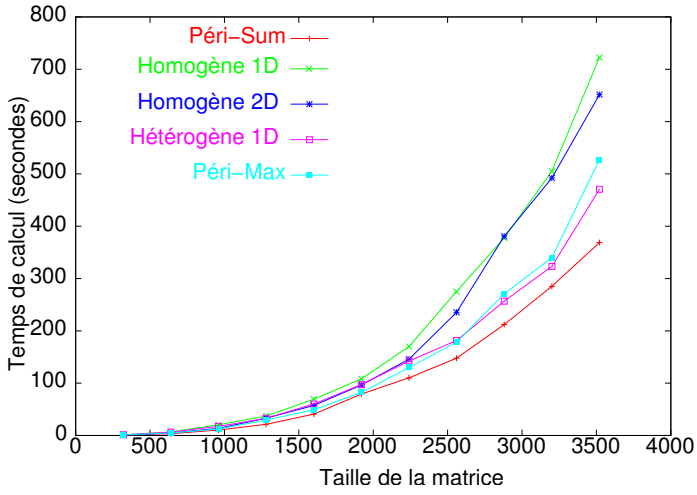
La première série d'expériences a été réalisée sur un réseau de type Fast-Ethernet avec le protocole TCP. La qualité d'interconnexion d'un tel réseau étant relativement faible, la fonction de coût à minimiser s'appuie donc sur une somme. La figure 2.5 montre le temps moyen d'exécution d'un produit de matrices pour différentes distributions, différentes tailles et différents nombres de machines. Deux Pentium III 550 MHz et deux Pentium PRO 200 MHz ont été utilisés pour réaliser les mesures de la figure 2.5(a). Les distributions tenant compte de l'hétérogénéité des machines sont bien meilleures que les distributions classiques (homogène unidimensionnelle et homogène bidimensionnelle). La distribution solution de Col-Péri-Sum donne effectivement les meilleurs résultats. Deux Pentium II 350 MHz supplémentaires ont été utilisés pour les expériences de la figure 2.5(b). La différence entre les distributions unidimensionnelles et les autres commence à apparaître. Les distributions unidimensionnelles sont moins bonnes que toutes les autres (hétérogènes ou non).

Enfin, une grande variété de machines (1 P-PRO 200 MHz, 2 P-II 350 MHz, 2 P-II 400 MHz, 3 P-II 500 MHz, 2 P-III 550 MHz) a été utilisée pour les résultats de la figure 2.5(c). Dans ce dernier cas, l'optimisation des communications devient cruciale et la supériorité des distributions bidimensionnelles par rapport aux autres devient flagrante. Encore une fois, c'est la distribution solution de Col-Péri-Sum qui est la plus efficace.

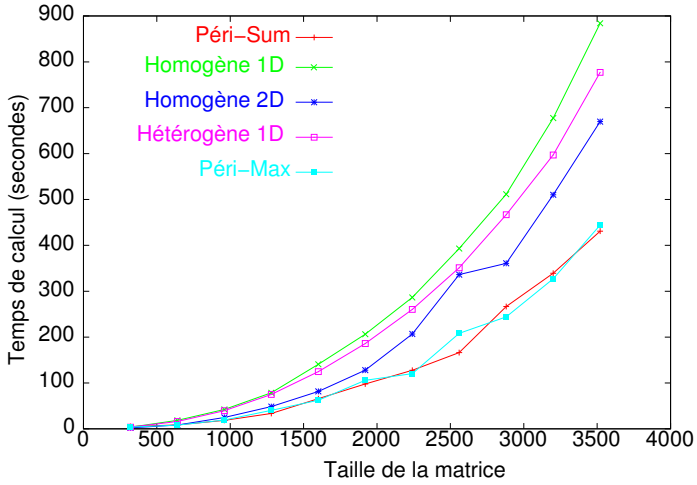
La seconde série d'expériences a été réalisée sur un réseau de type Myrinet avec le protocole BIP. La fonction de coût à minimiser s'appuie donc plutôt sur un max que sur une somme. La figure 2.6 montre le temps moyen d'exécution d'un produit de matrices pour différentes distributions, tailles et nombres de nœuds. La machine utilisée est composée de 12 Pentium Pro 200 MHz. Cette machine étant homogène, nous avons simulé l'hétérogénéité en utilisant pour certains des nœuds un produit efficace et pour d'autres un produit particulièrement lent.

Quatre nœuds (deux rapides et deux lents) ont été utilisés pour réaliser les mesures de la figure 2.6(a). Les distributions tenant compte de l'hétérogénéité des machines sont encore une fois bien meilleures que les distributions classiques (homogène unidimensionnelle et homogène bidimensionnelle). Les distributions solutions de Col-Péri-Max et de Hétérogène 1D donnent les meilleurs résultats. Deux nœuds rapides ont été rajoutés pour les expériences de la figure 2.6(b). La distribution Hétérogène 1D est la meilleure distribution et les distributions homogènes ont de très mauvaises performances. Enfin, six nœuds rapides et quatre nœuds lents sont utilisés pour l'expérience de la figure 2.6(c). Ici encore, les distributions hétérogènes sont largement plus performantes que les autres. Plus les performances du réseau de communication sont bonnes, plus il est important de prendre en compte l'hétérogénéité de vitesse des machines car cela devient le facteur déterminant.

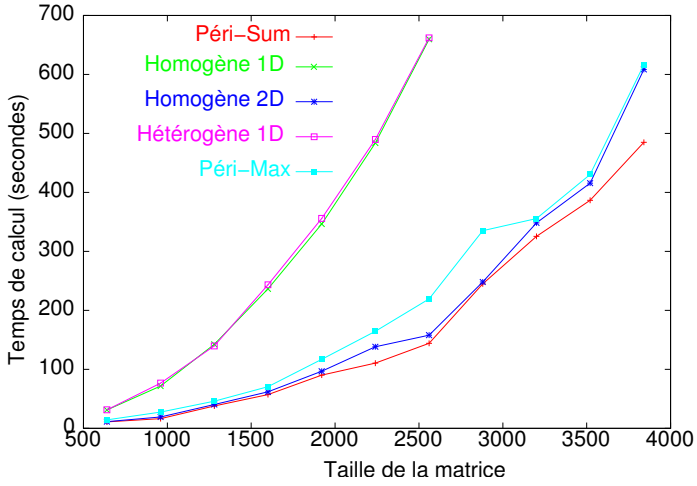
Ces expériences démontrent la supériorité des distributions hétérogènes dans deux



(a) Sur 4 machines : l'équilibrage de charge est encore prépondérant sur les communications

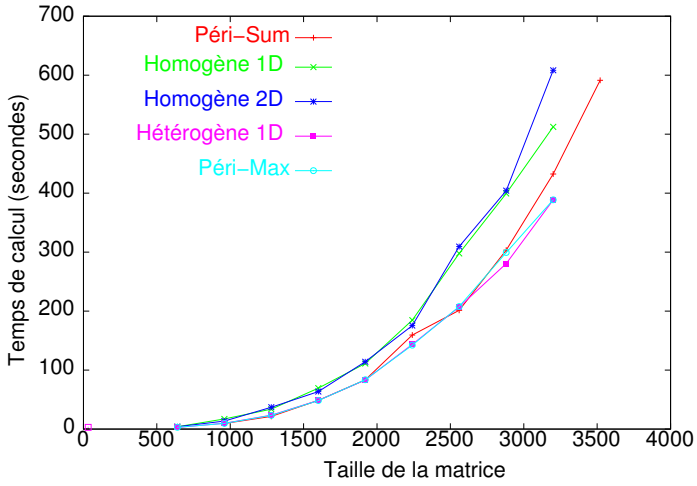


(b) Sur 6 machines : les communications prennent de l'importance

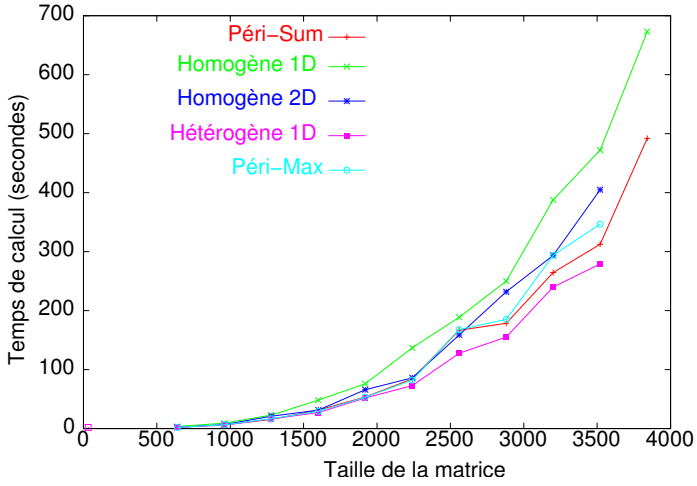


(c) Sur 10 machines : on voit clairement la différence entre les distribution unidimensionnelles et les autres

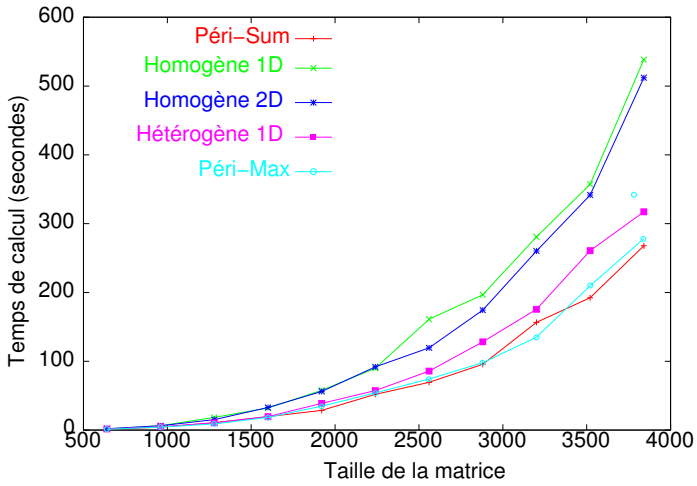
FIG. 2.5 – Temps moyen du produit matriciel sur un réseau de stations de vitesses hétérogènes (réseau lent).



(a) Sur 4 nœuds : l'équilibrage de charge est prépondérant sur les communications



(b) Sur 6 nœuds : le coût des communications est encore peu important et les performances dépendent de l'équilibrage de charge



(c) Sur 10 nœuds : les heuristiques de partitionnement hétérogène donnent des résultats largement supérieurs aux distributions classiques

FIG. 2.6 – Temps moyen du produit matriciel sur une grappe de PCs interconnectés par Myrinet (réseau rapide)

cas extrêmes (mais néanmoins courants) : un réseau composé de stations de vitesses très différentes et doté d'une couche de communication relativement lente ; une grappe interconnectée par un réseau rapide.

2.3 Factorisation LU

Les méthodes directes de résolution de systèmes linéaires à base de factorisation matricielle exhibent d'excellentes propriétés numériques. Elles permettent de résoudre des systèmes linéaires en résolvant des systèmes intermédiaires moins coûteux et plus stables, ce qui permet généralement d'améliorer la convergence de méthodes numériques itératives tout en conservant des propriétés importantes (spectre, déterminant, etc.).

Dans cette section, nous montrons comment étendre l'algorithme de factorisation LU utilisé dans ScaLAPACK [20] pour prendre en compte l'hétérogénéité des ressources.

2.3.1 Principe des factorisations

La factorisation LU d'une matrice A consiste à déterminer L triangulaire inférieure avec des 1 sur la diagonale et U triangulaire supérieure telles que $A = LU$. Pour ce faire on effectue une élimination de Gauss, c'est-à-dire une application récursive de l'identité suivante :

$$A_n = \begin{pmatrix} a_{11} & {}^t w \\ v & A_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & {}^t w \\ 0 & A_{n-1} - (v/a_{11}).{}^t w \end{pmatrix} \tag{2.5}$$

Cette égalité n'est cependant valable que si a_{11} (le *pivot*) est non nul et on est donc obligé de procéder à des permutations de lignes. De plus, la taille du pivot influant énormément sur la stabilité numérique de l'algorithme, on essaie toujours d'utiliser une grande valeur en tant que pivot. La recherche du pivot ainsi que sa propagation aux différents processeurs se fait en temps linéaire (proportionnel à la taille de A_n) à chaque étape. Cependant le calcul du *complément de Schur* de A_n par rapport au pivot a_{11}, c'est-à-dire le terme $A_{n-1} - (v/a_{11}).{}^t w$, s'effectue en temps quadratique avec la taille de A_{n-1} et la complexité de la $k^{ème}$ étape (détermination du pivot, propagation, calcul du complément) est donc $\Theta((n-k)^2)$ si n est la taille de A.

L'algorithme de la factorisation LU a donc en commun avec l'algorithme du produit matriciel qu'il consiste en une suite de mises-à-jour (c'est à dire des opérations du type $X \leftarrow X + u.{}^t v$) et que ces mises à jour constituent la majorité du coût de l'algorithme. Comme pour le produit de matrice, il est donc indispensable que chacune de ces mises-à-jours soit bien équilibrée sur l'ensemble des processeurs. Mais à la différence du produit matriciel, la fraction de matrice sur laquelle on travaille diminue en largeur et en hauteur (voir figure 2.7) à chaque étape. La répartition des données doit par conséquent être prévue pour tenir compte de cette caractéristique et assurer un équilibrage de charge constant tout au long du calcul. En effet, si la distribution était identique à celle exposée

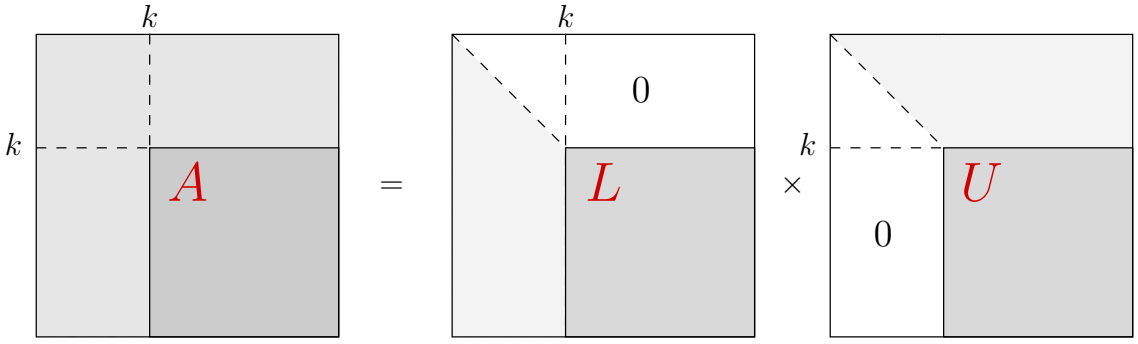


FIG. 2.7 – Principe de la factorisation LU

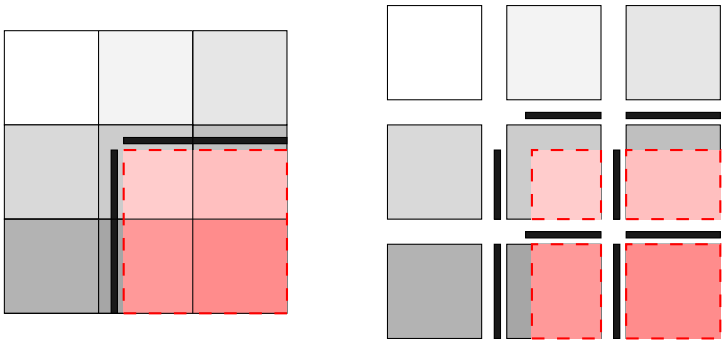
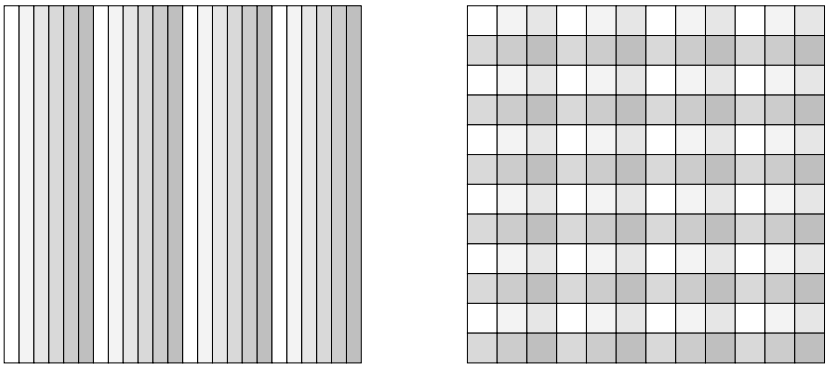


FIG. 2.8 – Déséquilibre des calculs dû à une distribution inadaptée des données. Au delà de $n/3$ étapes, les calculs de la mise à jour ne sont distribués que sur au plus 4 processeurs.

pour le produit de matrice (figure 2.8), les processeurs des la première colonne et de la première ligne ne seraient actifs que pendant le premier tiers des étapes.

Les distributions d'un noyeau d'algèbre linéaire dense devant être adaptées à la fois au produit de matrice et à la décomposition LU, on applique deux permutations σ_h et σ_v de $[\![1, n]\!]$ aux distributions du produit de matrice. Ces permutations n'ont pas d'impact sur le coût ni sur le déroulement du produit de matrice. En revanche, ils ont un impact important sur le coût de la décomposition LU.

Dans le cas homogène, l'utilisation d'une distribution cyclique par bloc permet de répartir uniformément sur l'ensemble des processeurs les sous-matrices $A_{[k,n]\times[k,n]}$, et ce quelque soit k. ScaLAPACK [20] préconise donc l'utilisation de distributions cycliques par blocs dans les deux directions (voir figure 2.9(b)). En effet, une distribution cyclique par blocs dans une seule des directions (voir figure 2.9(a)) permet de bien équilibrer les calculs mais ne passe pas bien à l'échelle pour les même raisons que qu'une distribution mono-dimensionnelle simple ne passait pas bien à l'échelle pour le produit matriciel (ce type de distribution engendre trop de communications). Sur un réseau de stations homogènes, une distribution cyclique par blocs dans les deux directions conduit donc à un équilibrage de charge parfait à chaque étape de l'algorithme et à une quantité de communications minimale.



(a) Cyclique par blocs unidimensionnelle

(b) Cyclique par blocs bidimensionnelle

FIG. 2.9 – Distributions homogènes non contiguës adaptées aux factorisations LU pour une plate-forme composée de 6 processeurs

En pratique, on utilise bien évidemment une version par blocs de cet algorithme. C'est le cas notamment de la bibliothèque ScaLAPACK [20], qui utilise une version par blocs et pipelinée (dite *right-looking*) de cet algorithme.

2.3.2 Factorisation LU sur une ligne hétérogène de processeurs

Comme nous l'avons montré dans la section précédente, les schémas de distribution présentés dans la section 2.2.3 ne sont pas adaptées aux factorisations LU, et ce pour les mêmes raisons que celles du cas homogène. Dans cette section, nous rappelons l'existence d'un algorithme incrémental exposé dans [24] permettant d'obtenir une distribution monodimensionnelle parfaite de la matrice en assurant une répartition du calcul du complément optimale à chaque étape dans le cas d'une ligne de processeurs de vitesses différentes.

ALLOCATION_OPTIMALE(t_1,...,t_p, M)
1: $C = (c_1, \ldots, c_p) = (0, \ldots, 0)$
2: **Pour** $n = 1..M$ **:**
3: $\quad i \leftarrow \text{argmin}_{1 \leqslant j \leqslant p}(t_j(c_j + 1))$
4: $\quad Alloc(n) \leftarrow i$
5: $\quad c_i \leftarrow c_i + 1$
6: **Renvoyer**($Alloc$)

Algorithme 2.1: Algorithme calculant une allocation optimale de M tâches de même taille indépendantes sur p processeurs

Comme nous l'avons vu dans la section 2.3.1, la factorisation LU consiste en une

suite de factorisations (calcul du pivot) et de mises à jour (calcul du *complément de Schur*). Ce dernier constituant le terme dominant de la complexité de la factorisation, il convient de le répartir de façon optimale en fonction des vitesses relatives des processeurs. L'algorithme 2.1 (exposé dans [24]) nous permet d'obtenir une distribution parfaite de la matrice en nous assurant une répartition du calcul du complément optimale à chaque étape. En effet, le théorème suivant est démontré dans [24].

Théorème 2.1. *Étant donné les temps de cycle $t_1, \ldots, t_p$ de p processeurs, l'algorithme 2.1 calcule une allocation unidimensionnelle Alloc de M blocs de colonnes optimale pour tout n inférieur à M.*

Aussi surprenant que cela puisse paraître, la solution optimale pour n colonnes peut être obtenue en complétant une solution optimale à $n-1$ colonnes. Cet algorithme minimise donc $\max c_i t_i$ à chaque étape, ce qui signifie que l'équilibrage est optimal à chaque étape. La figure 2.10(a) illustre le fonctionnement de cet algorithme sur un exemple.

Si on distribue les colonnes dans l'ordre inverse de celui donné par l'algorithme incrémental précédemment exposé, on obtient donc une distribution optimale pour chaque étape de la factorisation LU (chaque calcul du complément est effectué sur une distribution optimale). La figure 2.10(b) compare l'efficacité de la distribution ainsi obtenue à celle d'une allocation cyclique. Cependant, les distributions monodimensionnelles passent mal à l'échelle. Après avoir trouvé l'équivalent des distributions cycliques par blocs monodimensionnelles pour les plates-formes hétérogènes, il est donc important d'arriver à trouver l'équivalent des distributions bidimensionnelles cycliques par blocs.

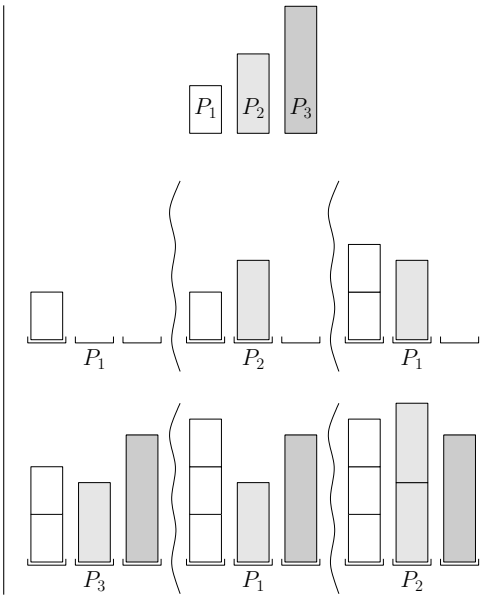
2.3.3 Distribution bidimensionnelle

Si on dispose de pq processeurs de vitesses différentes et que l'on souhaite effectuer une décomposition en utilisant une distribution bidimensionnelle, deux cas se présentent :

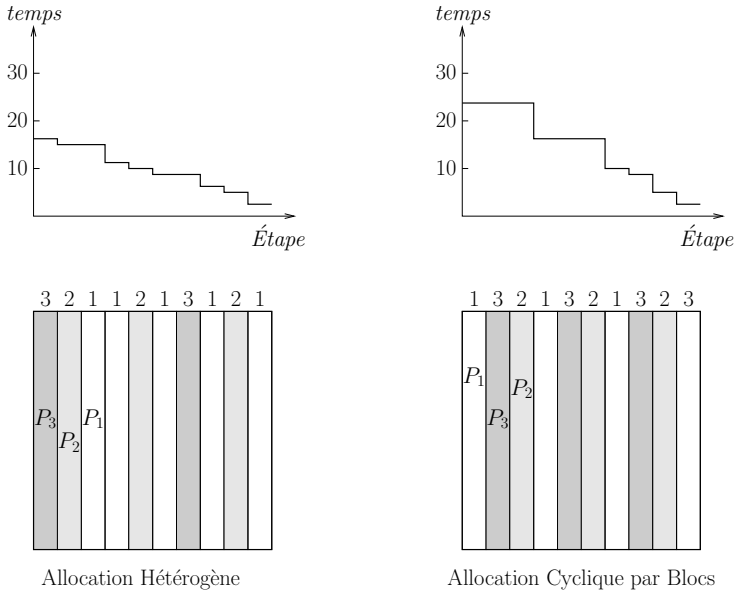
Grille 2D (cas simple) : supposons qu'il existe un arrangement des vitesses normalisées des processeurs en une matrice de rang 1 $r.^t c$ (on peut donc supposer $r_i t_{ij} c_j = 1$). Soit $Alloc = (\sigma_1, \sigma_2)$ une distribution d'une matrice $n \times n$ sur cette grille $p \times q$ (avec σ_1 une application de $[\![1, n]\!]$ dans $[\![1, p]\!]$ et σ_2 une application de $[\![1, n]\!]$ dans $[\![1, q]\!]$). Pour $t \in [\![1, n]\!]$, notons r'_i le nombre de lignes de type i dans la sous-matrice $A_{[t,n]\times[t,n]}$ (c'est-à-dire le nombre de valeurs l de $[\![t, n]\!]$ telles que $\sigma_1(l) = i$) et c'_j le nombre de colonnes de type j (c'est-à-dire le nombre de valeurs l de $[\![t, n]\!]$ telles que $\sigma_2(l) = j$). Le temps d'exécution de la $t^{\text{ème}}$ étape est donc $\max_{i,j} r'_i t_{ij} c'_j = \max_i \max_j r'_i t_{ij} c'_j = \max_i \max_j \frac{r'_i c'_j}{r_i c_j} = \max_i \frac{r'_i}{r_i} \max_j \frac{c'_j}{c_j}$. Ces deux quantités indépendantes étant minimisées par l'algorithme 2.1, l'application de cet algorithme dans chacune des dimensions fournit une distribution des blocs optimale pour chaque étape.

Grille 2D (cas non trivial) : s'il n'existe pas d'arrangement des vitesses en une matrice de rang 1 alors il n'existe pas forcément de distribution optimale à chaque étape. La figure 2.11 illustre ce problème. La distribution optimale pour une matrice carrée

Temps ce cycle	$t_1 = 3$	$t_2 = 5$	$t_3 = 8$		
Nombre de tâches	c_1	c_2	c_3	Coût	Processeur
0	0	0	0		1
1	1	0	0	3	2
2	1	1	0	2.5	1
3	2	1	0	2	3
4	2	1	1	2	1
5	3	1	1	1.8	2
6	3	2	1	1.67	1
7	4	2	1	1.71	1
8	5	2	1	1.87	2
9	5	3	1	1.67	3
10	5	3	2	1.6	



(a) Exécution de l'algorithme 2.1 pour un ensemble de 3 processeurs de temps de cycle 3, 5 et 8. À l'étape k, on alloue le $k^{\text{ème}}$ bloc au processeur minimisant le temps de traitement de k blocs tout en utilisant l'allocation précédente pour les $k-1$ premiers blocs. On alloue donc le premier bloc au processeur P_1 (c'est le plus rapide), puis le second au processeur P_2 (c'est le processeur inoccupé le plus rapide). En revanche, le troisième bloc n'est pas alloué au processeur P_3 car le temps de traitement serait alors de 8. Il est plus rentable d'allouer le troisième bloc au processeur P_1 puisque le temps de traitement est alors de 6.



(b) Comparaison de la distribution construite par l'algorithme 2.1 avec la distribution cyclique par blocs pour un ensemble de 3 processeurs de temps de cycle 3, 5 et 8 et des compléments de taille inférieure à 10. Le temps de calcul d'une factorisation LU est donc donné par la surface située sous la courbe correspondant à la distribution des données. L'allocation hétérogène est clairement meilleure que la distribution cyclique par blocs.

FIG. 2.10 – Utilisation de l'algorithme 2.1 pour créer des distributions hétérogènes adaptées aux factorisations LU

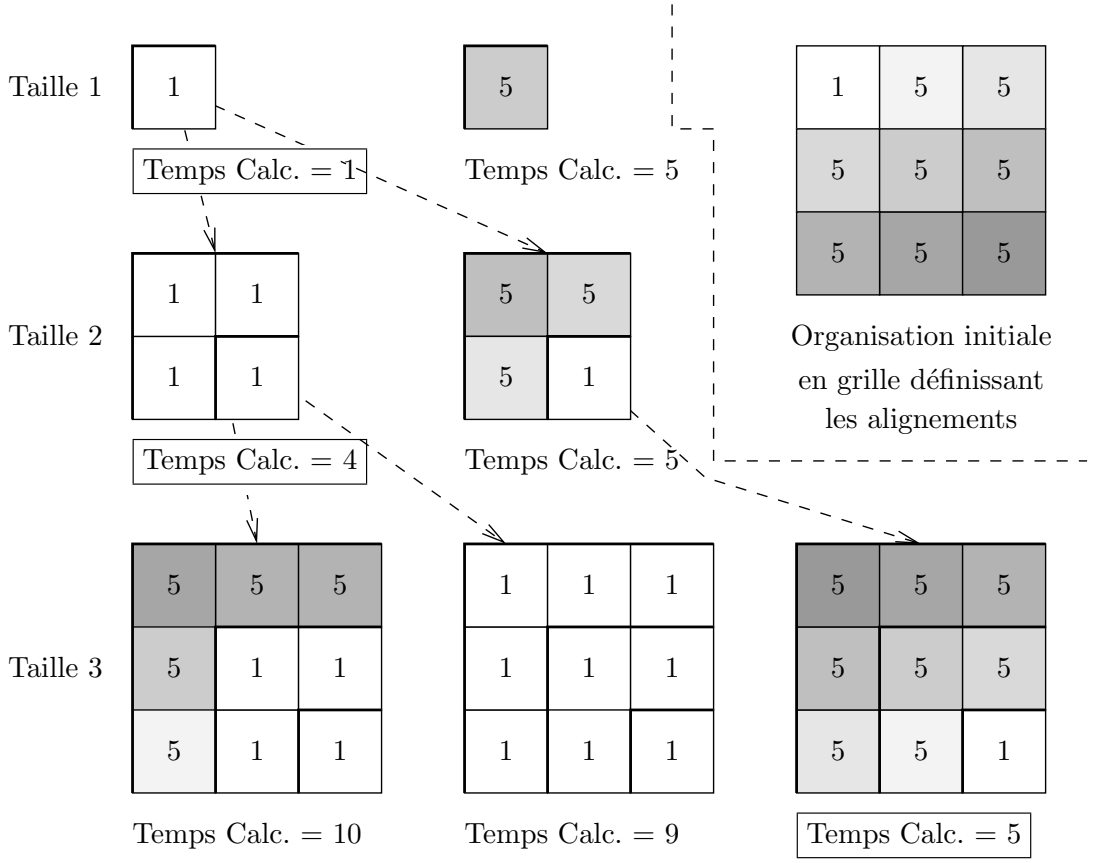


FIG. 2.11 – On part d'une grille 3×3 constituée de 8 processeurs de temps de cycles 5 et d'un processeur de temps de cycle 1. La disposition des différents processeurs sur la grille est représentée en haut à droite. Il n'existe pas nécessairement de solution optimale à chaque étape : sur cet exemple, la solution optimale pour un complément de taille 3 ne peut pas être obtenue à partir d'une solution optimale pour un complément de taille 2

composée de 9 blocs ne peut être obtenue à partir d'une distribution optimale pour une matrice carrée composée de 4 blocs.

2.3.4 Distributions en colonnes

Le problème d'équilibrage précédent est dû à la rigidité des distributions en grille et à un mauvais équilibrage dès le départ. Pour pallier ce problème, nous proposons de «virtualiser» les distributions en colonnes (qui sont parfaitement équilibrées), c'est-à-dire de les considérer comme des grilles hétérogènes en prolongeant les frontières locales des différentes colonnes de processeurs en frontières globales (cf. figure 2.12 et 2.13). Sur cet exemple, la distribution en colonne originale, parfaitement équilibrée pour un ensemble de 10 processeurs, est virtuellement transformée en une grille 6×3. On a donc construit une grille virtuelle hétérogène et chaque processeur est responsable de plusieurs zones

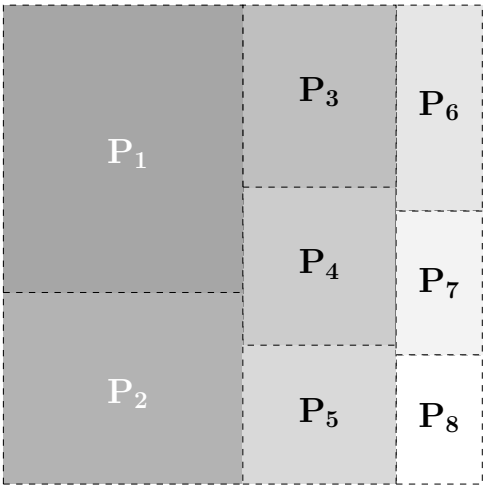


FIG. 2.12 – Distribution en colonne originale et adaptée au produit matriciel

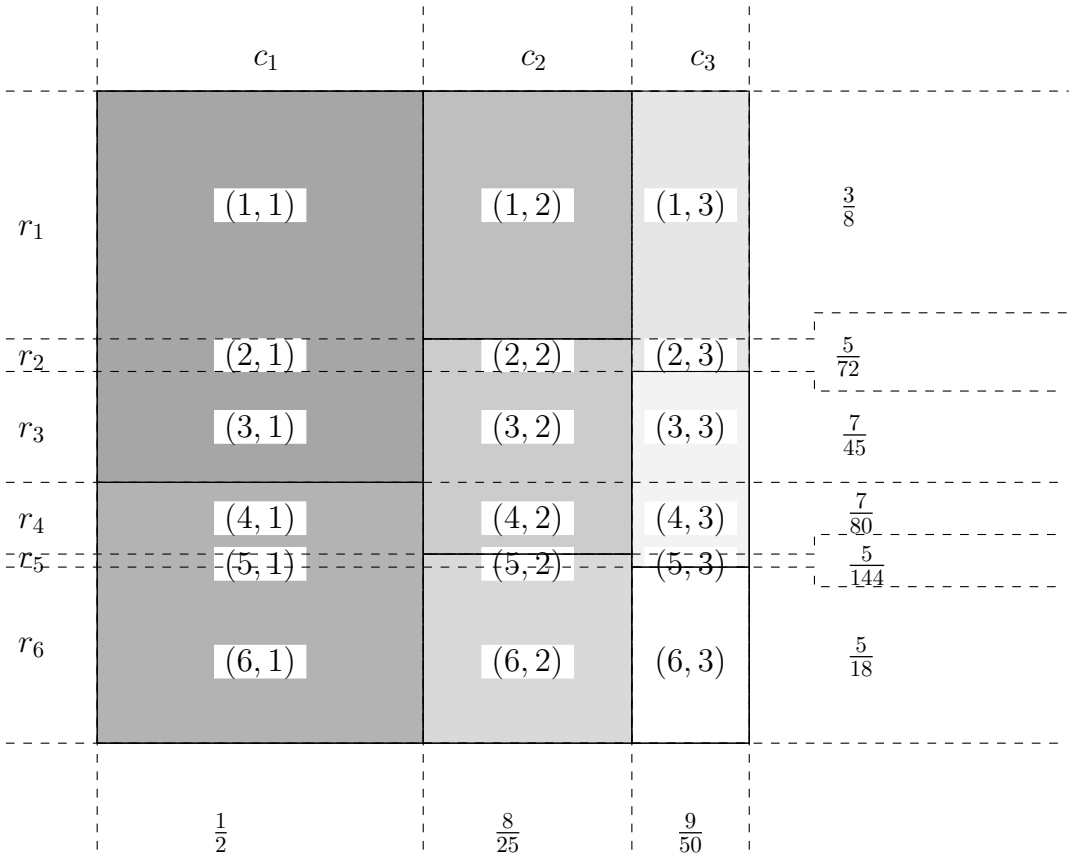


FIG. 2.13 – Virtualisation d'une distribution en colonnes : on prolonge les frontières horizontales de chaque colonne afin de créer une grille hétérogène virtuelle

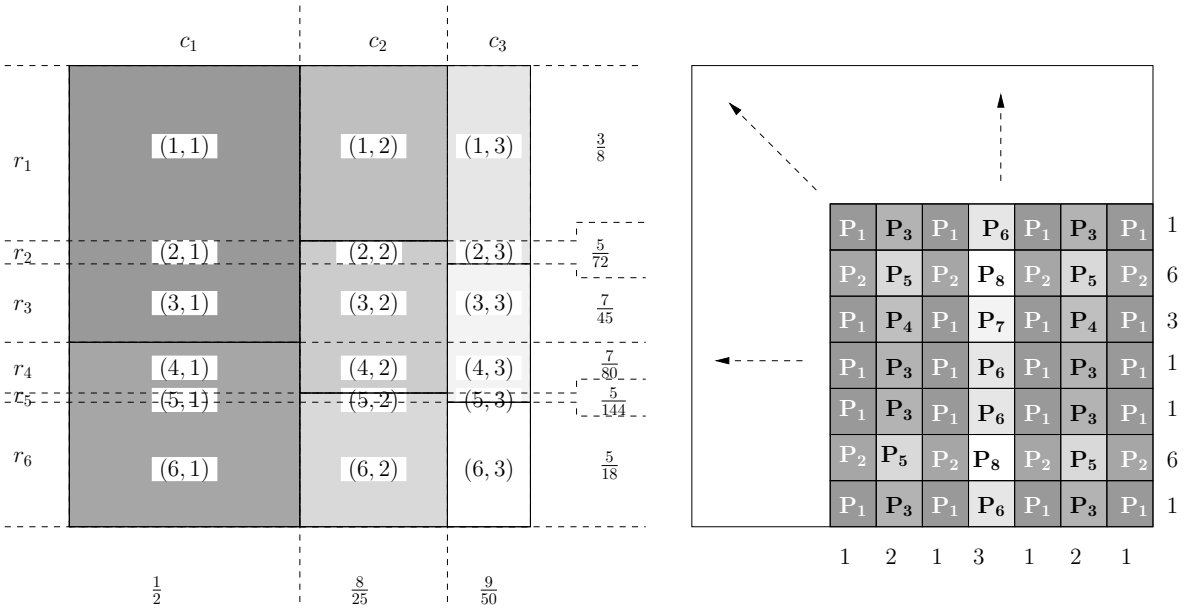


FIG. 2.14 – Allocation des données pour la factorisation LU en utilisant l'algorithme incrémental indépendamment dans chacune des deux dimensions

contiguës de cette grille.

Notons r_i et c_j la hauteur et la largeur normalisées des décompositions en rectangles sur la grille (virtuelle) hétérogène (voir figure 2.13). Nous allons utiliser l'algorithme introduit dans la section 2.3.2, une fois dans chacune des deux dimensions, puis mélanger les résultats.

Considérons l'exemple représenté sur la figure 2.14. La distribution initiale, parfaitement équilibrée pour un ensemble hétérogène de 7 processeurs, est représentée sur la figure 2.12. Cette distribution est transformée en une grille virtuelle hétérogène 6×3 représentée par des traits pointillés sur la figure 2.13). Les hauteurs et largeurs de cette nouvelle grille sont les suivantes : $r = (\frac{3}{8}, \frac{5}{72}, \frac{7}{45}, \frac{7}{80}, \frac{7}{144}, \frac{5}{18})$ et $c = (\frac{1}{2}, \frac{8}{25}, \frac{9}{50})$. En utilisant l'algorithme 2.1 selon la première dimension, on obtient la séquence (r_1, r_6, r_1, r_3, r_6, r_1, r_1, r_6, r_4, r_3, r_1, r_6, r_2, ...). En l'utilisant selon la seconde dimension, on obtient la séquence (c_1, c_2, c_1, c_3, c_1, c_2, c_1, c_2, c_1, c_3, c_1, c_2, c_1, ...). La distribution résultante est représentée sur la partie droite de la figure 2.14.

Cette solution n'est pas optimale à chaque étape mais elle est cependant asymptotiquement optimale.

Théorème 2.2. *Virtualiser des distributions parfaitement équilibrées et les traiter comme des grilles bidimensionnelles conduit à des distributions asymptotiquement optimales pour LU.*

Démonstration. On va d'abord considérer le cas des distributions en colonnes, le cas des

distributions quelconques se traitant de façon similaire.

Notons $u_{i,j}$ le plus petit numéro de zone dont est responsable le processeur P_{ij} (le $i^{\text{ème}}$ processeur de la $j^{\text{ème}}$ colonne) et $l_{i,j}$ le plus grand. Ainsi, P_{ij} est responsable des zones de hauteurs normalisées $r_{u_{i,j}}, r_{u_{i,j}+1}, \ldots, r_{l_{i,j}}$ et de largeur normalisée c_j. Cette distribution en colonnes étant parfaitement équilibrée, on pourra supposer :

$$\left(\sum_{k=u_{i,j}}^{l_{i,j}} r_k \right) c_j = h_{ij} w_{ij} = s_{ij} = \frac{1}{t_{ij}}. \tag{2.6}$$

Ainsi, le temps d'exécution de l'étape t est donc supérieur à :

$$T_{compl}^{min}(t) = \alpha t^2 \max_{i,j} \left(\sum_{k=u_{i,j}}^{l_{i,j}} r_k t_{ij} c_j \right) = \alpha t^2, \tag{2.7}$$

ce qui correspond au cas où l'équilibrage de l'étape t est parfait.

Pour $t \in [\![1, n]\!]$, on notera $r_k^{(t)}$ la fraction de lignes appartenant à la $k^{\text{ème}}$ zone horizontale dans la sous-matrice $A_{[t,n]\times[t,n]}$ et $c_j^{(t)}$ celle de colonnes de type j. Par construction, on a donc :

$$r_k^{(t)} t = r_k t + \varepsilon_k^{(t)} \text{ et } c_j^{(t)} t = c_j t + \eta_j^{(t)} \text{ avec } \varepsilon_k^{(t)} \text{ et } \eta_j^{(t)} \in [-1, 1]. \tag{2.8}$$

Le temps de calcul total pour l'étape t est donc :

$$\begin{aligned}
T_{compl}(t) &= \alpha t^2 \max_{i,j} \left(\sum_{k=u_{i,j}}^{l_{i,j}} r_k^{(t)} t_{ij} c_j^{(t)} \right) \\
&= \alpha t^2 \max_{i,j} \left(\frac{\sum_{k=u_{i,j}}^{l_{i,j}} r_k^{(t)}}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \cdot \frac{c_j^{(t)}}{c_j} \right) && \text{(en utilisant (2.6))} \\
&= \alpha t^2 \max_{i,j} \left(\frac{\sum_{k=u_{i,j}}^{l_{i,j}} r_k + \varepsilon_k^{(t)}/t}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \cdot \frac{c_j + \eta_j^{(t)}/t}{c_j} \right) && \text{(en utilisant (2.8))} \\
&= \alpha t^2 \max_{i,j} \left(\left(1 + \frac{\sum_{k=u_{i,j}}^{l_{i,j}} \varepsilon_k^{(t)}/t}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \right) \cdot \left(1 + \frac{\eta_j^{(t)}/t}{c_j} \right) \right)
\end{aligned} \tag{2.9}$$

On a donc,

$$\begin{aligned}
\sum_{t=1}^{n} T_{compl}(t) &= \sum_{t=1}^{n} \alpha t^2 \max_{i,j} \left(\left(1 + \frac{\sum_{k=u_{i,j}}^{l_{i,j}} \varepsilon_k^{(t)}/t}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \right) \left(1 + \frac{\eta_j^{(t)}/t}{c_j} \right) \right) \\
&\leqslant \sum_{t=1}^{n} \alpha t^2 \left(1 + \frac{K_1}{t} \right) \left(1 + \frac{K_2}{t} \right) \qquad \text{avec } K_1 \text{ et } K_2 \text{ deux constantes indépendantes de } n \text{ et de } t \\
&\leqslant \alpha \left(\sum_{t=1}^{n} t^2 + \sum_{t=1}^{n} (K_1 + K_2)t + \sum_{t=1}^{n} K_1 K_2 \right) \text{ , d'où en développant} \\
\sum_{t=1}^{n} T_{compl}(t) &\leqslant \left(\sum_{t=1}^{n} \alpha t^2 \right) \left(1 + O\left(\frac{1}{n} \right) \right) = \left(\sum_{t=1}^{n} T_{compl}^{min}(t) \right) \left(1 + O\left(\frac{1}{n} \right) \right) \qquad (2.10)
\end{aligned}$$

Ainsi, la virtualisation conduit à une solution asymptotiquement optimale pour l'équilibrage des phases de mises à jour durant l'exécution de l'algorithme de factorisation LU.

Le coût des communications est donné par :

$$\begin{aligned}
T_{comm}(t) &= \beta \sum_{i,j} t \left(c_j^{(t)} + \sum_{k=u_{i,j}}^{l_{i,j}} r_k^{(t)} \right) \\
&= \beta \sum_{i,j} \left(t \left(c_j + \sum_{k=u_{i,j}}^{l_{i,j}} r_k \right) + \left(\eta_j^{(t)} + \sum_{k=u_{i,j}}^{l_{i,j}} \varepsilon_k^{(t)} \right) \right) \text{ (en utilisant (2.8))} \\
&= \beta t \left(\sum_r h_r + w_r \right) + O(1)
\end{aligned}$$

On a donc :

$$\begin{aligned}
\sum_{t=1}^{n} T_{comm}(t) &= \sum_{t=1}^{n} \beta t \left(\sum_r h_r + w_r \right) + O(1) \\
&\leqslant \left(\sum_{t=1}^{n} \beta t \right) \left(\sum_r h_r + w_r \right) \left(1 + O\left(\frac{1}{n} \right) \right) \qquad (2.11)
\end{aligned}$$

On pourra remarquer que le coût total des communications lors des mises à jour est approximativement deux fois plus petit pour la factorisation LU que pour le produit matriciel, ce qui est également le cas pour une grille homogène.

L'utilisation dans cette démonstration avec les distributions en colonnes permet de ne pas trop alourdir des notations déjà bien chargées. Le résultat s'étend sans difficulté aux distributions plus générales (récursives ou quelconques) du moment qu'elles sont parfaitement équilibrées pour le produit matriciel. ■

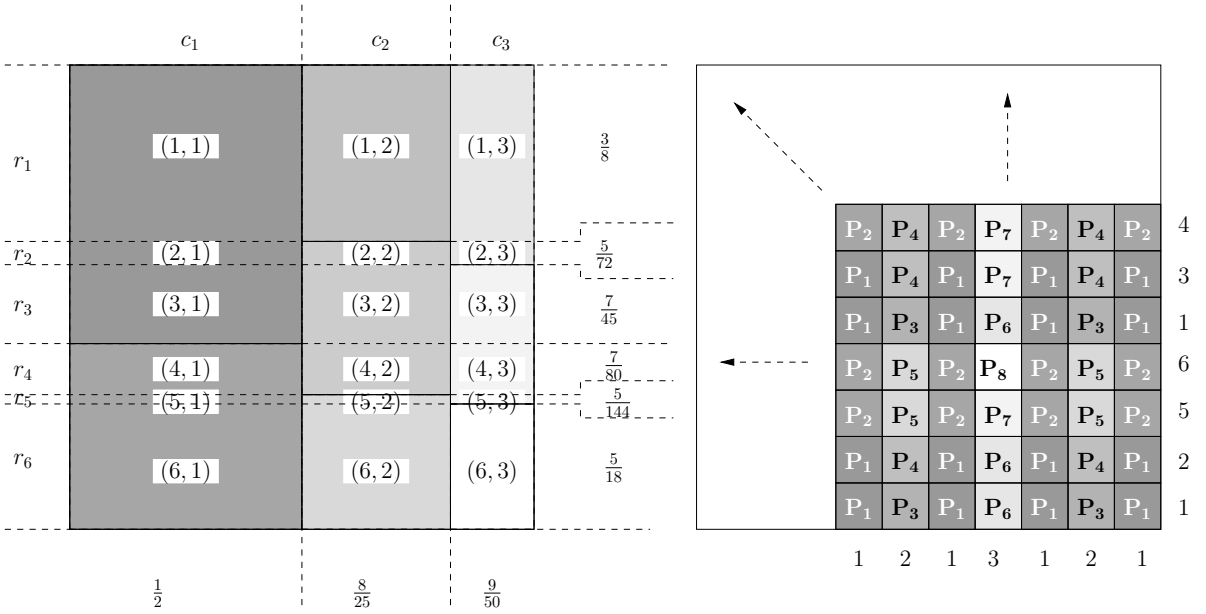


FIG. 2.15 – Principe de l'heuristique gloutonne bidimensionnelle

2.3.5 Simulations

Dans cette section, nous comparons l'efficacité des différents schémas de distributions pour la factorisation LU à l'aide de simulations. Nous nous intéressons au rapport entre le temps de calcul qu'induirait une distribution dont chaque étape de mise à jour serait parfaitement équilibrée et celui qu'induiraient les diverses distributions suivantes :

Cyclique par blocs C'est la distribution monodimensionnelle par blocs classique (voir figure 2.9(a)).

Distribution en colonnes hétérogène C'est la version initiale (non mélangée) des distributions en colonnes adaptées au produit matriciel (section 2.2.3). La charge est bien équilibrée uniquement au début de la factorisation.

Heuristique gloutonne bidimensionnelle Cette heuristique est fondée sur la distribution en colonnes hétérogène précédente. Une allocation est construite itérativement en choisissant de façon gloutonne les zones horizontales et verticales minimisant le temps de calcul. La figure 2.15 montre la distribution construite par cette heuristique. À l'étape 1, la zone (c'est-à-dire le couple numéro de colonne/numéro de ligne) choisie est celle contenant le processeur le plus rapide et donc la zone choisie est $(1, 1)$. À l'étape 2, c'est la zone $(2, 2)$ qui donne les meilleurs résultats (c'est-à-dire qu'aucun couple numéro de colonne/numéro de ligne ne donne un meilleur équilibrage de charge pour la factorisation du complément de taille 2) ; à l'étape 3, le couple $(5, 1)$ est sélectionné et ainsi de suite : $(6, 3)$, $(1, 1)$, $(3, 2)$, $(4, 1)$ ...

Algorithme incrémental dans chacune des deux dimensions C'est l'algorithme décrit en section 2.3.4.

Les figures 2.16(a) et 2.16(b) comparent le ratio entre le temps de calcul total pour les quatre distributions précédentes et la borne absolue qui correspondrait à un équilibrage de charge parfait et qui ne peut être atteint dans le cas général (voir figure 2.11). Dans les deux cas, c'est l'utilisation de l'algorithme incrémental dans chacune des deux dimensions qui donne les meilleurs résultats.

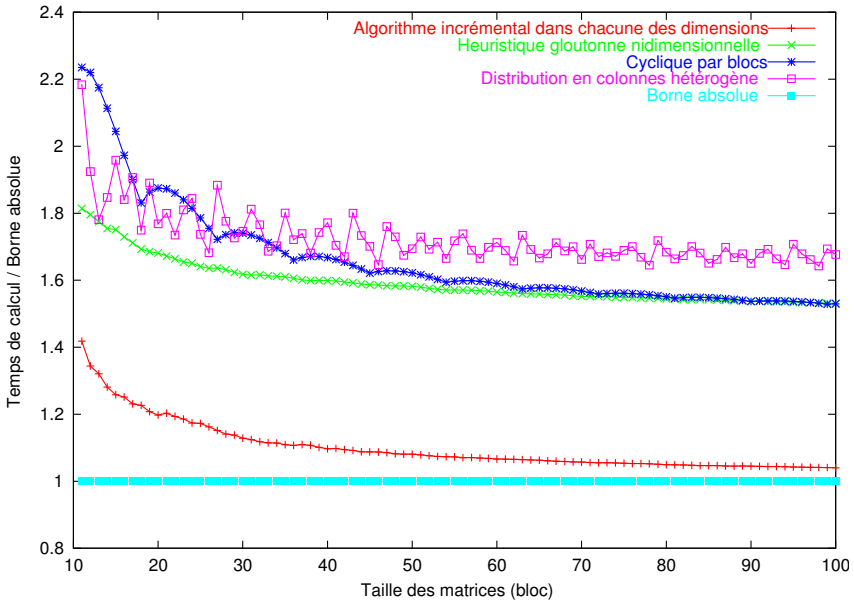
2.4 Redistributions

Nous avons donc montré comment réaliser des distributions de données adaptées aux algorithmes classiques d'algèbre linéaire en tenant compte de l'hétérogénéité des vitesses des machines et des caractéristiques du réseau d'interconnexion. Ces solutions ne sont cependant efficaces que si la charge des processeurs ne varie pas au cours de l'exécution de ces algorithmes. La garantie de la stabilité d'environnements tels qu'un réseau de stations (partagé par un grand nombre d'utilisateurs) ou des machines parallèles (dont les politiques d'ordonnancement interne peuvent varier et n'assurent pas forcément le monopole de la machine à une application) reliées par des liaisons longues distances est difficile voire souvent impossible à garantir.

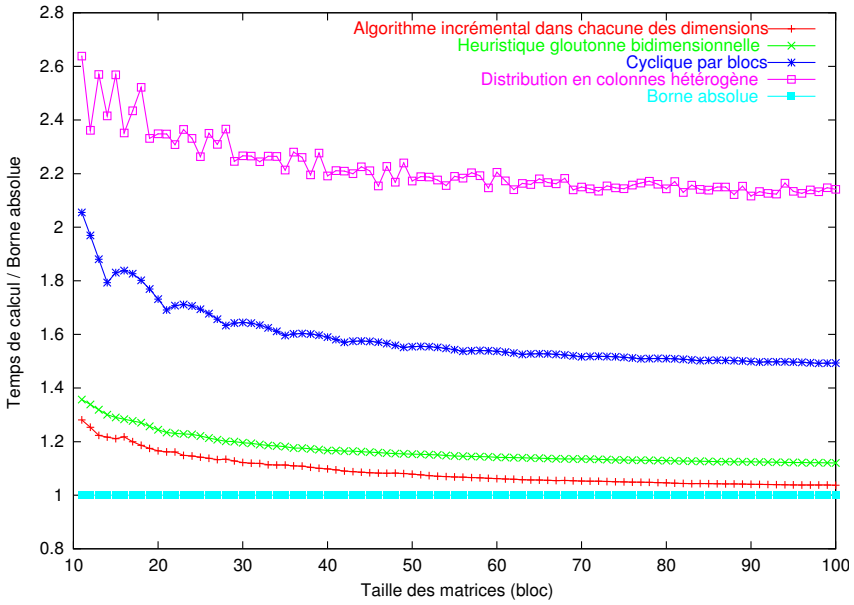
La découpe de ces algorithmes en tâches et la mise en place d'un modèle maître/esclave ne peut pas être efficace en raison des dépendances de données entre les différentes tâches (voir [10] pour un exemple plus complet). Une redistribution complète des données pour passer d'une distribution optimale à une autre n'est pas non plus envisageable en raison de la quantité de communications que cela engendrerait (elle peut être très supérieure à la quantité totale de communications générées par l'algorithme). Nous allons donc montrer comment des redistributions légères, basées sur les distributions précédemment exposées, peuvent être effectuées entre chaque étape de calcul.

2.4.1 Règles du jeu

Les répartitions en colonnes étant relativement proches de l'optimum, c'est sur ces dernières que nous allons travailler. Ce paragraphe présente donc des opérations élémentaires de rééquilibrage permettant de transformer une distribution anciennement acceptable en une distribution plus adaptée aux performances courantes des processeurs. Connaissant les nouvelles vitesses relatives des machines, on peut déterminer le nombre de blocs que chacune d'elles devrait avoir pour être dans une situation équilibrée. Cela permet donc de calculer leur déséquilibre $D = (\delta_1, \ldots, \delta_p)$, où δ_i représente le nombre de blocs que P_i a en trop. D vérifie donc $\sum_{i=1}^{p} \delta_i = 0$ et $\sum_{i=1}^{p} |\delta_i|$ représente la mesure du déséquilibre. Les opérations que nous allons exposer ne modifient pas $\sum_{i=1}^{p} \delta_i$ mais visent à diminuer $\sum_{i=1}^{p} |\delta_i|$, c'est à dire à obtenir un meilleur équilibrage de charge.

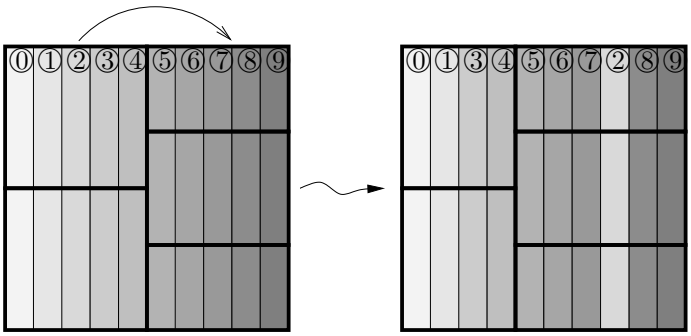


(a) Comparaison de diverses allocations des données pour des processeurs de temps de cycles 1, 5, 5, 5, 5, 5, 5, 5 et 5 (cas pathologique évoqué figure 2.11). L'heuristique garantie (l'algorithme incrémental décrit en section 2.3.4) tend rapidement vers la borne absolue et est bien meilleure que les autres distributions.

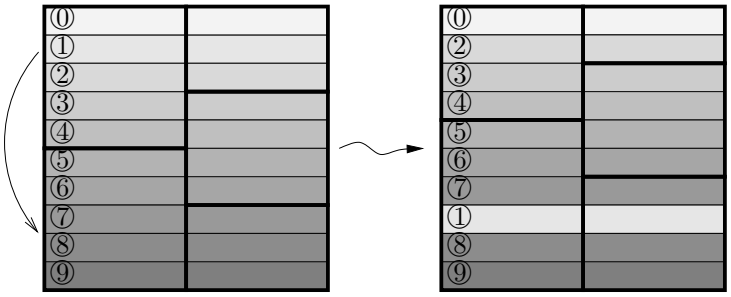


(b) Comparaison de diverses allocations des données pour des processeurs de vitesses relatives 0.1, 0.1, 0.12, 0.15, 0.15, 0.18 et 0.2 (exemple utilisé figure 2.13)

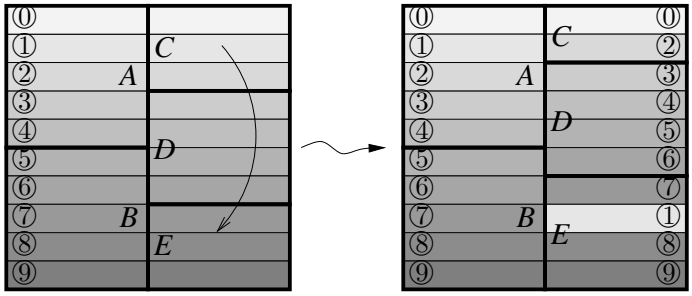
FIG. 2.16 – Comparaison du ratio entre le temps de calcul total pour les quatre distributions précédentes et la borne absolue qui correspondrait à un équilibrage de charge parfait



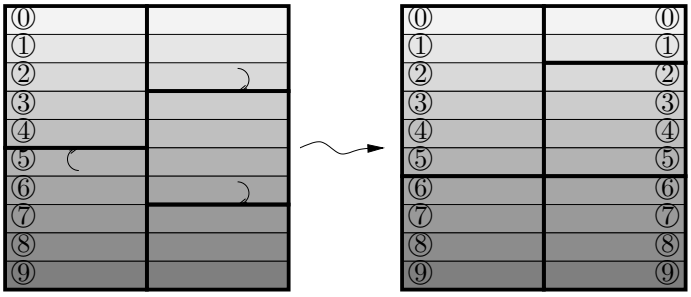
(a) Migration globale d'une colonne : aucun impact sur la diffusion des lignes ou des colonnes



(b) Migration globale d'une ligne : aucun impact sur la diffusion des colonnes ou des lignes



(c) Migration locale d'une ligne : impact désastreux sur la diffusion des colonnes. Le processeur A est voisin de C, D et E et il envoie de petits messages à ce dernier



(d) Pousse-pousse : communication des frontières sans rupture des alignements ; aucun impact sur la diffusion des colonnes ou des lignes

FIG. 2.17 – Différentes opérations élémentaires de rééquilibrage

2.4.1.1 Migrations de colonnes

La migration globale d'une colonne est illustrée par la figure 2.17(a) : aucun alignement des données n'est brisé. Cette opération équivaut à effectuer une permutation de la matrice puis à modifier les surfaces dont sont responsables les différents processeurs. Il n'y a donc aucun impact sur les performances de nos algorithmes. Cette opération nous permet donc de rééquilibrer le déséquilibre inter-colonne d'une distribution.

2.4.1.2 Migrations de lignes

La migration globale d'une ligne est illustrée figure 2.17(b) : cette fois-ci encore, aucun alignement des données n'est brisé. Comme précédemment, cette opération est équivalente à effectuer une permutation de la matrice et à modifier les surfaces dont sont responsables les différents processeurs. Il n'y a donc toujours aucun impact sur les performances de nos algorithmes.

Les deux opérations précédentes ne nous permettent cependant pas de rééquilibrer complètement une distribution. En effet, en partant d'une distribution ayant un déséquilibre de type

-1	1
1	-1

on ne peut aboutir qu'à

-2	0
0	-2

ou

0	2
2	0

en effectuant des migrations globales de lignes.

Il faudrait pouvoir migrer un fragment de ligne localement à une colonne. C'est l'opération qui est présentée figure 2.17(c). On peut cependant remarquer que cette opération rompt les alignements de données et a donc un effet désastreux sur le mouvement des colonnes. En effet, on peut remarquer sur la figure 2.17(c) qu'une fois l'opération de migration locale effectuée, le processeur situé en haut à droite doit communiquer des données à tous les processeurs de la seconde colonne. Les envois de données sont donc fragmentés et après quelques échanges locaux, chaque processeur est susceptible d'avoir à envoyer des données à tous les autres, ce qui n'est pas acceptable.

2.4.1.3 Pousse-pousse

Il faut donc contraindre quelque peu ces migrations locales de façon à conserver les alignements. Si les processeurs ne peuvent donner que des fragments de lignes situés sur leurs frontières, les alignements ne sont pas brisés. Cependant, les communications nécessaires au rééquilibrage peuvent être plus importantes. En effet, on peut voir sur la figure 2.17(d) que dans la seconde colonne deux migrations sont faites alors qu'une seule était nécessaire dans le cas de migrations locales non restreintes. C'est la raison du nom de cette opération (pousse-pousse) : les déséquilibres doivent être propagés de proche en proche jusqu'à leur disparition complète.

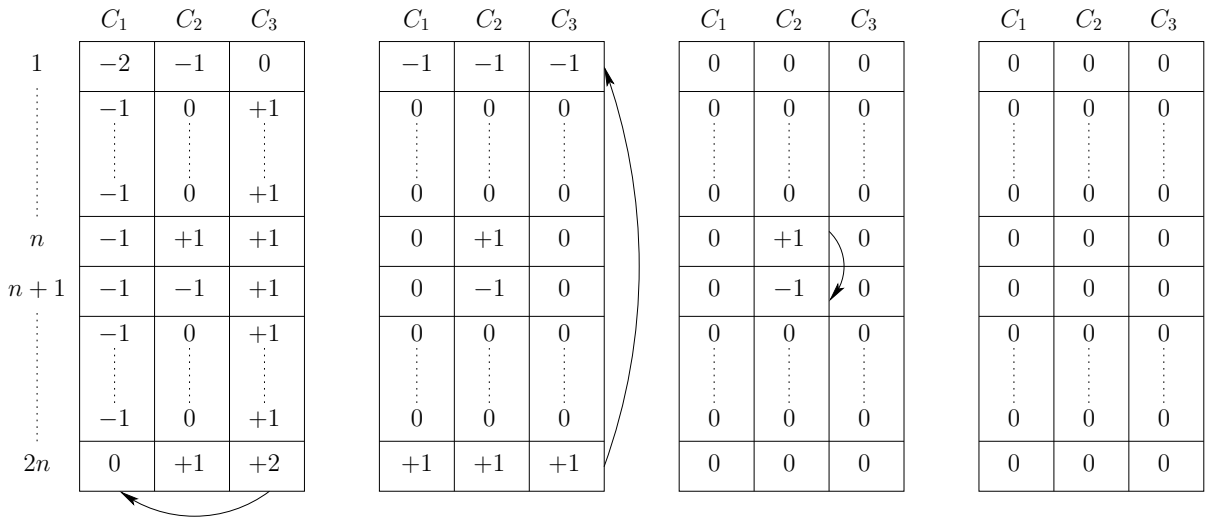


FIG. 2.18 – Exemple de combinaisons des opérations précédentes

2.4.1.4 Plan d'attaque

Les seules opérations que nous allons utiliser vont donc être la migration globale de colonnes, la migration globale de lignes et l'opération de pousse-pousse. Ces opérations étant commutatives, si le coût des communications point-à-point est homogène et si le coût d'une migration de ligne est constant, on rééquilibre d'abord dans chaque colonne à l'aide de migrations de colonnes, puis on effectue des migrations de lignes pour minimiser le coût des opérations de pousse-pousse.

Un exemple de combinaison des opérations précédentes est donné figure 2.18. Toutes les colonnes sont de largeur 1 et chaque ligne est de hauteur 1. La migration d'une colonne de C_3 vers C_1 coûte $2n$ et permet de rééquilibrer la distribution par colonne. Effectuer directement une opération de pousse-pousse dans les trois colonnes a un coût de $(2n-1)+2(n-1)+(2n-1) = 6n-4$ alors qu'un simple mouvement de ligne suivit d'un pousse-pousse dans C_2 rééquilibre la distribution pour un coût de $3+1=4$.

Cet exemple nous montre la nécessité de chacune de ces opérations dans le mécanisme de rééquilibrage.

2.4.2 Un algorithme glouton efficace

Les migrations de colonnes sont inévitables mais l'exemple précédent nous a montré que la migration des lignes avait une influence importante sur le coût des opérations de pousse-pousse. L'exemple de la figure 2.19 essaie de montrer que le meilleur choix des migrations de lignes à effectuer n'est pas forcément celui qui paraît le plus évident. Cette distribution est déjà équilibrée par colonne. Nous ne pouvons donc effectuer que des migrations de lignes ou des opérations de pousse-pousse. Les choix pour le rééquilibrage

	C_1	C_2	C_3
1	$+$	$+$	$+$
2	0	0	0
3	$-$	$+$	$-$
	0 ⋮ 0	0 ⋮ 0	0 ⋮ 0
$n-2$	$+$	$-$	$+$
$n-1$	0	0	0
n	$-$	$-$	$-$

FIG. 2.19 – Un choix pas aussi évident qu'il n'en a l'air

restent quand même nombreux :

- rééquilibrer en n'utilisant que des opérations de pousse-pousse $\big(PP(C_1); PP(C_2); PP(C_3)\big)$ coûte $4 + \big(4 + 2(n-4)\big) + 4 = 2n + 4$;
- la ligne 1 et la ligne n étant complémentaires, il peut paraître naturel de migrer la ligne 1 vers la ligne n puis finir de rééquilibrer avec des opérations de pousse-pousse $\big(L(1,n); PP(C_1); PP(C_2); PP(C_3)\big)$. Ces opérations engendrent des communications égales à $3 + 3(n-4) = 3n - 9$, ce qui est pire qu'avant ;
- cette situation s'améliore en migrant la ligne 1 vers la ligne n, puis la ligne $n-2$ vers la ligne 3 et en finissant le rééquilibrage avec les opérations de pousse-pousse dans la colonne 2 $\big(L(1,n); L(n-2,3); PP(C_2)\big)$. Ce rééquilibrage coûte alors $6 + 2(n-4) = 2n - 2$ qui est optimal.

Cet exemple pourrait être étendu de façon à ce que m migrations non rentables de lignes soient nécessaires avant d'atteindre un rééquilibrage peu coûteux et donne donc l'impression qu'un algorithme glouton a peu de chances d'arriver à trouver une solution optimale.

Cependant, un rééquilibrage à l'aide des opérations $\big(L(1,3); PP(C_1); PP(C_2); PP(C_3)\big)$ coûte $3+2+\big(2+2(n-4)\big)+2 = 2n+1$, ce qui est meilleur que $\big(PP(C_1); PP(C_2); PP(C_3)\big)$. Enfin, rééquilibrer la distribution en effectuant $\big(L(1,3); L(n-2,n); PP(C_2)\big)$ coûte $6 + 2(n-4) = 2n - 2$, ce qui est optimal. Il existait donc un choix glouton des migrations de lignes conduisant à une solution optimale.

2.4.2.1 Minimisation du déséquilibre inter-colonne

Les migrations de lignes et les opérations de pousse-pousse ne diminuant le déséquilibre que dans chacune des colonnes et pas d'une colonne à l'autre, il est indispensable

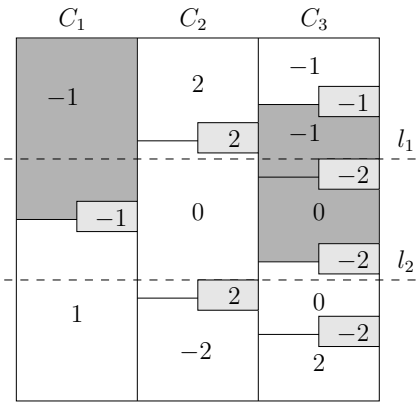


FIG. 2.20 – Représentation d'un déséquilibre

d'effectuer en premier des migrations de colonnes de façon à minimiser le déséquilibre inter-colonne. Si on suppose que le coût de communication d'un processeur à l'autre est constant, le coût de migration d'une colonne est alors constant et est proportionnel à la hauteur H de la matrice.

Si notre distribution est constituée de n_{col} colonnes, notons $d_1, \ldots, d_{n_{col}}$ le déséquilibre de chaque colonne (exprimé en nombre de colonnes). Le déséquilibre inter-colonne est alors égal à $\sum_{k=1}^{n_{col}} |d_k|$ et on a $\sum_{k=1}^{n_{col}} d_k = 0$. Au moins $\frac{1}{2}\sum_{k=1}^{n_{col}} |d_k|$ colonnes doivent donc être déplacées pour annuler le déséquilibre inter-colonne.

Notons k_+ la colonne ayant le plus gros excédent de colonne et k_- la colonne ayant le plus gros déficit de colonnes. Alors, en migrant $|d_{k_+} + d_{k_-}|$ colonnes de k_+ vers k_-, on diminue le déséquilibre inter-colonne de $2|d_{k_+} + d_{k_-}|$ pour un coût égal à $|d_{k_+} + d_{k_-}|H$. En itérant le procédé, on annule donc le déséquilibre inter-colonne pour un coût égal à $\frac{1}{2}\sum_{k=1}^{n_{col}} |d_k|H$.

Cet algorithme est optimal puisque le nombre de colonnes excédentaires sur l'ensemble de la matrice, et qui devait donc être déplacé, était égal à $\frac{1}{2}\sum_{k=1}^{n_{col}} |d_k|$. On peut donc se ramener au cas où le déséquilibre de chaque colonne est nul.

2.4.2.2 Modélisation des distributions

Étant donnée une répartition à n_{col} colonnes, on notera n_i le nombre de rectangles dans la colonne i et w_i sa largeur. Pour $i \in [\![1, n_{col}]\!]$ et $j \in [\![1, n_i]\!]$ on définira le $j^{\text{ème}}$ rectangle de la colonne i par son ordonnée $y_{i,j}$, sa hauteur $h_{i,j}$ et son déséquilibre $\delta_{i,j}$ (exprimé en nombre de lignes de largeur w_i).

On a vu en section 2.4.2.1 que l'on pouvait se ramener au cas où $\sum_{j=1}^{n_i} \delta_{i,j} = 0$ (en effectuant d'abord les migrations de colonnes) pour tout i.

Alors, Si on note $b_{i,j} = \sum_{k=1}^{j} \delta_{i,k}$ pour $i \in [\![1, n_{col}]\!]$ et $j \in [\![1, n_i - 1]\!]$, $b_{i,j}$ représente le nombre de lignes que les j premiers processeurs de la colonne i ont en trop. Lors d'une opération de pousse-pousse sur la colonne i, le premier processeur a donc $|b_{i,1}w_i|$ blocs à

échanger avec le second processeur. Une fois ces blocs transférés, le second processeur a $\delta_{i,1} + \delta_{i,2}$ lignes en trop et doit donc échanger $|(\delta_{i,1} + \delta_{i,2})w_i| = |b_{i,2}w_i|$ avec le troisième processeur et ainsi de suite. Le coût d'un rééquilibrage complet par pousse-pousse vaut donc :

$$PP(D) = \sum_{i=1}^{n_{col}} \sum_{j=1}^{n_i} w_i |b_{i,j}| \tag{2.12}$$

Définissons maintenant l'ensemble des processeurs dont une opération de rééquilibrage par pousse-pousse est concernée par une migration de ligne. Pour $l_1 < l_2$ on définit $\llbracket\, l_2, l_1 \,\rrbracket = \llbracket\, l_1, l_2 \,\rrbracket$ par

$$\left\{ b_t \mid t \in \llbracket\, l_1, l_2 \,\rrbracket \right\} = \left\{ b_{i,j} \mid i \in \llbracket\, 1, n_{col} \,\rrbracket, l_1 \leqslant y_{i,j} + h_{i,j} < l_2 \right\} \tag{2.13}$$

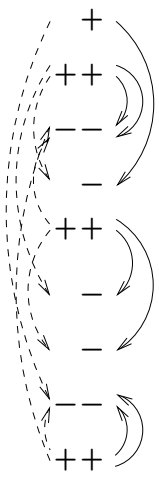
Les zones gris-foncé de la figure 2.20 représentent $\llbracket\, l_1, l_2 \,\rrbracket$. On notera que pour $a < b < c$ on a $\llbracket\, a, c \,\rrbracket = \llbracket\, a, b \,\rrbracket \cup \llbracket\, b, c \,\rrbracket$.

La migration de la ligne l_1 vers la ligne l_2 modifie les valeurs des $b_{i,j}$ (et donc le coût du pousse-pousse) de la façon suivante :

- Si $l_1 < l_2$: pour $t \in \llbracket\, l_1, l_2 \,\rrbracket$: $b_t \leftarrow b_t - 1$
- Si $l_2 < l_1$: pour $t \in \llbracket\, l_2, l_1 \,\rrbracket$: $b_t \leftarrow b_t + 1$

2.4.2.3 Représentation canonique des migrations de lignes

L'ordre dans lequel on effectue des migrations de lignes n'ayant pas d'influence, si on suppose que le coût des migrations de lignes est constant (c'est à dire que toutes les migrations de lignes sont équivalentes), un ensemble de migrations de lignes peut être vu comme une liste de couples (abscisse, nombre de lignes à insérer ou à enlever). Ainsi, un ensemble de migrations de lignes se réduit à un mot sur $\{+, -\}$ comportant autant de $+$ que de $-$.



Lemme 2.1. *Soit u un mot de longueur finie sur $\{+, -\}$ et p le plus petit préfixe non vide de u comportant autant de $+$ que de $-$ (s'il existe). Alors $\varphi(p)$, où φ est un morphisme de $\{+, -\}$ vers $\{(,)\}$ associant p_1 à "(" et $p_{|p|}$ à ")", est bien parenthésé.*

En itérant le procédé, on ramène donc tout ensemble de migrations de lignes sous une forme canonique où les migrations sont bien parenthésées.

Propriété. *Dans un ensemble de migrations de lignes bien parenthésé, toutes les migrations de lignes «encadrées» par une migration de ligne (l_1, l_2) se font dans le même sens que (l_1, l_2).*

Lemme 2.2. $\forall x \in \mathbb{R} : \forall k \in \mathbb{N} :$ $|x + k + 1| - |x + k| \geqslant |x + 1| - |x|$
$|x - k - 1| - |x - k| \geqslant |x - 1| - |x|$

Lemme 2.3. *Soit $e_2, \dots, e_n$ un ensemble de migrations de lignes bien parenthésé. Si (l_1, l_2) (avec $l_1 < l_2$) est une migration ne remettant pas en cause le parenthèsage de $e_2, \dots, e_n$ alors si on note b'_t la valeur de b_t après l'application de $e_2, \dots, e_n$:*

$$\sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t - 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t| \Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t - 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t|$$

En revanche, si $l_1 > l_2$ et que (l_1, l_2) ne remet pas en cause le bon parenthèsage de $e_2, \dots, e_n$, alors on a :

$$\sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t + 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t| \Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t + 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t|$$

Démonstration. Notons $E = \{e_2, \dots, e_n\}$ un ensemble de migrations de lignes bien parenthésé. Notons E' l'ensemble des migrations encadrées par (l_1, l_2).

Si $l_1 < l_2$ alors toutes les migrations de lignes de E' sont de la forme (j_1, j_2) avec $j_1 < j_2$ (puisque l'ajout de (l_1, l_2) à E ne remet pas en cause son bon parenthèsage). Ainsi pour tout $t \in \langle\!\langle\, l_1, l_2 \,\rangle\!\rangle$, $b'_t = b_t - |\{e \in E'/t \in \langle\!\langle\, e \,\rangle\!\rangle\}|$ puisque les migrations se font toutes dans le même sens, c'est-à-dire de l_1 vers l_2. On a donc :

$$\begin{aligned}
\sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t - 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t| &\Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t(|b'_t - 1| - |b'_t|) < 0 \\
&\Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t(|b_t - 1| - |b_t|) < 0 \text{(en utilisant le lemme 2.2)} \\
&\Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t - 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t|.
\end{aligned}$$

Si $l_1 > l_2$ alors toutes les migrations de lignes de E' sont de la forme (j_1, j_2) avec $j_1 > j_2$ (puisque l'ajout de (l_1, l_2) à E ne remet pas en cause son bon parenthèsage). Ainsi pour tout $t \in \langle\!\langle\, l_1, l_2 \,\rangle\!\rangle$, $b'_t = b_t + |\{e \in E'/t \in \langle\!\langle\, e \,\rangle\!\rangle\}|$ puisque les migrations se font toutes dans le même sens, c'est-à-dire de l_2 vers l_1. En utilisant le lemme 2.2, on a :

$$\begin{aligned}
\sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t + 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b'_t| &\Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} (w_t|b'_t + 1| - |b'_t|) < 0 \\
&\Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t(|b_t + 1| - |b_t|) < 0 \text{(en utilisant le lemme 2.2)} \\
&\Rightarrow \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t + 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t|.
\end{aligned}$$

■

2.4.2.4 Choix glouton des migrations de lignes

Une fois les migrations de colonnes effectuées, se pose la question de savoir s'il existe des migrations de lignes améliorant le coût du rééquilibrage par pousse-pousse.

Lemme 2.4. *S'il existe un ensemble de migrations de lignes améliorant le coût du pousse-pousse, alors, il existe une migration améliorant le coût du pousse-pousse.*

Démonstration. Supposons par l'absurde qu'il existe un ensemble E de migrations de lignes qui permette de diminuer le coût d'un rééquilibrage complet par pousse-pousse et qu'aucune migration de ligne ne permette à elle seule de diminuer ce coût. On prendra E le plus petit possible (et donc de taille supérieure ou égale à 2) et bien parenthésé puisque cela ne modifie pas le coût du rééquilibrage par pousse-pousse.

S'il existe (l_1, l_2) une migration de ligne de E qui n'est encadrée par aucune autre migration de ligne et qui encadre au moins une migration de ligne. En utilisant le lemme 2.3 avec (l_1, l_2) et $E\backslash\{(l_1, l_2)\}$ (E étant le plus petit possible, $E\backslash\{(l_1, l_2)\}$ est moins efficace pour diminuer le coût du rééquilibrage par pousse-pousse que E) , on en déduit que (l_1, l_2) diminue le coût du pousse-pousse sur la zone (l_1, l_2). En considérant E' l'ensemble des migrations de E qui ne sont pas encadrées (au sens large) par (l_1, l_2), on obtient donc que $E' \cup (l_1, l_2)$ améliore également le coût du rééquilibrage par pousse-pousse, ce qui entre en contradiction avec la minimalité de la taille de E.

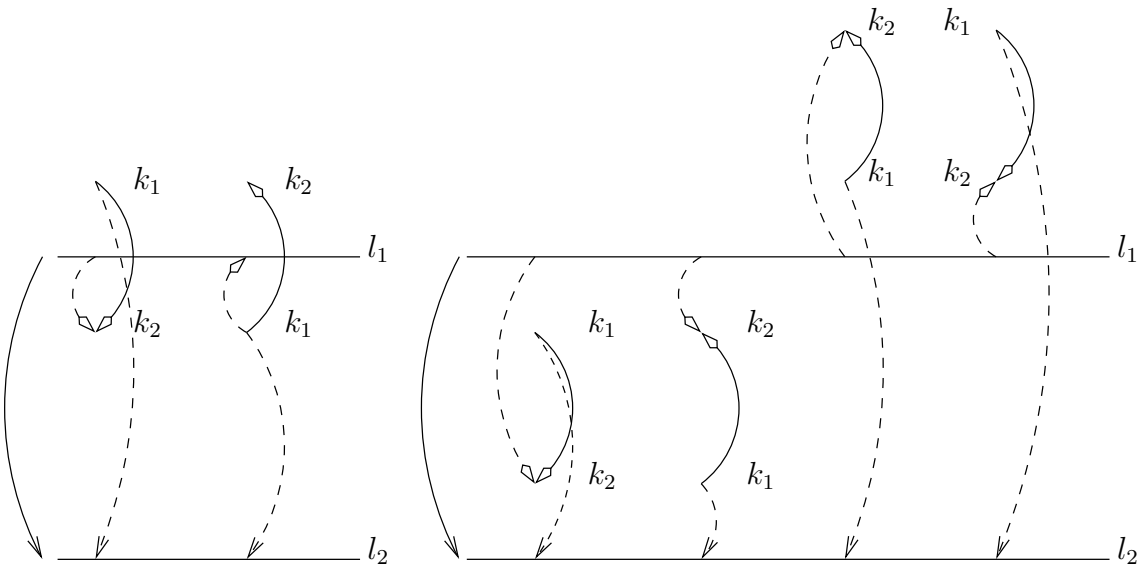
Si toutes les migrations de lignes sont indépendantes les unes des autres alors n'importe laquelle d'entre elle améliore le coût du rééquilibrage par pousse-pousse, ce qui est également absurde. ■

Définition 2.8 (Choix glouton de migration des ligne). *On appellera* Choix glouton de migration des ligne *le choix du couple de ligne (l_1, l_2) qui, parmi tous les autres couples de lignes, minimise le coût du rééquilibrage par pousse-pousse après migration de l_1 vers l_2.*

On notera que le nombre de couples de lignes à tester est majoré par $p(p-1)$, que l'évaluation du coût du rééquilibrage par pousse-pousse après une migration de ligne est un $O(p)$ et que ce choix peut donc s'effectuer assez simplement.

Lemme 2.5. *Pour tout $k \in \mathbb{N}$, il existe un ensemble de k migrations de lignes minimisant (parmi les ensembles de k migrations de lignes) le coût du rééquilibrage par pousse-pousse et qui touche les mêmes lignes que la migration de ligne choisie de façon gloutonne.*

Démonstration. Soit $k \in \mathbb{N}$. Supposons que Glouton ait choisi de déplacer la ligne l_1 vers l_2 (par symétrie, on peut supposer $l_1 < l_2$) et qu'aucun ensemble optimal de k migrations de lignes ne fasse intervenir l_1. Soit E un ensemble de k migrations de lignes sous forme canonique (c'est-à-dire bien parenthésé) ne faisant pas intervenir l_1 et minimisant le coût du rééquilibrage par pousse-pousse (parmi tous les ensembles de k migrations de lignes). On va exhiber un ensemble de k migrations de lignes E' faisant intervenir l_1 (et donc par hypothèse moins bon que E) et arriver à une contradiction sur le choix de l'algorithme glouton.



(a) Une des migrations de lignes de E encadre l_1

(b) Aucune des migrations de lignes de E n'encadre l_1

FIG. 2.21 – Différents cas de figure

Cas 1 : Supposons que l'une des migrations de lignes de E encadre l_1 (figure 2.21(a)). Notons $E = \{(k_1, k_2), e_2, \ldots, e_n\}$ avec (k_1, k_2) la migration de ligne encadrant l_1 la plus imbriquée. Notons $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ si $k_1 < k_2$ ($E' = \{(k_1, l_1), e_2, \ldots, e_n\}$ si $k_2 < k_1$, le reste de la démonstration étant similaire). E' est bien parenthésé et E' étant un ensemble de migrations de lignes valide moins bon que E, on va montrer que (k_1, l_2) est une migration de ligne plus rentable que (l_1, l_2) et donc que le choix effectué par l'algorithme glouton est absurde. Notons $b'_{i,j}$ les quantités à transférer après les migrations $\{e_2, \ldots, e_n\}$. $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ étant moins bon que $E = \{(k_1, k_2), e_2, \ldots, e_n\}$, on a :

$$\sum_{\langle\!\langle k_1,k_2 \rangle\!\rangle} w_t|b'_t - 1| < \sum_{\langle\!\langle k_1,l_1 \rangle\!\rangle} w_t|b'_t| + \sum_{\langle\!\langle l_1,k_2 \rangle\!\rangle} w_t|b'_t - 1| \quad \text{soit, en décomposant } \langle\!\langle\, k_1, k_2 \,\rangle\!\rangle$$

$$\sum_{\langle\!\langle k_1,l_1 \rangle\!\rangle} w_t|b'_t - 1| < \sum_{\langle\!\langle k_1,l_1 \rangle\!\rangle} w_t|b'_t| \quad \text{et donc (lemme 2.3)}$$

$$\sum_{\langle\!\langle k_1,l_1 \rangle\!\rangle} w_t|b_t - 1| < \sum_{\langle\!\langle k_1,l_1 \rangle\!\rangle} w_t|b_t| \quad \text{d'où}$$

$$\sum_{\langle\!\langle k_1,l_2 \rangle\!\rangle} w_t|b_t - 1| < \sum_{\langle\!\langle k_1,l_1 \rangle\!\rangle} w_t|b_t| + \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} |b_t - 1|$$

(k_1, l_2) est donc une migration de ligne plus intéressante pour Glouton que (l_1, l_2),

ce qui est absurde.

Cas 2 : Supposons qu'aucune des migrations de lignes de E n'encadre l_1 (figure 2.21(b)). On va supposer que (l_1, l_2) encadre une migration de lignes et que cette dernière est dans le même sens que (l_1, l_2) mais les autres cas se traitent de façon similaire (voir figure 2.21(b)). Notons $E = \{(k_1, k_2), e_2, \ldots, e_n\}$ avec (k_1, k_2) la migration de ligne encadrée par (l_1, l_2) la moins imbriquée et la plus proche de l_1. Alors $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ étant un ensemble de migrations de lignes bien parenthésé, valide et moins bon que E, on va montrer -comme précédemment- que (k_1, l_2) est une migration de ligne plus rentable que (l_1, l_2) et que le choix effectué par l'algorithme glouton est absurde. Notons $b'_{i,j}$ les quantités à transférer après les migrations $\{e_2, \ldots, e_n\}$. $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ étant moins bon que $E = \{(k_1, k_2), e_2, \ldots, e_n\}$, on a :

$$\sum_{\langle\!\langle l_1,k_1 \rangle\!\rangle} w_t|b'_t| + \sum_{\langle\!\langle k_1,k_2 \rangle\!\rangle} w_t|b'_t - 1| < \sum_{\langle\!\langle l_1,k_2 \rangle\!\rangle} w_t|b'_t - 1| \quad \text{soit, en décomposant } \langle\!\langle\, l_1, k_2 \,\rangle\!\rangle$$

$$\sum_{\langle\!\langle l_1,k_1 \rangle\!\rangle} w_t|b'_t| < \sum_{\langle\!\langle l_1,k_1 \rangle\!\rangle} w_t|b'_t - 1| \quad \text{et donc (lemme 2.3)}$$

$$\sum_{\langle\!\langle l_1,k_1 \rangle\!\rangle} w_t|b_t| < \sum_{\langle\!\langle l_1,k_1 \rangle\!\rangle} w_t|b_t - 1| \quad \text{d'où}$$

$$\sum_{\langle\!\langle l_1,k_1 \rangle\!\rangle} w_t|b_t| + \sum_{\langle\!\langle k_1,l_2 \rangle\!\rangle} w_t|b_t - 1| < \sum_{\langle\!\langle l_1,l_2 \rangle\!\rangle} w_t|b_t - 1|$$

(k_1, l_2) est donc une migration de ligne plus intéressante pour glouton que (l_1, l_2), ce qui est absurde. ■

Notons $(l_1^{(1)}, l_2^{(1)})$ un choix glouton de migration de ligne, notons $(l_1^{(2)}, l_2^{(2)})$ un choix glouton de migration de ligne une fois $(l_1^{(1)}, l_2^{(1)})$ appliquée et, de manière générale, notons $(l_1^{(k)}, l_2^{(k)})$ une migration de ligne choisie de façon gloutonne une fois $(l_1^{(1)}, l_2^{(1)})$,...,$(l_1^{(k-1)}, l_2^{(k-1)})$ appliquée.

Notons $E_k = \{(l_1^{(1)}, l_2^{(1)}), \ldots, (l_1^{(k)}, l_2^{(k)})\}$ les k premières migrations de lignes choisies par l'algorithme glouton.

Lemme 2.6. *Pour tout k, E_k minimise le coût du rééquilibrage par pousse-pousse parmi les ensembles de k migrations de ligne.*

Démonstration. Soit k entier. Soit E un ensemble de k migrations de lignes optimal pour le pousse-pousse. On peut donc supposer que E touche $l_1^{(1)}$ et$l_2^{(1)}$ et quitte à reparenthéser E, on peut donc se ramener au cas où $E = (l_1^{(1)}, l_2^{(1)}) \cup E'$.

E' est donc un ensemble de ligne optimal pour le pousse-pousse, une fois $(l_1^{(1)}, l_2^{(1)})$ appliquée. Mais on peut donc supposer que E' touche $l_1^{(2)}$ et $l_2^{(2)}$ et quitte à reparenthéser E', on peut donc se ramener au cas où $E' = (l_1^{(2)}, l_2^{(2)}) \cup E''$.

En procédant par récurrence, on montre donc que E_k est un ensemble de k migrations de lignes minimisant le coût du pousse-pousse. ■

2.4.2.5 Algorithme

Nous présentons un algorithme glouton qui permet d'obtenir un rééquilibrage optimal d'une distribution avec un coût minimal (en utilisant uniquement les transformations présentées en section 2.4.1).

ÉQUILIBRAGE_GLOUTON($D = (\delta_1, \ldots, \delta_p)$)
1: **Tant que** D n'est pas équilibrée entre chaque colonne **:**
2: Effectuer une migration de colonne diminuant la charge de la colonne la plus excédentaire et augmentant celle de la plus déficitaire.
3: $D' \leftarrow D$
4: $k \leftarrow 1$
5: $k_{opt} \leftarrow 0$
6: $C_{opt} \leftarrow$ PP(D') {PP(D') représente le coût du rééquilibrage par pousse-pousse de D'}
7: **Tant que** il existe une migration de ligne améliorant un pousse-pousse complet sur D' **:**
8: Soit $(l_1^{(k)}, l_2^{(k)})$ la migration de ligne améliorant le plus le coût d'un rééquilibrage complet par pousse-pousse de D'
9: Appliquer $(l_1^{(k)}, l_2^{(k)})$ à D'
10: **Si** Coût($(l_1^{(1)}, l_2^{(1)}), \ldots, (l_1^{(k)}, l_2^{(k)})$)+PP($D'$)$< C_{opt}$ **Alors**
11: $C_{opt} \leftarrow$ Coût($(l_1^{(1)}, l_2^{(1)}), \ldots, (l_1^{(k)}, l_2^{(k)})$)+PP($D'$)
12: $k_{opt} \leftarrow k$
13: Incrémenter k
14: Appliquer $(l_1^{(1)}, l_2^{(1)}), \ldots, (l_1^{(k_{opt})}, l_2^{(k_{opt})})$ à D
15: Finir l'équilibrage de D avec des opérations de pousse-pousse

Algorithme 2.2: Algorithme de rééquilibrage glouton

L'algorithme 2.2 se décompose en trois phases. Comme expliqué en section 2.4.2.1, on détermine d'abord les migrations globales de colonnes (lignes 1 et 2) nécessaires à l'annulation du déséquilibre de chacune des colonnes de processeurs. Ensuite, on choisit de façon gloutonne les migrations globales de lignes à effectuer de façon à réduire au mieux le coût du pousse-pousse (lignes 7 et 8). Nous avons vu dans la section 2.4.2.2 comment évaluer le coût du pousse pousse et l'impact d'une migration de lignes sur ce coût. Il suffit donc d'effectuer ce calcul pour chacun des couples de lignes possibles. On construit donc incrémentalement les $(l_1^{(1)}, l_2^{(1)}), \ldots, (l_1^{(k)}, l_2^{(k)})$ et on choisit l'ensemble de migrations de lignes $E_{k_{opt}}$ qui minimise la somme du coût des migrations $E_{k_{opt}}$ et du rééquilibrage après application de ces migrations.

Enfin, on finit de rééquilibrer la distribution avec un pousse-pousse complet dans chacune des colonnes (ligne 15).

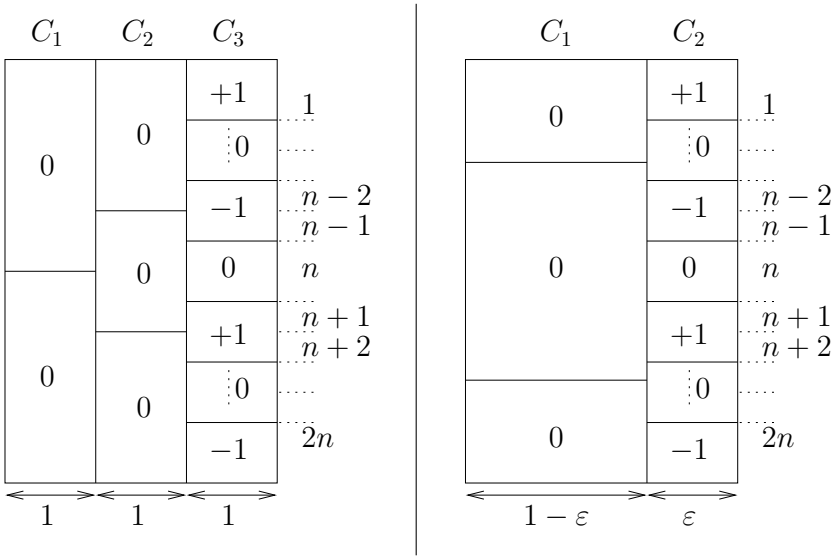


FIG. 2.22 – Contre-exemples du choix glouton en cas de coût de migrations de lignes non constant.

Théorème 2.3. *Si le coût des migrations de lignes est constant, l'algorithme 2.2 est optimal.*

Démonstration. Si le coût des migrations de lignes est constant alors, pour tout k, E_k minimise, parmi les ensembles de k migrations de lignes, la somme des coût de migrations de lignes et de rééquilibrage par pousse-pousse (grâce au lemme 2.6).

$E_{k_{opt}}$ est donc l'ensemble de migrations de lignes qui rééquilibre de façon optimale la distribution initiale. ■

2.4.2.6 Relâchement des hypothèses

Cet algorithme est donc relativement efficace mais n'est correct que si l'on suppose que le coût des migrations de lignes est constant.

Proposition 2.1. *Si on retire l'hypothèse que le coût d'une migration de ligne est constant, l'algorithme glouton précédent n'est plus optimal.*

Démonstration. Nous exhibons un exemple pour lequel l'algorithme 2.2 ne trouve pas les bonnes opérations à effectuer.

Dans la distribution de gauche de la figure 2.22, seule la colonne 3 est déséquilibrée. Il est donc naturel de rééquilibrer avec $PP(C_3)$, ce qui coûte $2n-6$. Cependant, il y a moyen de rééquilibrer la distribution en n'effectuant que des migrations de lignes $\big(L(1,n-2);L(n+2,2n)\big)$ pour un coût très faible égal à $1+1=2$.

Le meilleur choix glouton est $L(1, 2n); PP(C_1); PP(C_2); PP(C_3)$ (et coûte $3+1+2+2 = 8$). Le choix suivant porte sur $L(n+2, n-2)$, ce qui nous permet de rééquilibrer totalement la situation pour un coût égal à $3 + 3 = 6$.

Glouton propose donc *in fine* de faire les migrations $\{(1, 2n), (n+2, n-2)\}$ alors que la solution optimale est $\{(1, n-2), (n+2, 2n)\}$. Le choix glouton permet effectivement de déterminer les bonnes lignes à migrer mais pas de les associer correctement les unes aux autres.

On remarquera que la bonne façon de les associer, ici, consiste à utiliser un bon parenthésage. Il ne suffit cependant pas de bien parenthéser un ensemble de migration de lignes pour en minimiser le coût. Regardons en effet la distribution de droite de la figure 2.22. Sur cet exemple, les migrations $\{(1, 2n), (n+2, n-2)\}$ ont un coût égal à $1 + \varepsilon$ alors que les migrations $\{(1, n-2), (n+2, 2n)\}$ ont un coût égal à $1 + 1$. ■

Cette hypothèse n'est plus valable dès que le débit des connexions n'est pas homogène ou quand on a une grande disparité de vitesses. En revanche, les lemmes 2.4, 2.5 et 2.6 restent valides, même sans supposer que les temps de communications d'un processeur à l'autre sont homogènes. Pour rééquilibrer de façon optimale la distribution, il faut donc savoir quel E_k utiliser et comment associer entre elles les migrations de lignes. Il n'est pas non plus exclu qu'il existe un ensemble de migrations de lignes non optimal pour le coût du rééquilibrage par pousse-pousse mais dont le faible coût compense cette non-optimalité. La complexité exacte de ce problème est donc encore une question ouverte.

2.5 Conclusion

Dans ce chapitre, nous avons montré que la mise en œuvre de schémas de distributions statiques efficaces pour des plates-formes hétérogènes était un problème difficile même pour de simples noyaux d'algèbre linéaire comme le produit matriciel ou la factorisation LU. Même les cas simples où l'hétérogénéité est uniquement présente dans les vitesses de calcul (voir section 2.2.3) ou ceux où l'hétérogénéité est uniquement présente dans les temps de communication (voir [14]), conduisent à des résultats de NP-complétude.

Nous avons rappelé en section 2.2.3 les travaux menés avant notre arrivée dans notre groupe sur les distributions de données hétérogènes adaptées au produit matriciel. Même si ces fonctions de coût peuvent paraître simplistes, nous avons montré en section 2.2.4 qu'elles conduisent en pratique à des améliorations significatives. L'introduction d'une modélisation un peu plus fine a été initiée dans [14] où nous montrons que si on se place dans le cas où toutes les communications sont séquentielles, l'hétérogénéité des vitesses de communications suffit à rendre le problème NP-complet. Les travaux d'Hélène Renard, actuellement en thèse avec Yves Robert et Frédéric Vivien vont également dans ce sens puisqu'ils portent sur des partitionnements monodimensionnels en utilisant une modélisation extrêmement fine (prise en compte de la topologie de la plate-forme et donc des phénomènes de contention) du type de celles utilisées dans les chapitres 4, 5 et 6.

Les problèmes sont évidemment encore plus durs puisque déterminer une distribution monodimensionnelle hétérogène efficace devient un problème NP-complet [92].

Nous avons également montré en section 2.3 comment adapter les distributions développées pour le produit de matrices afin qu'elles soient performantes pour les factorisations LU. L'algorithme incrémental est optimal pour les distributions monodimensionnelles et asymptotiquement optimal pour toute autre distribution pourvu qu'elle soit parfaitement équilibrée pour le produit de matrices. De plus, comme nous n'effectuons que des permutations sur les colonnes et sur les lignes à partir d'une distribution adaptée au produit matriciel, ces distributions sont valables pour les deux noyaux (exactement comme dans le cas homogène où les distributions cycliques par blocs sont efficaces à la fois pour la factorisation LU et le produit matriciel).

Dans le cadre du calcul sur des plates-formes hétérogènes, des stratégies de distributions purement statiques risquent de poser quelques problèmes. En effet, dans un environnement non dédié, des variations de la charge des processeurs peuvent survenir. Ces problèmes de variations de charge peuvent être traités à l'aide de redistributions légères ayant lieu entre des phases bien déterminées de l'algorithme. Ces noyaux d'algèbre linéaire étant fortement couplés, des approches purement dynamiques sont inefficaces en raison des mouvements de données incessants. Des distributions statiques remises régulièrement en jeu à l'aide de redistributions légères sont donc une bonne alternative. Toutes les heuristiques que nous avons présentées conduisent à des distributions en colonnes, ce qui fournit un cadre de travail unifié pour étudier ces techniques de redistributions et nous a permis de proposer des opérations élémentaires de redistribution et de montrer comment, sous certaines conditions, les utiliser pour obtenir un rééquilibrage optimal (en terme de charge de travail) à moindre coût.

Chapitre 3

Ordonnancement sur plate-forme hétérogène

3.1 Introduction

La paradigme maître-esclave consiste en l'exécution de tâches indépendantes sur un ensemble de processeurs, appelés *esclaves*, sous la houlette d'un processeur particulier, le *maître*. Cette technique est utilisée depuis longtemps en raison de sa stabilité et de la simplicité de sa mise en œuvre. Dans un cadre homogène, son efficacité est évidente (fermes de processeurs [44]). Néanmoins, sur une plate-forme où les processeurs n'ont pas tous la même puissance de calcul et où les temps de communication du maître vers les esclaves ne sont pas uniformes, il est nécessaire de prendre en compte cette hétérogénéité pour déterminer et pour utiliser au mieux la puissance de calcul agrégée de cette plate-forme.

Nous nous intéressons dans ce chapitre à l'extension de ce paradigme aux plates-formes hétérogènes. Dans la section 3.2, nous présentons la modélisation de la plate-forme que nous utilisons tout au long de ce chapitre. Nous y présentons également en détail les différents types d'applications auxquels nous nous intéressons, ainsi que la métrique que nous cherchons à minimiser et les problèmes d'optimisation qui en découlent. Dans la section 3.3, nous proposons un algorithme polynomial qui donne une solution optimale au problème de l'allocation des tâches lorsqu'une seule communication est nécessaire avant le traitement des tâches sur les différents processeurs. Lorsqu'une communication avant et une communication après le traitement des tâches sont nécessaires, la situation est plus complexe. Nous montrons dans la section 3.4 que le problème est NP-complet. Nous présentons également un algorithme d'approximation polynomial. Enfin, nous montrons dans la section 3.5 que quand une communication est nécessaire avant le traitement de chacune des tâches, le problème redevient polynomial. Nous revenons sur ce dernier problèmes dans les chapitres 4, 5 et 6, en nous plaçant en régime permanent. Comme nous le verrons, ce changement de métrique permet de simplifier le problème et de l'adapter à des plates-formes plus générales et au cas où les tâches recèlent du parallélisme interne.

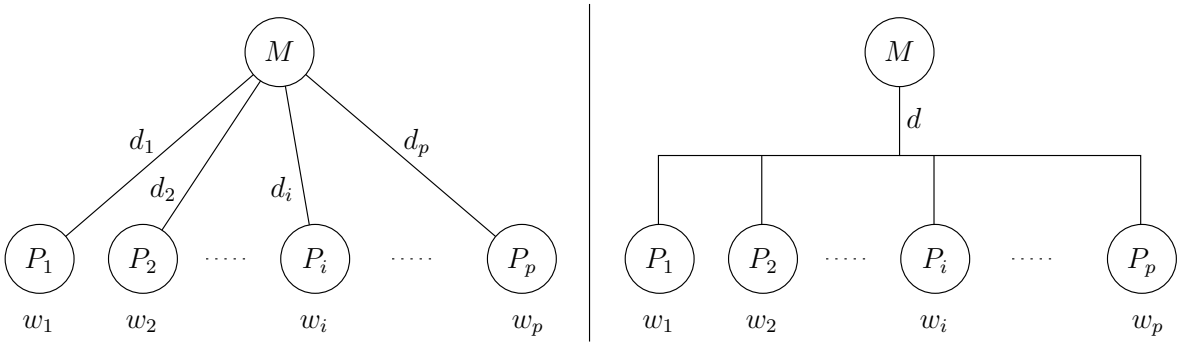


FIG. 3.1 – Architecture maître/esclave : à gauche une organisation «en étoile», à droite une organisation en «bus». Dans les deux cas, on suppose qu'une seule communication entre le maître et un esclave peut avoir lieu à la fois.

La section 3.6 conclut ce chapitre en donnant des pistes sur les extensions possibles de ces résultats et de ces problèmes.

3.2 Modèles

Le modèle de plate-forme que nous utilisons dans ce chapitre est représenté figure 3.1. Le maître M et les p esclaves $P_1, P_2, \ldots, P_p$ sont reliés par un réseau aux capacités variables. On suppose que le maître n'est capable de communiquer qu'avec *un seul* de ses esclaves à la fois (modèle 1-port en sortie). Le temps de communication d'un message élémentaire de M vers P_i est noté d_i.

On suppose disposer d'un ensemble de tâches indépendantes représentant la même quantité de calcul. Le temps nécessaire au processeur P_i pour traiter une tâche est noté w_i et représente le temps de cycle du processeur P_i. On notera c_i le nombre de tâches exécutées par le processeur P_i. Indépendamment des hypothèses que l'on peut faire sur le schéma de communications, on peut distinguer deux problèmes d'optimisation :

Définition 3.1 (MinTime(C)). *Étant donné un nombre de tâches C, déterminer la meilleure allocation des tâches aux esclaves, c'est-à-dire l'allocation $\mathcal{C} = \{c_1, \ldots, c_p\}$ telle que $\sum_{i=1}^{p} c_i = C$ et qui minimise le temps total d'exécution.*

Ce problème est appelé *NOW-rental problem* dans [94].

Définition 3.2 (MaxTasks(T)). *Étant donné un temps T, déterminer la meilleure allocation des tâches aux esclaves, c'est-à-dire l'allocation $\mathcal{C} = \{c_1, \ldots, c_p\}$ telle que chacun des p processeurs termine ses calculs en un temps T et qui maximise le nombre de tâches traitées $C = \sum_{i=1}^{p} c_i$*

Ce problème est appelé *NOW-exploitation problem* dans [94].

Dans ce chapitre, nous nous intéressons à la résolution du second problème MaxTasks(T). Si l'on sait résoudre ce problème en temps polynomial, il est également possible de résoudre MinTime(C) en temps polynomial en effectuant une recherche dichotomique sur T.

Dans les sections suivantes, nous formulons plus précisément les problèmes d'optimisation que nous allons étudier.

3.2.1 Sans aucune communication

Intéressons-nous d'abord au cas simple où aucune communication n'est nécessaire. La solution du problème MaxTasks(T) est alors triviale. En effet, chaque processeur peut travailler indépendamment des autres pendant T unité de temps. Les seules contraintes que l'on doive respecter s'écrivent donc :

$$\forall P_i : c_i \times w_i \leqslant T \qquad \text{soit} \qquad \forall P_i : c_i \leqslant \frac{T}{w_i}$$

Les c_i étant entiers, en posant $c_i = \lfloor T/w_i \rfloor$ pour $1 \leqslant i \leqslant p$, on obtient clairement une allocation valide et optimale.

3.2.2 Avec une communication initiale (Scatter)

La formulation de ce problème est reprise d'un article écrit par Andonie, Chronopoulos, Grosu et Galmeanu [4]. Cet article porte sur la mise en œuvre, utilisant la bibliothèque PVM [59], d'un calcul distribué de rétro-propagation dans un réseau de neurones. L'apprentissage du réseau de neurones est divisé en différentes phases de calcul. À chaque étape, le motif à apprendre est distribué aux différents esclaves, qui sont donc des processeurs de vitesses différentes. Avant d'exécuter une tâche, chaque esclave doit donc attendre d'avoir reçu des données du maître.

Dans ce chapitre, mise à part la section 3.5 où la plate-forme est une étoile, nous nous restreignons au cas où la plate-forme cible est organisée en «bus» (voir figure 3.1). Pour plus de clarté, nous noterons t_{com}le temps de communication entre le maître et l'un quelconque de ses esclaves. Les communications étant exclusives, les p messages doivent être envoyés les uns après les autres. On peut sans restriction supposer que les messages sont envoyés au plus tôt, c'est à dire aux temps $0, t_{\text{com}}, 2t_{\text{com}}, \ldots, (p-1)t_{\text{com}}$. Le problème se ramène donc à déterminer l'ordre d'envoi des messages, c'est-à-dire une permutation σ de $[\![1, p]\!]$ telle que le processeur P_i communique dans l'intervalle de temps $[(\sigma(i)-1)t_{\text{com}}, \sigma(i)t_{\text{com}}]$. Notre premier problème d'optimisation peut donc se formuler de la façon suivante :

Définition 3.3 (MaxTasks1(T)). *Étant donné une borne T, déterminer la meilleure allocation des tâches aux esclaves, c'est-à-dire une permutation σ et une allocation $\mathcal{C} =$*

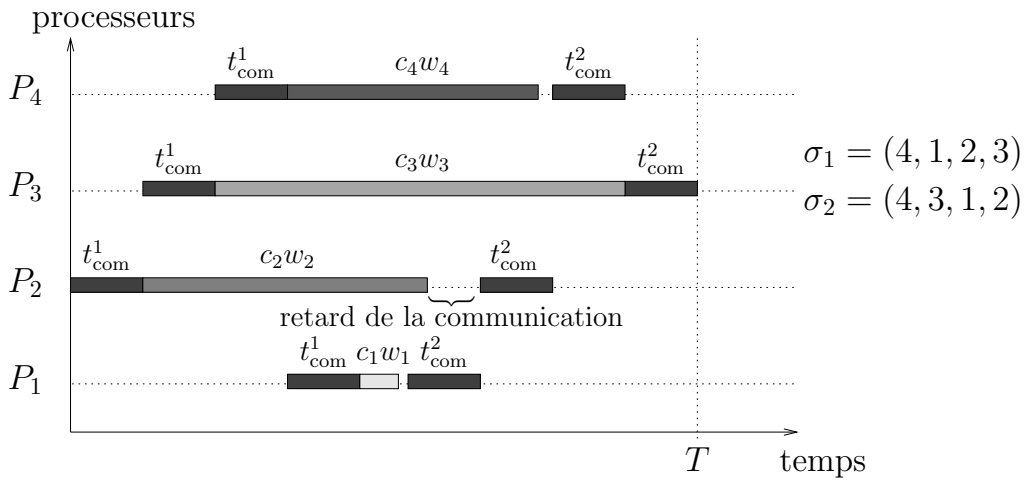


FIG. 3.2 – Retard des messages de retour.

$\{c_1, \ldots, c_p\}$ *telle que chacun des* p *processeurs termine ses calculs en un temps* T *et qui maximise le nombre de tâches traitées* $C = \sum_{i=1}^{p} c_i$ *:*

$$\max \left(\sum_{i=1}^{p} c_i \;\middle|\; \sigma \text{ est une permutation et } \forall i \in [\![1, p]\!] : \sigma(i) t_{com} + c_i w_i \leqslant T \right)$$

3.2.3 Avec une communication initiale et une communication finale (Scatter/Gather)

Dans le modèle d'Andonie et al. [4], il n'y a pas de coût de communication pour envoyer une réponse au maître. Quand les esclaves calculent une réponse du type «oui/non», ces communications peuvent effectivement être négligées par rapport aux communications initiales et aux temps de calcul. Dans cette section, nous définissons le problème d'optimisation, cousin du précédent, où les esclaves doivent renvoyer une réponse de taille non négligeable au maître. Les messages initiaux et finaux pouvant être de tailles différentes, nous modélisons cette situation par l'introduction de deux coûts de communications distincts : t^1_{com} pour les messages envoyés par le maître aux esclaves et t^2_{com} pour les messages envoyés par les esclaves au maître.

Comme précédemment, nous cherchons une permutation σ_1 qui détermine l'ordre des émissions initiales : le maître envoie ses données au processeur P_i durant l'intervalle de temps $[(\sigma_1(i) - 1)t^1_{\text{com}}, \sigma_1(i)t^1_{\text{com}}]$. Mais nous cherchons également une seconde permutation σ_2 qui détermine l'ordre des émissions finales : étant donné une borne T, l'esclave P_i communique les résultats de ses calculs au maître dans l'intervalle de temps $[T - \sigma_2(i)t^2_{\text{com}}, T - (\sigma_2(i) - 1)t^2_{\text{com}}]$. σ_2 correspond donc à l'ordre inverse des communications retour (voir figure 3.2). On peut chercher les solutions sous cette forme sans perte de généralité car, quitte à retarder sa communication retour, il est toujours possible de

faire communiquer un esclave avec le maître durant cette période (voir figure 3.2). Nous pouvons donc définir notre deuxième problème d'optimisation.

Définition 3.4 (MaxTasks2(T)). *Étant donné une borne T, déterminer la meilleure allocation des tâches aux esclaves, c'est-à-dire deux permutations, σ_1 et σ_2, et une allocation $\mathcal{C} = \{c_1, \ldots, c_p\}$ telle que chacun des p processeurs termine ses calculs en un temps T et qui maximise le nombre de tâches traitées $C = \sum_{i=1}^{p} c_i$:*

$$\max \left(\sum_{i=1}^{p} c_i \;\middle|\; \begin{array}{c} \sigma_1, \sigma_2 \text{ permutations et} \\ \forall i \in [\![1,p]\!] : \sigma_1(i) t^1_{com} + c_i w_i + \sigma_2(i) t^2_{com} \leqslant T \end{array} \right)$$

3.2.4 Avec une communication avant le traitement de chaque tâche (allocations de tâches indépendantes).

Un autre modèle d'application, probablement plus courant que les précédents, est celui où l'on dispose d'un ensemble de tâches indépendantes, de quantité de calcul équivalentes et disposant de données propres. Ainsi, une communication est donc nécessaire avant le traitement de chacune des tâches. Ce modèle est adapté à un grand nombre d'applications, tout particulièrement celles qui rentrent dans le cadre du calcul collaboratif comme SETI@home [120] ou la factorisation de nombres de Mersenne [90], même s'il est évident que les problèmes techniques auxquels sont d'abord confrontés de tels projets portent plutôt sur la dynamicité et la volatilité de leur plate-forme.

Nous résolvons ce problème dans le cas d'une plate-forme en étoile à l'aide d'un algorithme polynomial en section 3.5. Nous revenons sur ce problèmes dans les chapitres 4, 5 et 6, en nous plaçant en régime permanent, ce qui simplifie le problème et permet d'obtenir des résultats bien plus généraux.

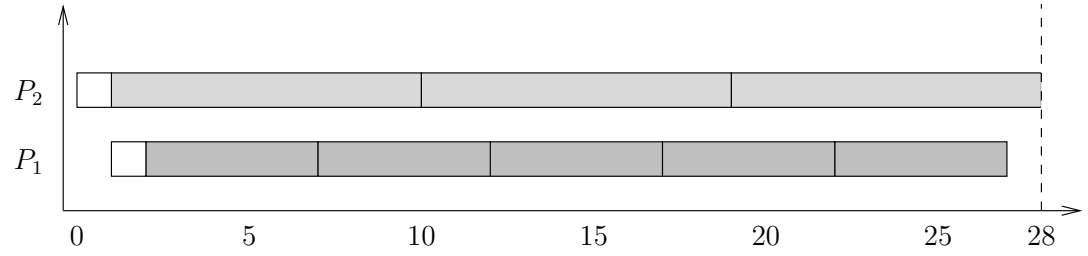
3.3 Scatter sur un bus

3.3.1 Recherche partielle

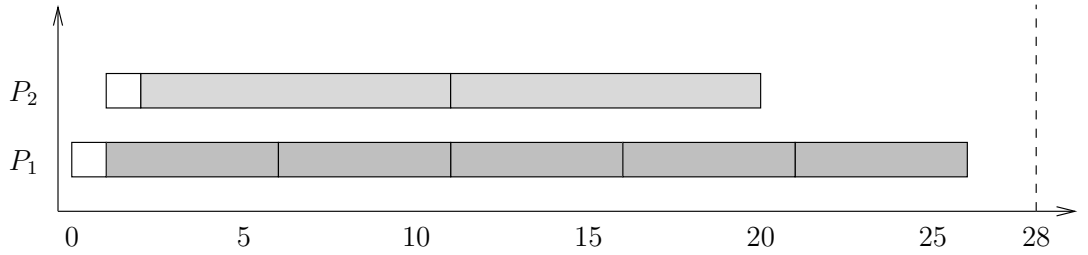
Afin de résoudre (partiellement) le problème MaxTasks1(T) défini en section 3.2.2, Andonie et al. [4] restreignent leur recherche aux allocations telles que les processeurs les plus rapides commencent à calculer en premier. Ils utilisent un algorithme de programmation dynamique pour résoudre le problème MinTime(C). Cette méthode, transposée pour le problème MaxTasks1(T) revient à trier les processeurs par temps de cycle de telle façon que $w_1 \leqslant w_2 \leqslant \cdots \leqslant w_p$ et à utiliser $\sigma = Id$. L'intuition sous-tendant cette approche est que les processeurs exécutant leurs tâches plus rapidement que les autres devraient travailler plus longtemps.

Cependant, cette intuition est parfois trompeuse. Intéressons-nous par exemple à l'instance composée de deux esclaves ($p = 2$) de temps de cycle $w_1 = 5$ et $w_2 = 9$ (voir

figure 3.3) et supposons enfin que $t_{\text{com}} = 1$. Alors, pour la borne $T = 28$, il est préférable de commencer par le processeur le plus lent (P_2). Il est alors capable de traiter 3 tâches, car $t_{\text{com}} + 3w_2 = 28 \leqslant T$, et le processeur le plus rapide (P_1), qui commence à calculer au temps $2t_{\text{com}}$, est capable d'en traiter 5 car $2t_{\text{com}} + 5w_2 = 27 \leqslant T$ (voir figure 3.3(b)). Si l'on avait commencé par le processeur le plus rapide, P_1 aurait encore exécuté 5 tâches, mais P_2 n'aurait pu en exécuter que 2 (voir figure 3.3(a)) au lieu de 3.



(a) On commence par le processeur le plus lent ($\sigma(1) = 2, \sigma(2) = 1$) : 8 tâches



(b) On commence par le processeur le plus rapide ($\sigma(1) = 1, \sigma(2) = 2$) : 7 tâches

FIG. 3.3 – L'intuition est parfois trompeuse. Quand $w_1 = 5$, $w_2 = 9$, $t_{\text{com}} = 1$ et $T = 28$, il est préférable de commencer par le processeur le plus lent.

3.3.2 Couplage

Il existe cependant un algorithme pour résoudre en temps polynomial le problème MaxTasks1(T). L'idée consiste à utiliser un graphe biparti pondéré à $2p$ sommets comme celui représenté figure 3.4. Les sommets de gauche représentent les processeurs et les sommets de droite représentent les différentes valeurs possibles de σ. L'arête reliant P_i à S_j est pondérée par le nombre maximum de tâches que P_i peut traiter s'il reçoit ses données en position j, c'est-à-dire si $\sigma(i) = j$. Le poids de cette arête est donc égal à

$$W(P_i, S_j) = \left\lfloor \frac{T - jt_{\text{com}}}{w_i} \right\rfloor$$

Il est aisé de remarquer qu'un couplage parfait dans ce graphe biparti équivaut à une permutation σ. Le poids total d'un couplage parfait étant égal au nombre maximal

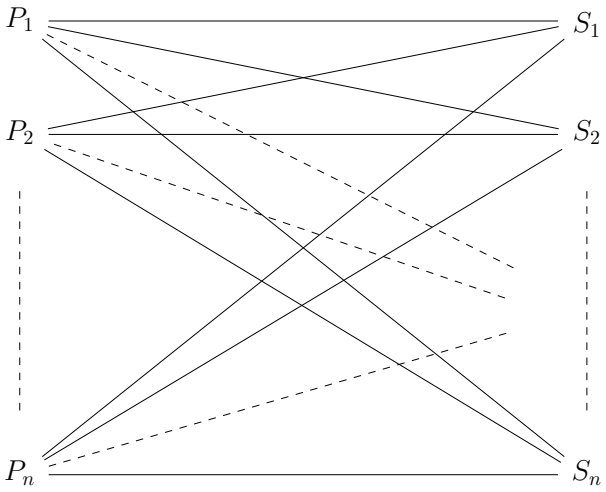


FIG. 3.4 – Graphe biparti permettant de résoudre MaxTasks1(T).

de tâches que l'on peut traiter si l'on utilise la permutation associée, notre problème se ramène à chercher un couplage de poids maximum dans ce graphe biparti. Des algorithmes efficaces existent pour résoudre ce problème [60, 114]. Plus précisément, il est possible de trouver un couplage de poids maximal dans un graphe biparti à $2p$ sommets en temps $O(p^{\frac{5}{2}})$ [66], ce qui nous permet d'établir le résultat suivant :

Théorème 3.1. *Le problème MaxTasks1(T) pour une instance à p processeurs peut être résolu en temps $O(p^{\frac{5}{2}})$ en utilisant un algorithme de recherche de couplage de poids maximal dans un graphe biparti.*

Notons que le coût de la recherche de la solution optimale est polynomial uniquement en p et ne dépend pas de T, ni du nombre de tâches exécutées.

3.4 Scatter/Gather sur un bus

La recherche de la solution du problème MaxTasks2(T), avec une communication initiale et une communication finale, s'avère bien plus difficile que le précédent. Nous montrons en section 3.4.1 que ce problème est NP-complet au sens fort. En section 3.4.2, nous présentons un algorithme d'approximation utilisant, une fois de plus, une recherche de couplage.

3.4.1 NP-complétude au sens fort de MaxTasks2(T)

Dans cette section, nous démontrons la NP-complétude de MaxTasks2(T). Le problème de décision associé à MaxTasks2(T) est le suivant :

Définition 3.5 (MaxTasks2-Dec(T, K)). *Étant donné une borne T et un entier K, existe-t-il une allocation des tâches aux esclaves, c'est-à-dire deux permutations, σ_1 et σ_2, et une allocation $\mathcal{C} = \{c_1, \ldots, c_p\}$, telle que chacun des p processeurs termine ses calculs en un temps au plus T et telle que le nombre de tâches traitées $\sum_{i=1}^{p} c_i$ est égal à K :*

$$\textit{Existe-t-il } \sigma_1, \sigma_2, \textit{ deux permutations, et } c_1, \ldots, c_p \textit{ tels que } \begin{cases} \sum_{i=1}^{p} c_i = K \\ \forall i \in [\![1, p]\!] : \sigma_1(i) t^1_{com} + c_i w_i + \sigma_2(i) t^2_{com} \leqslant T \end{cases} ?$$

Pour démontrer la NP-complétude au sens fort de MaxTasks2-Dec(T, K), nous utilisons une réduction à RN-3DM, une instance particulière de 3-Dimensional Matching dont la NP-complétude au sens fort a été démontrée par Yu [119]. Ce problème se définit comme suit :

Définition 3.6 (RN-3DM). *Soit $U = (u_1, u_2, \ldots, u_p)$ et e tel que $\sum_{i=1}^{p} u_i = pe - p(p+1)$. Existe-t-il deux permutations λ_1 et λ_2 telles que*

$$\forall i \in [\![1, p]\!] : \lambda_1(i) + \lambda_2(i) + u_i = e ?$$

Nous pouvons donc maintenant démontrer le théorème suivant :

Théorème 3.2. *MaxTasks2-Dec(T, K) est NP-complet au sens fort.*

Démonstration. Soit $U = (u_1, u_2, \ldots, u_p)$ et e une instance de RN-3DM. Considérons l'instance suivante de MaxTasks2-Dec(T, K).

$$\begin{cases} t^1_{\text{com}} = t^2_{\text{com}} = 1 \\ m = \max_{1 \leqslant i \leqslant p} u_i \\ w_i = u_i + e + m \text{ pour } i \in [\![1, p]\!] \\ T = 2e + m \\ K = p \end{cases}$$

La taille de cette instance est clairement polynomiale (et même linéaire) en la taille de l'instance originale de RN-3DM. Montrons que MaxTasks2-Dec(T, K) a une solution si et seulement si RN-3DM(U, e) en a une.

Supposons d'abord que RN-3DM ait une solution. Alors, soit λ_1 et λ_2 deux permutations telles que $\forall i \in [\![1, p]\!] : \lambda_1(i) + \lambda_2(i) + u_i = e$. Soit $\sigma_1 = \lambda_1$ et $\sigma_2 = \lambda_2$. Alors, pour tout $i \in [\![1, p]\!]$, on a

$$\begin{array}{lcl} \lambda_1(i) + \lambda_2(i) + u_i = e & \text{soit} & \sigma_1(i) + u_i + e + m + \sigma_2(i) = 2e + m \\ & \text{et donc} & \sigma_1(i) + w_i + \sigma_2(i) = T \end{array}$$

Chaque esclave peut donc traiter exactement une tâche en temps T, et nous avons donc construit une solution à notre instance de MaxTasks2-Dec(T, K).

Réciproquement, supposons que notre instance de MaxTasks2-Dec(T, K) ait une solution. Alors soit σ_1, σ_2 deux permutations et $\mathcal{C} = \{c_1, \ldots, c_p\}$ tels que :

$$\begin{cases} \sum_{i=1}^{p} c_i = p \text{ et} \\ \forall i \in [\![1, p]\!] : \sigma_1(i) + c_i w_i + \sigma_2(i) \leqslant T \end{cases}$$

Alors, pour tout $i \in [\![1, p]\!]$, on a :

$$\sigma_1(i) + \sigma_2(i) + c_i(u_i + e + m) \leqslant 2e + \max u_i.$$

Remarquons d'abord que $\forall i \in [\![1, p]\!] : c_i < 2$. En effet, s'il existait i tel que $c_i \geqslant 2$, alors on aurait

$$\begin{aligned} c_i(u_i + e + \max u_i) &\geqslant 2u_i + 2e + 2\max u_i \\ &> 2e + \max u_i = T, \end{aligned}$$

ce qui est impossible puisque cela signifierait que les tâches assignées au processeur P_i ne seraient pas traitées en un temps inférieur à T. Ainsi, comme $\sum_{i=1}^{p} c_i = p$ et que pour tout i on a $c_i < 2$, on a $c_i = 1$ pour tout i. En conséquence, si nous définissons $\lambda_1 = \sigma_1$ et $\lambda_2 = \sigma_2$, alors

$$\forall 1 \in [\![1, p]\!], \sigma_1(i) + u_i + e + m + \sigma_2(i) \leqslant 2e + m \text{ et donc } \lambda_1(i) + \lambda_2(i) + u_i \leqslant e.$$

En sommant toutes les inégalités précédentes, on obtient $p(p+1) + \sum_{i=1}^{p} u_i \leqslant pe$ et, comme on a $p(p+1) + \sum_{i=1}^{p} u_i = pe$, cela signifie que les inégalités précédentes sont en fait des égalités. L'instance de RN-3DM a donc une solution, ce qui achève la démonstration de la NP-complétude de MaxTasks2-Dec(T, K). ■

3.4.2 Heuristique garantie

La section 3.3.2 nous a bien montré le lien qui pouvait exister entre couplage et permutation. Afin de déterminer les deux permutations σ_1 et σ_2, il est donc naturel d'utiliser le graphe biparti pondéré à $3p$ sommets représenté figure 3.5. Les sommets F_i correspondent à la première permutation, les sommets S_i correspondent à la seconde, et les sommets P_i correspondent aux processeurs. Deux permutations définissent sur un tel graphe deux couplages indépendants (un premier entre les F_i et les P_i et un second entre les S_i et les P_i) et réciproquement.

Néanmoins, si la correspondance entre couplage et permutation est toujours valable, à la différence du problème précédent, la fonction objectif que l'on cherche à maximiser ne se transpose pas aussi simplement. En effet, l'inégalité $\sigma_1(i)t^1_{\text{com}} + c_i w_i + \sigma_2(i)t^2_{\text{com}} \leqslant T$ se réécrit

$$c_i \leqslant \left\lfloor \frac{T - \sigma_1(i)t^1_{\text{com}} - \sigma_2(i)t^2_{\text{com}}}{w_i} \right\rfloor,$$

et il est donc nécessaire de connaître à la fois $\sigma_1(i)$ et $\sigma_2(i)$ pour calculer c_i. Nous utilisons donc l'approximation suivante :

$$c_i = \left\lfloor \frac{T/2 - \sigma_1(i)t^1_{\text{com}}}{w_i} \right\rfloor + \left\lfloor \frac{T/2 - \sigma_2(i)t^2_{\text{com}}}{w_i} \right\rfloor.$$

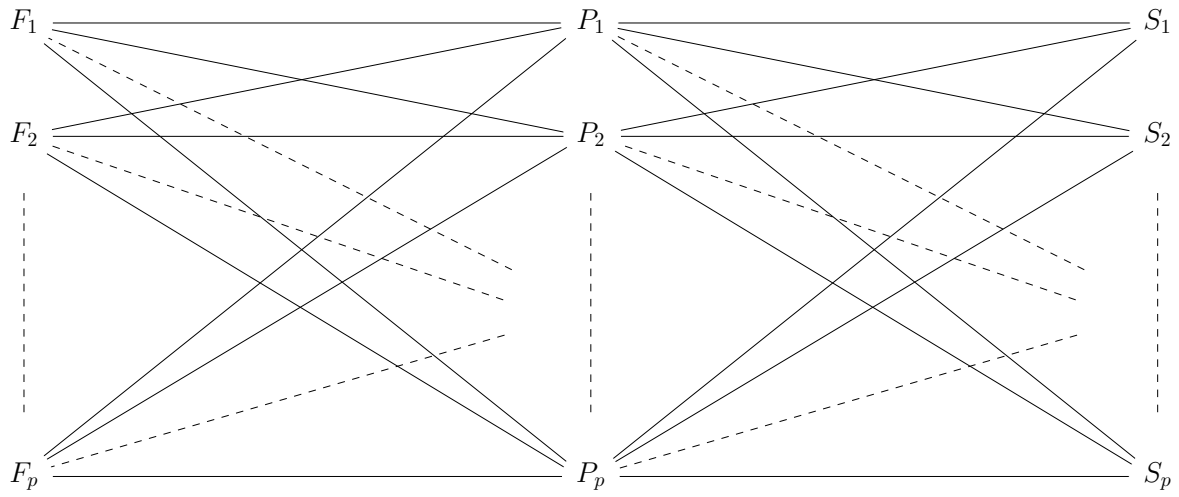


FIG. 3.5 – Graphe biparti représentant les communications initiales et les communications finales.

Cette approximation nous permet de pondérer les arêtes de la façon suivante :

$$W(P_i, F_j) = \left\lfloor \frac{T/2 - jt^1_{\text{com}}}{w_i} \right\rfloor \text{ et } W(P_i, S_k) = \left\lfloor \frac{T/2 - kt^2_{\text{com}}}{w_i} \right\rfloor$$

Il suffit alors de chercher deux couplages de poids maximum, ce qui, une fois de plus, est de l'ordre de $O(p^{\frac{5}{2}})$ puisqu'il suffit d'extraire les deux couplages de poids maximum dans chacun des graphes bipartis indépendamment l'un de l'autre.

Théorème 3.3. *L'approximation précédente conduit à une allocation de tâches qui diffère d'au plus p de la solution optimale.*

Démonstration. Remarquons d'abord que

$$\forall a, b : 0 \leqslant \lfloor a + b \rfloor - \lfloor a \rfloor - \lfloor b \rfloor \leqslant 1. \tag{3.1}$$

En conséquence, pour toute allocation $(\sigma_1, \sigma_2, \mathcal{C})$ construite en utilisant l'approximation précédente, nous avons

$$\forall i,\ c_i = \left\lfloor \frac{T/2 - \sigma_1(i)t^1_{\text{com}}}{w_i} \right\rfloor + \left\lfloor \frac{T/2 - \sigma_2(i)t^2_{\text{com}}}{w_i} \right\rfloor \leqslant \left\lfloor \frac{T - \sigma_1(i)t^1_{\text{com}} - \sigma_2(i)t^2_{\text{com}}}{w_i} \right\rfloor$$

et $(\sigma_1, \sigma_2, \mathcal{C})$ est donc bien une solution du problème initial.

Notons

$$C_{app}(\sigma_1, \sigma_2) = \sum_{i=1}^{p} \left\lfloor \frac{T/2 - \sigma_1(i)t^1_{\text{com}}}{w_i} \right\rfloor + \left\lfloor \frac{T/2 - \sigma_2(i)t^2_{\text{com}}}{w_i} \right\rfloor$$

le coût approché que nous utilisons dans notre algorithme et

$$\sigma_{app} = (\sigma_1^{(app)}, \sigma_2^{(app)}) = \text{argmax}_{\sigma_1,\sigma_2} C_{app}(\sigma_1, \sigma_2)$$

les permutations ainsi construites.

Notons

$$C_{opt}(\sigma_1, \sigma_2) = \sum_{i=1}^{p} \left\lfloor \frac{T - \sigma_1(i)t^1_{\text{com}} - \sigma_2(i)t^2_{\text{com}}}{w_i} \right\rfloor$$

le coût réel de deux permutations et notons

$$\sigma_{opt}(\sigma_1^{(opt)}, \sigma_2^{(opt)}) = \text{argmax}_{\sigma_1,\sigma_2} C_{opt}(\sigma_1, \sigma_2)$$

deux permutations solution optimale du problème initial.

Par définition, on a

$$C_{opt}(\sigma_{app}) \leqslant C_{opt}(\sigma_{opt}) \tag{3.2}$$

et

$$C_{app}(\sigma_{opt}) \leqslant C_{app}(\sigma_{app}). \tag{3.3}$$

En utilisant l'équation (3.1) on trouve que

$$\forall \sigma : C_{opt}(\sigma) - p \leqslant C_{app}(\sigma) \leqslant C_{opt}(\sigma) \tag{3.4}$$

et on en déduit que

$$\begin{aligned}
C_{opt}(\sigma_{opt}) - p &\leqslant C_{app}(\sigma_{opt}) && \big(\text{en utilisant (3.4)}\big)\\
&\leqslant C_{app}(\sigma_{app}) && \big(\text{en utilisant (3.3)}\big)\\
&\leqslant C_{opt}(\sigma_{app}) && \big(\text{en utilisant (3.4)}\big)\\
C_{opt}(\sigma_{opt}) - p \leqslant C_{opt}(\sigma_{app}) &\leqslant C_{opt}(\sigma_{opt}) && \big(\text{en utilisant (3.2)}\big),
\end{aligned}$$

ce qui signifie que notre approximation conduit à une allocation dont le nombre de tâches diffère au plus de p de l'optimal. ■

3.5 Allocation de tâches indépendantes sur une étoile

Dans cette section, on ne suppose plus que la plate-forme est un bus mais une étoile (voir figure 3.1). On suppose également que chacune des tâches à ses données propres et qu'une communication est donc nécessaire avant le traitement de chacune des tâches. Comme précédemment, les communications avec le maître sont exclusives. Nous pouvons donc définir notre nouveau problème d'optimisation :

Définition 3.7 (MasterSlave $(P_1(d_1, w_1), \ldots, P_p(d_p, w_p), T)$). *Étant donné une plate-forme maître-esclave de caractéristique $(d_1, w_1), \ldots, (d_p, w_p)$, et une borne T pour le temps d'exécution, quel est le nombre maximal de tâches pouvant être exécutées en temps T ?*

Une telle formulation n'est cependant pas très réaliste. En effet, on attend d'un algorithme résolvant ce problème, qu'il nous fournisse la liste des tâches traitées par chaque processeur ainsi que leur ordonnancement : même si les tâches ont le même temps d'exécution, elles correspondent certainement à des données (ou des fichiers) distinctes.

La taille de la description d'un tel ordonnancement est *a priori* proportionnelle à T (puisque le nombre de tâches pouvant être traitées augmente au mieux linéairement avec T). Si les données du problème se résument simplement à la liste des caractéristiques (d_i, w_i) des processeurs et à la valeur de T, la taille d'une telle description n'a donc aucune chance d'être polynomiale en la taille des données (qui est en $O(\log T)$ bits). On va donc faire figurer dans les données du problème la liste des tâches que possède initialement le maître et redéfinir ce problème d'optimisation de la façon suivante :

Définition 3.8 (MasterSlave-Schedule $(n, \mathcal{F}, p, \mathcal{W}, \mathcal{D}, T)$). *Étant donné :*
- *un ensemble $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ de n tâches indépendantes et de même taille ;*
- *un ensemble $\mathcal{W} = \{w_1, w_2, \ldots, w_p\}$ de p temps d'exécution ;*
- *un ensemble $\mathcal{D} = \{d_1, d_2, \ldots, d_p\}$ de p délais de communication ;*
- *une borne temporelle T ;*

déterminer un sous-ensemble de $\mathcal{F}$ de cardinal maximal, composé de tâches pouvant toutes être exécutées en T unités de temps sur une plate-forme maître-esclave de p processeurs P_i de paramètres w_i (calcul) et d_i (communication), pour $1 \leqslant i \leqslant p$.

On va montrer que ce problème peut être résolu par un algorithme glouton non trivial. On procède en trois étapes : (i) réduction au cas où chaque processeur exécute au plus une tâche ; (ii) preuve d'un lemme technique caractérisant un ordre d'exécution valide ; (iii) preuve que l'algorithme glouton proposé est optimal.

3.5.1 Réduction du problème

Dans cette section, nous montrons comment se ramener au cas où chaque processeur exécute au plus une tâche. Il suffit de transformer chaque processeur P_i par un ensemble de l_i+1 esclaves $P_{i,j}$, $0 \leqslant j \leqslant l_i = \lfloor \frac{T-w_i}{M_i} \rfloor$, où $M_i = \max(d_i, w_i)$. Considérons la figure 3.6 : chaque nouvel esclave a la même capacité de communication que P_i, mais $P_{i,j}$ exécute une tâche en $w_{i,j} = w_i + j.M_i$ unités de temps. Pour démontrer l'effectivité de cette transformation, nous introduisons le problème suivant :

Définition 3.9 (MasterSlave-Single $(P_1(d_1, w_1), \ldots, P_p(d_p, w_p), T)$). *Étant donné une borne T pour le temps d'exécution, une plate-forme maître-esclave de caractéristique $(d_1, w_1), \ldots, (d_p, w_p)$ et telle que chaque processeur exécute au plus une tâche, quel est le nombre maximal de tâches pouvant être exécutées en temps T ?*

Lemme 3.1. *MasterSlave $(P_1(d_1, w_1), \ldots, P_p(d_p, w_p), T)$ est égal à MasterSlave-Single $(\{P_{i,j}\}, T)$ avec $P_{i,j}$ de temps de communication d_i et de temps de calcul $j.M_i + w_i$.*

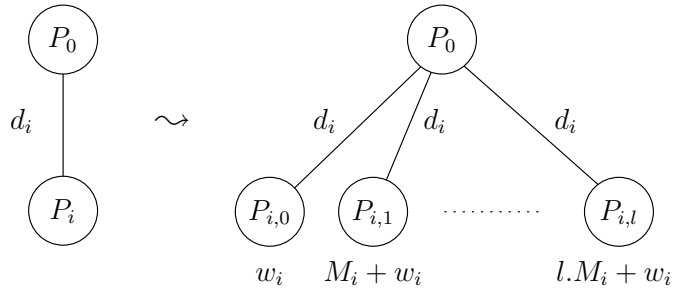


FIG. 3.6 – Remplacement d'un esclave P_i par un ensemble d'esclaves à tâche unique.

Démonstration. Intuitivement, l'idée de la réduction est que la $j^{\text{ème}}$ tâche exécutée par P_i dans l'instance originale du problème sera exécutée par $P_{i,j}$ dans la nouveau modèle. On va d'abord montrer que s'il est possible de traiter $n = \sum_i n_i$ tâches en temps inférieur à T sur la plate-forme originale alors il est également possible d'en traiter n en temps inférieur à T sur la seconde. On montre ensuite la réciproque :

i) Supposons donc que le processeur P_i traite n_i tâches et notons $t_1^{(i)}, t_2^{(i)}, \ldots, t_{n_i}^{(i)}$ (avec $t_j^{(i)} \leqslant t_{j+1}^{(i)}$) les dates de début d'émission de ces tâches vers P_i et $z_1^{(i)}, z_2^{(i)}, \ldots, z_{n_i}^{(i)}$ (avec $z_j^{(i)} \leqslant z_{j+1}^{(i)}$) les dates de début de calcul de ces tâches. On ne rallonge pas la durée ni la validité de l'ordonnancement en exécutant les tâches le plus tôt possible. On peut donc supposer sans perte de généralité que

$$\begin{cases} z_1^{(i)} & = t_1^{(i)} + d_i \\ z_2^{(i)} & = \max(z_1^{(i)} + w_i, t_2^{(i)} + d_i) \\ \vdots & \quad \vdots \\ z_{n_i}^{(i)} & = \max(z_{n_i-1}^{(i)} + w_i, t_{n_i}^{(i)} + d_i) \end{cases}$$

La date de terminaison de la dernière tâche sur P_i est donc $z_{n_i}^{(i)} + w_i$.
Cette tâche va être exécutée sur le processeur P_{i,n_i-1}. On va montrer qu'en servant ce processeur avant les $P_{i,j}$ pour $0 \leqslant j < n_i - 1$, il est possible de terminer le traitement de la tâche dans les temps. P_{i,n_i-1} est de caractéristique $(d_i, w_i + (n_i - 1)\max(d_i, w_i))$. La date de terminaison de cette tâche sur P_{i,n_i-1} est donc

$$t_1^{(i)} + d_i + w_i + (n_i - 1)\max(d_i, w_i),$$

et on souhaite montrer qu'il peut terminer de la traiter dans les temps, c'est-à-dire avant la date $z_{n_i}^{(i)} + w_i$. On a :

$$\begin{aligned} z_2^{(i)} &\geqslant z_1^{(i)} + w_i \geqslant t_1^{(i)} + d_i + w_i \\ z_3^{(i)} &\geqslant z_2^{(i)} + w_i \geqslant t_1^{(i)} + d_i + w_i + w_i \\ &\vdots \quad \vdots \\ z_{n_i}^{(i)} &\geqslant z_{n_i-1}^{(i)} + w_i \geqslant t_1^{(i)} + d_i + (n_i - 1)w_i \end{aligned}$$

et donc

$$z_{n_i}^{(i)} + w_i \geqslant t_1^{(i)} + d_i + (n_i)w_i = t_1^{(i)} + d_i + w_i + (n_i - 1)w_i.$$

De même, on a :

$$\begin{aligned} z_2^{(i)} &\geqslant t_2^{(i)} + d_i \geqslant t_1^{(i)} + 2d_i \\ z_3^{(i)} &\geqslant t_3^{(i)} + d_i \geqslant t_1^{(i)} + 3d_i \\ \vdots &\quad \vdots \\ z_{n_i}^{(i)} &\geqslant t_{n_i-1}^{(i)} + d_i \geqslant t_1^{(i)} + (n_i)d_i \end{aligned}$$

et donc

$$z_{n_i}^{(i)} + w_i \geqslant t_1^{(i)} + w_i + (n_i)d_i = t_1^{(i)} + w_i + d_i + (n_i - 1)d_i.$$

On a donc bien

$$z_{n_i}^{(i)} + w_i \geqslant t_1^{(i)} + w_i + d_i + (n_i - 1)M_i,$$

ce qui signifie que sur la nouvelle plate-forme cette tâche n'est pas terminée plus tard que sur la plate-forme initiale. En procédant de la même façon, on montre que pour tout $j \in [\![0, n_i - 1]\!]$, on a

$$z_{n_i-j}^{(i)} + w_i \geqslant t_j^{(i)} + w_i + d_i + (n_i - j - 1)M_i,$$

En envoyant la $j^{\text{ème}}$ tâche à $P_{i,j}$ à l'instant $t_j^{(i)}$, aucune de ces tâches ne termine donc plus tard sur la nouvelle plate-forme que sur la plate-forme initiale.

ii) Supposons maintenant que l'on soit capable de traiter n tâches. Supposons que n_i processeurs $P_{i,*}$ soient utilisés et notons $t_1^{(i)}, t_2^{(i)}, \ldots, t_{n_i}^{(i)}$ (avec $t_i^{(i)} \leqslant t_{i+1}^{(i)}$) les dates de réception de ces tâches sur chacun des différents $P_{i,*}$. On ne rallonge pas la durée de l'ordonnancement en commençant pas les processeurs les plus lents (en terme de puissance de calcul). On ne rallonge pas non plus la durée de l'ordonnancement en utilisant les processeurs les plus rapides en cas d'égalité de bande-passante. On peut donc supposer sans perte de généralité que :

$$\begin{aligned} t_1^{(i)} + d_i + w_i + (n_i - 1)M_i &\leqslant T \\ t_2^{(i)} + d_i + w_i + (n_i - 2)M_i &\leqslant T \\ &\vdots \\ t_{n_i}^{(i)} + d_i + w_i &\leqslant T \end{aligned} \tag{3.5}$$

On peut alors utiliser l'ordonnancement suivant pour la plate-forme initiale et vérifier qu'il est valide :
– le processeur P_i reçoit sa $j^{\text{ème}}$ tâche à l'instant $t_j^{(i)}$,

– le processeur P_i commence à traiter sa $j^{\text{ème}}$ tâche à l'instant $z_j^{(i)} = t_j^{(i)} + d_i + (j-1)M_i$.

Tout d'abord, les communications avec les esclaves étant équivalentes dans les deux modèles du point de vue du maître, comme il est possible d'effectuer les communications en les commençant aux instants $t_j^{(i)}$ dans le nouveau modèle, cela est également vrai dans le modèle original. Cet ordonnancement est donc valide dans le modèle originale et termine en un temps inférieur à T (grâce aux inégalités (3.5)). ■

Dans la suite, on supposera cette transformation effectuée et on se concentre donc sur le cas où chaque processeur est limité à l'exécution d'au plus une tâche.

3.5.2 Lemme technique

Lemme 3.2. *Soit $(P_{i_1}, \ldots, P_{i_k})$ une liste de k processeurs capables d'exécuter k tâches (une chacun) en T unités de temps, à l'aide d'un ordonnancement inconnu. Il est possible d'exécuter ces k tâches en triant les processeurs dans l'ordre des w_i décroissants et en communiquant du maître aux esclaves dans cet ordre.*

Démonstration. Pour simplifier les notations, soient $P_1, P_2, \ldots, P_k$ les processeurs triés dans l'ordre des w_i décroissants : $w_1 \geqslant w_2 \geqslant \ldots \geqslant w_k$. Soit $(P_{i_1}, \ldots, P_{i_k})$ l'ordre des communications émises par P_0 dans l'ordonnancement donné. On a donc les équations suivantes :

$$\begin{array}{lll}
(L_1) & d_{i_1} + w_{i_1} & \leqslant T \quad \text{première tâche} \\
(L_2) & d_{i_1} + d_{i_2} + w_{i_2} & \leqslant T \quad \text{deuxième tâche} \\
\vdots & \ddots & \vdots \qquad \vdots \\
(L_k) & d_{i_1} + d_{i_2} + \cdots + d_{i_k} + w_{i_k} & \leqslant T \quad \text{dernière tâche.}
\end{array}$$

Soient $j_1, \ldots, j_k$ les indices, dans l'ordonnancement donné, des tâches des esclaves considérés dans l'ordre trié des w_i décroissants : $i_{j_1} = 1$, $i_{j_2} = 2$, … et $i_{j_k} = k$. Par exemple, $i_{j_1} = 1$ signifie que le processeur P_1 exécute la $j_1{}^{\text{ème}}$tâche dans l'ordonnancement original. Démontrons par récurrence que les tâches peuvent bien être allouées dans le nouvel ordre $1, 2, \ldots, k$ correspondant aux w_i décroissants :

1. Pour la première tâche, on a $d_1 + w_1 \leqslant T$. En effet, dans l'équation L_{j_1} on a $d_{i_1} + \ldots + d_{i_{j_1-1}} + d_1 + w_1 \leqslant T$.
2. Pour la deuxième tâche, on a $d_1 + d_2 + w_2 \leqslant T$. En effet, il y a deux possibilités :
   – soit $j_2 > j_1$ et alors l'inégalité L_{j_2} comprend les termes $d_{i_{j_1}} + d_{i_{j_2}} + w_{i_{j_2}}$, et on a bien $d_1 + d_2 + w_2 \leqslant T$;
   – soit $j_2 < j_1$ et alors l'inégalité L_{j_1} comprend les termes $d_{i_{j_1}} + d_{i_{j_2}} + w_{i_{j_1}}$. Comme $w_2 \leqslant w_1$, on en déduit que $d_1 + d_2 + w_2 \leqslant T$.
3. Passons au cas général, et soit $i < k$. Supposons que pour tout $l \leqslant i$ on ait $d_1 + \ldots + d_l + w_l \leqslant T$. On veut montrer que $d_1 + \ldots + d_{i+1} + w_{i+1} \leqslant T$. Notons $m_i = \max(j_1, \ldots, j_i)$:

– soit $j_{i+1} > m_i$ et on regarde directement l'équation $L_{j_{i+1}}$. En effet, cette équation contient les termes $d_1, d_2, \ldots, d_i$ (car $j_{i+1} > m_i$) ainsi que le terme $d_{i+1} + w_{i+1}$. D'où $d_1 + \ldots + d_{i+1} + w_{i+1} \leqslant T$;
– soit $j_{i+1} < m_i$ et on regarde l'équation L_{m_i}. En effet, cette équation contient les termes $d_1, d_2, \ldots, d_{i+1}$ ainsi que $d_i + w_i$ et donc $d_1 + \ldots + d_{i+1} + w_i \leqslant T$. En utilisant le fait que $w_i \geqslant w_{i+1}$, on obtient bien $d_1 + \ldots + d_{i+1} + w_{i+1} \leqslant T$. ■

3.5.3 Algorithme glouton

Pour résoudre le problème MasterSlave-Single, on propose l'algorithme glouton suivant :

```
MASTER-SLAVE(P_1, ..., P_p, T)
 1: Trier les processeurs pour avoir d_1 ⩽ d_2 ... ⩽ d_p
 2: L ← ∅
 3: Pour i = 1..p :
 4:    Si L ∪ {P_i} est ordonnançable en temps inférieur à T Alors
 5:       L ← L ∪ {P_i}
 6: Renvoyer(L)
```

La condition de la ligne 4 peut être vérifiée en utilisant le résultat du lemme précédent : il suffit de trier la liste courante L selon les temps de cycle des esclaves puis de vérifier que toutes les inégalités sont satisfaites.

Théorème 3.4. *L'algorithme glouton précédent renvoie la solution optimale du problème MasterSlave-Single$(P_1, \ldots, P_p, T)$, c'est-à-dire un ensemble maximal de tâches pouvant être exécutées en temps T.*

Démonstration. On dira qu'un ensemble d'indices $\mathcal{I} = \{i_1, \ldots, i_k\}$ est ordonnançable si et seulement si il existe un ordonnancement valide en temps inférieur à T des processeurs $\{P_{i_1}, \ldots, P_{i_k}\}$.

Supposons que l'ensemble $\mathcal{I} = \{i_1, \ldots, i_k\}$ soit ordonnançable, avec (sans perte de généralité) $d_{i_1} \leqslant d_{i_2} \leqslant \ldots \leqslant d_{i_k}$. On va montrer que l'algorithme glouton renvoie au moins k valeurs, c'est-à-dire un ensemble ordonnançable $\mathcal{G} = \{g_1, \ldots, g_k\}$, ce qui terminera la preuve. On procède par récurrence :

1. Posons $\mathcal{I}_0 = \{i_1, \ldots, i_k\}$. L'ensemble $\mathcal{I}_0$ est ordonnançable. Tout sous-ensemble de $\mathcal{I}_0$ est aussi ordonnançable. En particulier, c'est le cas de chaque singleton $\{i_j\}$. Par construction de l'algorithme glouton, g_1 existe et $g_1 \leqslant \min(i_1, \ldots, i_k)$: en effet, l'algorithme glouton renvoie d'abord le plus petit indice d'une tâche ordonnançable. Montrons maintenant qu'il existe un ensemble ordonnançable $\mathcal{I}_1$ de k éléments qui contient g_1. Si $g_1 \in \mathcal{I}_0$, on a fini : $\mathcal{I}_1 = \mathcal{I}_0$. Sinon, considérons l'ordre $i_{j_1}, i_{j_2}, \ldots, i_{j_k}$ selon lequel $\mathcal{I}_0$ est ordonnancé. On a les inégalités suivantes :

$$\begin{array}{lll} (L_1) & d_{i_{j_1}} + w_{i_{j_1}} & \leqslant T \\ (L_2) & d_{i_{j_1}} + d_{i_{j_2}} + w_{i_{j_2}} & \leqslant T \\ \vdots & \ddots & \vdots \\ (L_k) & d_{i_{j_1}} + d_{i_{j_2}} + \cdots + d_{i_{j_k}} + w_{i_{j_k}} & \leqslant T. \end{array}$$

On remplace le premier indice ordonnancé i_{j_1} par g_1 pour construire I_1 : on pose $I_1 = I_0 \setminus \{i_{j_1}\} \cup \{g_1\}$. La première inégalité $d_{g_1} + w_{g_1} \leqslant T$ est satisfaite parce que g_1 est ordonnançable. Les autres inégalités (L_2) à (L_k) restent satisfaites parce que $d_{g_1} \leqslant d_{i_{j_1}}$. Ainsi, $\mathcal{I}_1 = \{g_1, i_{j_2}, \ldots, i_{j_k}\}$ est ordonnançable.

2. Il existe donc un ensemble ordonnançable $\mathcal{I}_1 = \{g_1, i'_1, \ldots, i'_{k-1}\}$. Pour tout j, $1 \leqslant j \leqslant k_1$, la paire $\{g_1, i'_j\}$ est ordonnançable, ce qui établit que l'algorithme glouton renvoie bien un indice g_2 après avoir renvoyé g_1. De plus, $g_2 \leqslant \min(i'_1, \ldots, i'_{k-1})$. On va montrer qu'il existe un ensemble ordonnançable $\mathcal{I}_2$ de k éléments qui contient g_1 et g_2. Si $g_2 \in \{i'_1, \ldots, i'_{k-1}\}$, posons $\mathcal{I}_2 = \mathcal{I}_1$. Sinon, considérons l'ordre dans lequel $\mathcal{I}_1$ est ordonnancé. Il n'y a pas de raison *a priori* que g_1 soit ordonnancé en premier dans $\mathcal{I}_1$, d'où les deux cas :
   – g_1 est effectivement la première tâche de $\mathcal{I}_1$ communiquée par le maître P_0. Soit f l'indice de la seconde tâche de $\mathcal{I}_1$ communiquée par P_0. Comme la paire $\{g_1, g_2\}$ est ordonnançable, on peut remplacer f par g_2 : les deux premières inégalités (L_1) et (L_2) sont satisfaites (il faudra peut-être échanger g_1 avec g_2, dans tous les cas cette paire peut être ordonnancée), et les $k-2$ inégalités suivantes restent satisfaites parce que $d_{g_2} \leqslant d_f$;
   – l'indice f de la première tâche de $\mathcal{I}_1$ communiquée par P_0 est distinct de g_1. On remplace alors f par g_2 : la première inégalité (L_1) est satisfaite parce que $\{g_2\}$ est ordonnançable, et les $k-1$ inégalités suivantes restent satisfaites parce que $d_{g_2} \leqslant d_f$.

   On en conclut que $\mathcal{I}_2$ est ordonnançable.

3. Supposons avoir un ensemble ordonnançable $\mathcal{I}_{j-1} = \{g_1, \ldots, g_{j-1}, i'_1, \ldots, i'_{k-j+1}\}$. Pour tout l, $1 \leqslant l \leqslant k-j+1$, le sous-ensemble $\{g_1, \ldots, g_{j-1}, i'_l\}$ est ordonnançable, donc l'algorithme glouton renvoie bien un indice g_j après avoir renvoyé $g_1, \ldots, g_{j-1}$. De plus, $g_j \leqslant \min(i'_1, \ldots, i'_{k-j+1})$. On va montrer qu'il existe un ensemble ordonnançable $\mathcal{I}_j$ de k éléments qui contient $\{g_1, g_2, \ldots, g_j\}$. Si $g_j \in \{i'_1, \ldots, i'_{k-j+1}\}$, posons $\mathcal{I}_j = \mathcal{I}_{j-1}$. Sinon, considérons l'ordre dans lequel $\mathcal{I}_{j-1}$ est ordonnancé. On sélectionne la première tâche exécutée f qui soit différente des indices g_1, g_2, …, g_{j-1} renvoyés par l'algorithme glouton. On remplace f par g_j pour construire $\mathcal{I}'_j$. Montrons que $\mathcal{I}_j$ est ordonnançable. En effet, pour l'ordre dans lequel $\mathcal{I}_{j-1}$ est ordonnancé, soit l un indice tel que les l premières inégalités (L_1) à (L_l) fassent intervenir des indices g_{i_1}, g_{i_2}, …, g_{i_l} renvoyés par l'algorithme glouton, tandis que l'inégalité (L_{l+1}) fait intervenir l'indice f :

   a) $\{g_{i_1}, \ldots, g_{i_l}, g_j\}$ est ordonnançable en tant que sous-ensemble de $\{g_1, \ldots, g_j\}$: un ordre valide de ces $l+1$ tâches conduit aux $l+1$ premières inégalités ;

   b) les inégalités (L_{l+2}) à (L_k) restent satisfaites parce que $d_{g_j} \leqslant d_{i'_f}$. ■

Nous pouvons donc établir le résultat suivant :

Théorème 3.5. *MasterSlave-Schedule* $(n, \mathcal{F}, p, \mathcal{W}, \mathcal{D}, T)$ *peut être résolu en temps polynomial.*

Démonstration. Après la réduction à des processeurs effectuant au plus une tâche, le nombre d'esclaves est borné par $p \times n$, quantité polynomiale en la taille des données. En effet, $\mathcal{F}$ est de taille au moins $O(n)$, tandis que $\mathcal{W}$ et $\mathcal{D}$ sont de taille au moins $O(p)$.

La complexité de l'algorithme glouton est quadratique en le nombre d'esclaves : chaque tâche n'est considérée qu'une fois, et le test de son insertion se réalise aisément en temps linéaire en le nombre de tâches déjà sélectionnées. Il suffit de maintenir la liste des tâches déjà sélectionnées avec leur temps de fin d'exécution. Quand une nouvelle tâche F_i est insérée dans la liste, on incrémente de d_i le temps de fin d'exécution de toutes les tâches F_j de la liste dont le paramètre w_j est inférieur à w_i, en vérifiant au vol que les nouveaux temps d'exécution ne dépassent pas T.

Au final, la complexité totale est bornée par $O(n^2p^2)$, ce qui établit le résultat. ■

3.6 Conclusion

Dans ce chapitre, nous avons revisité le paradigme maître-esclave dans un contexte hétérogène. Nous nous sommes plus particulièrement intéressés au problème de la maximisation du nombre de tâches traitées en un temps T pour différents modèles d'applications et de plates-formes.

Dans le cas où le réseau d'interconnexion est un bus, c'est-à-dire quand le temps de communication du maître vers les esclave est uniforme, le problème de l'allocation de tâches identiques nécessitant une seule communication initiale est polynomial. Il devient NP-complet si l'on suppose qu'une communication finale supplémentaire est nécessaire.

Dans le cas où le réseau d'interconnexion est une étoile, c'est-à-dire quand l'hétérogénéité est présente à la fois dans les temps de calcul et dans les temps de communication, le problème de l'allocation de tâches identiques ayant leur données propres est polynomial. Dutot a étendu ce résultat au cas où le graphe d'interconnexion de plate-forme est une chaîne ou une pieuvre [50]. Il propose un algorithme polynomial pour résoudre le problème de l'ordonnancement en temps minimal ($\mathsf{MinTime}(C)$) de n tâches sur un ensemble de p processeurs interconnectés en chaîne. Cet algorithme construit l'ordonnancement «à l'envers», en partant de la dernière tâche traitée, et est de complexité $O(np^2)$. Il étend également ce résultat au cas où les processeurs sont interconnectés en pieuvre. Dutot a également démontré la NP-complétude de ce problème quand la plate-forme est un arbre [51].

La difficulté de ces problèmes provient, à la fois du caractère hétérogène de la plate-forme, du fait que la quantité de travail à effectuer est finie et discrète et que l'on cherche à traiter ces données en un temps minimal. Nous verrons dans les chapitres suivants que cette métrique et que ce modèle ne sont pas forcément adaptés à de telles applications et

à de telles plates-formes. En effet, en raison du temps de déploiement et de la difficulté de mise en œuvre sur de telles plates-formes, il n'est rentable de déployer que de grosses applications. Ces dernières exhibent souvent une certaine régularité dont il est possible de tirer parti. Notamment, en s'intéressant, non pas au *makespan* mais en se plaçant en régime permanent (situation qui se produit naturellement dès que le nombre de tâches à traiter devient important), nous verrons que les problèmes de ce chapitre sont beaucoup plus simples à traiter et qu'il est possible d'en déduire des ordonnancements dont le temps d'exécution est asymptotiquement optimal.

Chapitre 4

Ordonnancement de tâches indépendantes en régime permanent

4.1 Introduction

Dans ce chapitre, nous nous intéressons une fois de plus au paradigme maître-esclave sur une plate-forme hétérogène. Comme dans le chapitre 3, nous supposons disposer d'un ensemble de tâches indépendantes et de même taille. Cependant, à la différence de la section 3.5, nous supposons que le nombre de tâches à traiter est grand (sinon, étant donné le temps de déploiement et la difficulté de mise en œuvre sur de telles plates-formes, quel intérêt y aurait-il à utiliser une plate-forme aussi complexe ?), ce qui nous permet de nous placer en régime permanent et de considérer des plates-formes plus générales. La plate-forme est donc modélisée par un graphe, appelé *graphe de plate-forme*, où chaque nœud représente une ressource (un processeur, un cluster, un routeur,etc.) capable de calculer et/ou de communiquer avec ses voisins.

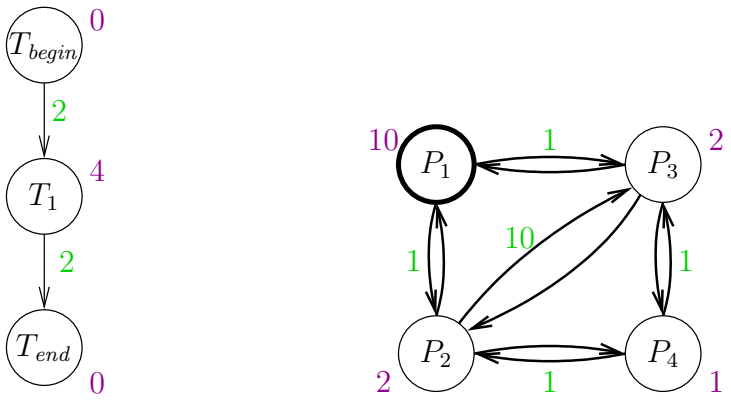
On suppose qu'un des nœuds, appelé maître, joue un rôle particulier et dispose initialement (ou génère au fur et à mesure) d'une grande quantité de tâches à traiter. Une tâche est constituée de fichiers spécifiques qui contiennent toutes les données nécessaires à son exécution. La question pour le maître est donc de décider quelles sont les tâches qu'il va exécuter lui même et quelles sont celles dont il va déléguer le traitement à ses voisins. En raison de l'hétérogénéité de la plate-forme le nombre de tâches envoyées à chaque voisin va être différent. Certains vont peut-être même ne rien recevoir du tout mais les autres vont, à leur tour, être confrontés au même problème.

Comme nous l'avons vu dans le chapitre 3, ces problèmes sont difficiles à traiter quand la métrique choisie est le temps d'exécution, ou *makespan*. Mais quand le nombre de tâches devient important, il est plus naturel de s'intéresser à l'optimisation du régime permanent. On cherche alors à déterminer pour chaque processeur la fraction de temps passée à calculer, la fraction de temps passée à recevoir des données et la fraction de temps passée à envoyer des données, de façon à ce que le nombre moyen de tâches traitées par unité de temps soit maximal.

Les résultats principaux de ce chapitre sont les suivants :

- Nous sommes capables de déterminer le régime permanent optimal pour des plates-formes quelconques, pouvant même contenir des chemins multiples pour aller d'un point à un autre (ce qui est le cas dans Internet). Ce régime permanent est calculé en formulant un programme linéaire, qui permet de prendre très simplement en compte la présence de plusieurs maîtres ou l'hétérogénéité des capacités de communications des différents processeurs. Nous donnons également une description polynomiale de l'ordonnancement permettant d'atteindre ce régime permanent.
- Dans le cas où la plate-forme d'interconnexion est une arborescence, c'est-à-dire quand le réseau se représente par un arbre orienté enraciné par le maître, nous sommes capables de déterminer une forme close caractérisant le régime permanent qui peut alors être calculé simplement en effectuant un parcours en profondeur de l'arbre d'interconnexion. De plus, ce régime permanent est obtenu en utilisant une stratégie orientée bande-passante (*bandwidth-centric*) : si la bande passante est suffisante, alors tous les enfants sont alimentés et dans le cas contraire les tâches doivent être d'abord envoyées aux enfants ayant des temps de communication suffisamment rapides, et ce par ordre de bande passante décroissante. Contrairement à ce que l'intuition pourrait suggérer au premier abord, il vaut mieux déléguer ses tâches le plus rapidement possible plutôt que de chercher à les envoyer aux enfants qui ont les temps de calcul les plus rapides. L'avantage d'une telle approche est qu'elle se prête bien à des protocoles de distributions non centralisés s'appuyant uniquement sur une vision locale de la plate-forme.
- Enfin, nous comparons la capacité des plates-formes représentées par une arborescence avec celle des plates-formes quelconques. Étant donné une topologie réseau pouvant inclure des cycles, comment extraire la «meilleure» arborescence couvrante, c'est-à-dire l'arbre couvrant enraciné par le maître et permettant de traiter le plus de tâches possible en régime permanent ? Nous montrons que ce problème est NP-complet et inapproximable, c'est-à-dire qu'il existe des plates-formes hétérogènes dont les performances de la meilleure arborescence couvrante peuvent être aussi éloignées que l'on veut de celles de la plate-forme complète.

Ce chapitre est donc organisé de la façon suivante. En section 4.2, nous précisons le modèle et les notations que nous utilisons dans ce chapitre. En section 4.3, nous établissons les équations qui décrivent le régime permanent et montrons comment les résoudre à l'aide d'un programme linéaire. Nous montrons ensuite comment en déduire un ordonnancement qui réalise le débit ainsi calculé. En section 4.4, nous montrons comment les équations se simplifient dans le cas d'une arborescence et fournissons des formes closes ainsi qu'un algorithme pour calculer le débit optimal. La section 4.5 est plus théorique. On y montre l'optimalité asymptotique de l'ordonnancement proposé en section 4.3.5. Plus précisément, nous montrons qu'en temps T un ordonnancement optimal n'exécute qu'un nombre borné (indépendant de T) de tâches de plus que notre ordonnancement. Elle porte également sur le problème de l'extraction d'une arborescence couvrante et contient la NP-complétude et l'inapproximabilité de ce problème. Nous concluons en section 4.6.



(a) Graphe de tâches

(b) Graphe de plate-forme : Les fichiers d'entrée $e_{begin,1}$ sont placé sur P_1 à l'origine et les fichiers de sortie $e_{1,end}$ doivent être rassemblés sur P_1

FIG. 4.1 – Exemple simple de modélisation d'une application constituée de tâches indépendantes à exécuter sur une plate-forme hétérogène.

4.2 Modèles

Cette section présente les différentes notations que nous allons utiliser jusqu'à la fin du chapitre ainsi que des notions importantes sur l'ordonnancement cyclique. Les sections 4.2.1 à 4.2.4 présentent la différents paramètre de notre problème, à savoir le graphe d'application générique que l'on souhaite ordonnancer, le graphe de plate-forme et comment ils sont liés l'un à l'autre. La section 4.2.5 formalise les contraintes que doit respecter un ordonnancement valide (contraintes de resources, contraintes 1-port, contraintes de précédences). Ces contraintes nous permettent d'adapter à notre cadre de travail les notions d'ordonnancement cycliques et périodiques présentées par Alix Munier dans [86]. On définit donc formellement en section 4.2.6 ces deux types d'ordonnancement ainsi qu'une mesure de leur efficacité. Intuitivement, un ordonnancement cyclique est l'ordonnancement d'une infinité de graphes de tâches identiques et la mesure de son efficacité est le nombre moyen de graphes traités par unité de temps. Un ordonnancement K-périodique de période T_p est un ordonnancement cyclique obtenu en répétant périodiquement (toutes les T_p unités de temps) un ordonnancement des K premières copies du graphe de tâches. La section 4.2.7 présente une propriété fondamentale des ordonnancement périodiques : il est possible d'oublier en partie les contraintes de précédences et de se concentrer sur les contraintes de resources et les contraintes 1-port pour construire un ordonnancement périodique valide.

4.2.1 L'application

Notre application est constituée de n problèmes $\mathcal{P}^{(1)}, \mathcal{P}^{(2)}, \ldots, \mathcal{P}^{(n)}$ à résoudre avec n grand. Chaque problème correspond à une copie $G_A^{(m)} = (V_A^{(m)}, E_A^{(m)})$ d'un même graphe de tâches $G_A = (V_A, E_A)$. Même si ce chapitre porte sur l'exécution de tâches simples indépendantes, on utilise un graphe de tâches pour pouvoir modéliser plus simplement différentes situations. En effet, on suppose que ce graphe est connexe, sans cycle et qu'il existe une seule tâche sans prédécesseur, appelée T_{begin}, et une seule tâche sans successeur, appelée T_{end}. Ces tâches permettent de modéliser les fichiers d'entrées et de sorties. Dans ce chapitre, on suppose que le graphe de tâches est comme celui représenté figure 4.1(a).

4.2.2 La plate-forme

La plate-forme est représentée à l'aide d'un graphe orienté $G_P = (V_P, E_P)$ appelé *graphe de plate-forme* comme celui de la figure 4.1(b). Ce graphe est composé de p nœuds $P_1, P_2, \ldots, P_p$ qui représentent les différents processeurs. Chaque arête $P_i \rightarrow P_j$ représente un lien de communication et est étiquetée par une valeur $c_{i,j}$ représentant le temps nécessaire à l'envoi d'un message de taille unitaire entre P_i et P_j. S'il n'est pas possible de communiquer directement entre P_i et P_j, on définit $c_{i,j}$ comme étant égal à $+\infty$. En utilisant cette convention, on peut supposer que le graphe est virtuellement complet. Notons enfin que ce graphe représente l'architecture physique de la plate-forme et que la matrice associée n'est donc pas nécessairement une matrice de distance (c'est-à-dire vérifiant $c_{i,j} \leqslant c_{i,k} + c_{k,j}$).

Le modèle de communication que nous utilisons est le suivant : les processeurs sont capables de recouvrir leurs communications avec leurs calculs (*full overlap*), ils ne sont capables de recevoir des données que d'un seul voisin à la fois (modèle 1 port en entrée), et de n'envoyer des données qu'à un seul voisin à la fois (modèle 1 port en sortie). À chaque étape, au plus une communication entrante et une communication sortante peuvent donc être mises en œuvre. La section 4.5.2 explique comment prendre d'autres modèles en compte. Nous préciserons en section 4.2.5 comment ces contraintes se formalisent en terme de contraintes d'ordonnancement.

4.2.3 Temps d'exécution

Il faut $w_{i,k}$ unités de temps au processeur P_i pour traiter la tâche T_k. Ce modèle est très général puisque les vitesses des processeurs sont hétérogènes et que les temps de traitement d'une tâche d'un processeur à l'autre ne sont pas corrélés (*inconsistants* pour reprendre la terminologie utilisée par [25]). Bien sûr, on peut toujours simplifier le modèle en supposant que $w_{i,k} = w_i \times \delta_k$, où w_i est l'inverse de la vitesse relative du processeur P_i, et δ_k le poids de la tâche T_k, mais cette hypothèse n'est pas nécessaire dans nos calculs. Enfin, notons que la modélisation des routeurs se fait simplement en considérant des nœuds dont la puissance de calcul est nulle, c'est-à-dire tels que $w_{i,k} = +\infty$ pour tout T_k.

Les tâches T_{begin} sont fictives et nous posons donc $w_{i,begin} = 0$ pour chaque processeur P_i qui possède des fichiers d'entrée (ou $w_{i,begin} \neq 0$ s'il les génère) et $w_{i,begin} = +\infty$ pour les autres. T_{end}, quant à lui peut être utilisé pour modéliser deux situations différentes : soit les résultats (les fichiers de sortie) n'ont pas besoin d'être rassemblés à un endroit particulier et peuvent rester sur place, soit ils doivent être rapatriés sur un processeur particulier P_{dest} (pour réaliser une visualisation ou un post-traitement par exemple). Dans la première situation (les fichiers de sortie restent sur place), aucun fichier de sortie ne doit circuler d'un processeur à l'autre et donc on pose $w_{i,end} = 0$. Dans le cas contraire, où les fichiers doivent être rassemblés sur le processeur P_{dest} alors on définit $w_{dest,end}$ comme étant égal à 0 et $w_{i,end} = +\infty$ pour les autres processeurs, ce qui force les fichiers de type $e_{k,end}$ à être envoyés jusqu'à P_{dest}.

4.2.4 Temps de communication

Chaque arête $e_{k,l} : T_k \rightarrow T_l$ du graphe de tâche est pondérée par un coût de communication $data_{k,l}$ représentant la quantité de données créées par T_k et utilisées par T_l. Le temps nécessaire au transfert d'un message de taille unitaire du processeur P_i au processeur P_j étant égal à $c_{i,j}$, le temps nécessaire au transfert d'un fichier de type $e_{k,l}$ de P_i à P_j est égal à $data_{k,l} \times c_{i,j}$ (à la différence des processeurs, introduire une notion d'affinité à l'aide d'un $X(i,j,k,l)$ n'aurait pas vraiment de sens ici).

4.2.5 Formalisation de la notion d'ordonnancement

Définition 4.1 (Allocation). *Une allocation valide est composée d'une application $\pi : V_A \mapsto V_P$ et d'une application $\sigma : E_A \mapsto \{\text{chemin dans } G_P\}$ vérifiant pour tout $e_{k,l} : T_k \rightarrow T_l$:*

$$\sigma(e_{k,l}) = (P_{i_1}, P_{i_2}, \ldots, P_{i_p}) \text{ avec } \begin{cases} P_{i_1} = \pi(T_k), P_{i_p} = \pi(T_l) \text{ et} \\ \left(P_{i_j} \rightarrow P_{i_{j+1}}\right) \in E_P \text{ pour tout } j \in [\![1, p-1]\!] \end{cases}.$$

Définition 4.2 (Ordonnancement). *Un ordonnancement valide associé à une allocation valide (π, σ) est composée d'une application $t_\pi : V_A \mapsto \mathbb{R}$ et d'une application $t_\sigma : E_A \times E_P \mapsto \mathbb{R}$ vérifiant les contraintes suivantes :*

– ***précédence :*** *Pour tout $e_{k,l} : T_k \rightarrow T_l$, si $\sigma(e_{k,l}) = (P_{i_1}, P_{i_2}, \ldots, P_{i_p})$ alors*

$$\begin{aligned} t_\pi(T_k) + w_{i_1,k} &\leqslant t_\sigma(e_{k,l}, P_{i_1} \rightarrow P_{i_2}) \\ t_\sigma(e_{k,l}, P_{i_1} \rightarrow P_{i_2}) + data_{k,l} \times c_{i_1,i_2} &\leqslant t_\sigma(e_{k,l}, P_{i_2} \rightarrow P_{i_3}) \\ &\vdots \\ t_\sigma(e_{k,l}, P_{i_{p-1}} \rightarrow P_{i_p}) + data_{k,l} \times c_{i_{p-1},i_p} &\leqslant t_\pi(T_l) \end{aligned}$$

– ***ressources de calcul :*** *Un processeur ne peut traiter qu'une seule tâche à la fois. Pour tout $T_k \neq T_l$ on a donc :*

$$\pi(T_k) = \pi(T_l) \Rightarrow [t_\pi(T_k), t_\pi(T_k) + w_{\pi(T_k),k}[\cap [t_\pi(T_l), t_\pi(T_l) + w_{\pi(T_l),l}[= \emptyset$$

– ***ressources de communications :****Il ne peut circuler qu'un seul fichier à la fois entre P_i et P_j. Pour tout $e_{k_1,l_1} \neq e_{k_2,l_2} \in E_A$ et tout $P_i \to P_j$ on a donc :*

$$[t_\sigma(e_{k_1,l_1}, P_i \to P_j), t_\sigma(e_{k_1,l_1}, P_i \to P_j) + data_{k_1,l_1} \times c_{i,j}[\cap [t_\sigma(e_{k_2,l_2}, P_i \to P_j), t_\sigma(e_{k_2,l_2}, P_i \to P_j) + data_{k_2,l_2} \times c_{i,j}[= \emptyset$$

– ***1-port en entrée*** *Un processeur ne peut envoyer de fichier qu'à un seul processeur à la fois. Pour tout $e_{k_1,l_1}, e_{k_2,l_2} \in E_A$ et tout $(P_{i_1} \to P_j) \neq (P_{i_2} \to P_j)$ on a donc :*

$$[t_\sigma(e_{k_1,l_1}, P_{i_1} \to P_j), t_\sigma(e_{k_1,l_1}, P_{i_1} \to P_j) + data_{k_1,l_1} \times c_{i_1,j}[\cap [t_\sigma(e_{k_2,l_2}, P_{i_2} \to P_j), t_\sigma(e_{k_2,l_2}, P_{i_2} \to P_j) + data_{k_2,l_2} \times c_{i_2,j}[= \emptyset$$

– ***1-port en sortie*** *Un processeur ne peut recevoir de données que d'un seul processeur à la fois. Pour tout $e_{k_1,l_1}, e_{k_2,l_2} \in E_A$ et tout $(P_i \to P_{j_1}) \neq (P_i \to P_{j_2})$ on a donc :*

$$[t_\sigma(e_{k_1,l_1}, P_i \to P_{j_1}), t_\sigma(e_{k_1,l_1}, P_i \to P_{j_1}) + data_{k_1,l_1} \times c_{i,j_1}[\cap [t_\sigma(e_{k_2,l_2}, P_i \to P_{j_2}), t_\sigma(e_{k_2,l_2}, P_i \to P_{j_2}) + data_{k_2,l_2} \times c_{i,j_2}[= \emptyset$$

Définition 4.3 (Durée d'un ordonnancement). *La durée d'un ordonnancement valide (t_π, t_σ) est définie par*

$$\max_{T_k \in V_A} \left(t_\pi(T_k) + w_{\pi(T_k),k}\right) - \min_{T_k \in V_A} t_\pi(T_k)$$

4.2.6 Formalisation de la notion d'ordonnancement cyclique

Définition 4.4 (Graphe développé). *Soit $G_A = (V_A, E_A)$ un graphe de tâches et S un ensemble. On notera :*

$$\begin{aligned} V_A \otimes S &= V_A \times S \\ E_A \otimes S &= \left\{ \left((T_k, n), (T_l, n)\right) \middle| e_{k,l} : T_k \to T_l \in E_A \text{ et } n \in S \right\} \\ G_A \otimes S &= (V_A \otimes S, E_A \otimes S) \end{aligned}$$

$G_A \otimes S$ est appelé graphe développé de G_A selon S.

Définition 4.5 (Ordonnancement cyclique). *Un ordonnancement cyclique d'un graphe de tâches G_A sur une plate-forme G_P est un ordonnancement valide de $G_A \otimes \mathbb{N}$ sur G_P.*

Définition 4.6 (Durée d'un ordonnancement cyclique). *La durée des N premières tâches d'un ordonnancement cyclique est définie par :*

$$D_N = \max_{(T_k,n)\in V_A\times[\![1,N]\!]} \left(t_\pi(T_k,n) + w_{\pi(T_k,n),k}\right) - \min_{(T_k,n)\in V_A\times[\![1,N]\!]} t_\pi(T_k,n)$$

Définition 4.7 (Suite K-périodique). *Une suite u est dite K-périodique si elle est croissante et s'il existe un entier n_0 et un réel T_p strictement positif tels que*

$$\forall n \geqslant n_0 : u_{n+K} = u_n + T_p$$

K est appelé facteur de périodicité et T_p représente la période de la suite. $\frac{K}{T_p}$ est alors la fréquence de la suite. Une suite telle que $n_0 = 0$ est dite strictement *K-périodique*

Définition 4.8 (Ordonnancement K-périodique). *On identifie $f(x,n)$ et $f(x)(n)$. Un ordonnancement cyclique (t_π, t_σ) est dit K-périodique de période T_p si pour tout $T_k \in V_A$, $t_\pi(T_k)$ est strictement K-périodique de période T_p et si pour tout $e_{k,l} \in E_A$ et tout $P_i \to P_j$, $t_\sigma(e_{k,l}, P_i \to P_j)$ est strictement K-périodique de période T_p.*

Définition 4.9 (Fréquence d'un ordonnancement cyclique). *La fréquence (ou débit) d'un ordonnancement est, sous réserve de son existence, la limite*

$$\lim_{N\to\infty} \frac{N}{D_N}$$

Remarque. *Un ordonnancement K-périodique de période T_p est entièrement caractérisé par les K premières valeurs des t_π (T_k) et des t_σ ($e_{k,l}$,$P_i \to P_j$).*

La fréquence d'un ordonnancement K-périodique de période T_p est égal à $\frac{K}{T_p}$.

4.2.7 Propriétés fondamentales des ordonnancements périodiques

Définition 4.10 (K-motif de longueur T_p). *Un K-motif de longueur T_p est une allocation (π,σ) de $G_A \otimes [\![1,K]\!]$ sur G_P et deux applications $\tilde{t}_\pi$ de $V_A \otimes [\![1,K]\!]$ dans $[0,T_p[$ et $\tilde{t}_\sigma$ de $(E_A \otimes [\![1,K]\!]) \times E_P$ dans $[0,T_p[$ vérifiant les contraintes de ressources et les contraintes 1-port modulo T_p.*

Définition 4.11 (M_{K,T_p}). *On note M_{K,T_p} l'application canonique de l'ensemble des ordonnancement K-périodiques de période T_p dans l'ensemble des K-motifs de longueur T_p définie par $M_{K,T_p}(t_\pi,t_\sigma) = (\tilde{t}_\pi,\tilde{t}_\sigma)$ avec :*

$$\forall T_k \in V_A, \forall n \in [\![1,K]\!] : \tilde{t}_\pi(T_k,n) = t_\pi(T_k,n) \mod T_p$$
$$\forall e_{k,l} \in E_A, \forall P_i \to P_j \in E_P, \forall n \in [\![1,K]\!] : \tilde{t}_\sigma(e_{k,l},n,P_i \to P_j) = t_\sigma(e_{k,l},n,P_i \to P_j) \mod T_p$$

Lemme 4.1 (M_{1,T_p} est surjective). *Soit $T_p \in \mathbb{R}_+^*$. Soit (π,σ), une allocation de G_A sur G_P. Étant donné $\tilde{t}_\pi$ de V_A dans $[0,T_p[$ et $\tilde{t}_\sigma$ de $E_A \times E_P$ dans $[0,T_p[$ vérifiant les contraintes de ressources et les contraintes 1-port modulo T_p, il existe un ordonnancement 1-périodique*

(t_π, t_σ) de G_A sur G_P (et donc vérifiant également les contraintes de précédence), de période T_p, tel que

$$\forall T_k \in V_A, \forall n \in \mathbb{N} : t_\pi(T_k, n) = \tilde{t}_\pi(T_k) \mod T_p$$
$$\forall e_{k,l} \in E_A, \forall P_i \to P_j \in E_P, \forall n \in \mathbb{N} : t_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma(e_{k,l}, P_i \to P_j) \mod T_p$$

Démonstration. Cherchons t_π et t_σ sous la forme

$$t_\pi(T_k, n) = \tilde{t}_\pi(T_k) + (n + \Delta_\pi(T_k))T_p$$
$$t_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma(e_{k,l}, P_i \to P_j) + (n + \Delta_\sigma(e_{k,l}, P_i \to P_j))T_p,$$

avec Δ_σ et Δ_π à valeurs dans $\mathbb{Z}$, et donnons des conditions nécessaires et suffisantes sur Δ_σ et Δ_π pour que (t_π, t_σ) soit bien un ordonnancement 1-périodique de G_A sur G_P.

– **Contraintes de précédence :** Soit $e_{k,l} \in E_A$. Notons $\sigma(e_{k,l}) = (P_{i_1}, P_{i_2}, \ldots, P_{i_p})$.

$$\forall n \in \mathbb{N} : t_\sigma((e_{k,l}, n), P_{i_{p-1}} \to P_{i_p}) + data_{k,l} \times c_{i_{p-1},i_p} \leqslant t_\pi(T_l, n)$$
$$\Leftrightarrow$$
$$\tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) + \Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p})T_p + data_{k,l} \times c_{i_{p-1},i_p} \leqslant \tilde{t}_\pi(T_l) + \Delta_\pi(T_l)T_p$$
$$\Leftrightarrow$$
$$\tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) + \Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p})T_p + data_{k,l} \times c_{i_{p-1},i_p} \leqslant \tilde{t}_\pi(T_l) + \Delta_\pi(T_l)T_p$$
$$\Leftrightarrow$$
$$\Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) \leqslant \left\lfloor \frac{\tilde{t}_\pi(T_l) - \tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) - data_{k,l} \times c_{i_{p-1},i_p}}{T_p} \right\rfloor + \Delta_\pi(T_l)$$

De même on a

$$\forall n \in \mathbb{N} : t_\sigma((e_{k,l}, n), P_{i_1} \to P_{i_2}) + data_{k,l} \times c_{i_1,i_2} \leqslant t_\sigma((e_{k,l}, n), P_{i_2} \to P_{i_3})$$
$$\Leftrightarrow$$
$$\Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) \leqslant \left\lfloor \frac{\tilde{t}_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}) - \tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - data_{k,l} \times c_{i_1,i_2}}{T_p} \right\rfloor + \Delta_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}),$$

et

$$\forall n \in \mathbb{N} : t_\pi(T_k) + w_{i_1,k} \leqslant t_\sigma(e_{k,l}, P_{i_1} \to P_{i_2})$$
$$\Leftrightarrow$$
$$\Delta_\pi(T_k) \leqslant \left\lfloor \frac{\tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - \tilde{t}_\pi(T_k) - w_{i_1,k}}{T_p} \right\rfloor + \Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2})$$

(t_π, t_σ) vérifie donc les contraintes de précédence les contraintes de précédence si et seulement si le système de contraintes de potentiel suivant a une solution :

$$\forall e_{k,l} : T_k \to T_l$$
$$\begin{cases} \Delta_\pi(T_k) - \Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) \leqslant \left\lfloor \dfrac{\tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - \tilde{t}_\pi(T_k) - w_{i_1,k}}{T_p} \right\rfloor \\ \Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - \Delta_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}) \leqslant \left\lfloor \dfrac{\tilde{t}_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}) - \tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - data_{k,l} \times c_{i_1,i_2}}{T_p} \right\rfloor \\ \vdots \\ \Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) - \Delta_\pi(T_l) \leqslant \left\lfloor \dfrac{\tilde{t}_\pi(T_l) - \tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) - data_{k,l} \times c_{i_{p-1},i_p}}{T_p} \right\rfloor \end{cases}$$

G_A étant un graphe sans cycle, le graphe de potentiel associé à ces équations ne comporte pas non plus de cycle et donc, en particulier ne contient aucun cycle de poids négatif. Il a donc une solution qui peut être calculée en temps polynomial [40].

- **Contraintes de ressource :** Pour tout $I_1, I_2 \subset [0, T_p[$, on a $(I_1 + T_p\mathbb{Z}) \cap (I_2 + T_p\mathbb{Z}) = (I_1 \cap I_2) + T_p\mathbb{Z}$. $(\tilde{t}_\pi, \tilde{t}_\sigma)$ vérifiant les contraintes de ressources, (t_π, t_σ) vérifient également les contraintes de ressources.
- **1-périodicité de période T_p :** pour tout T_k, $t_\pi(T_k)$ est clairement 1-périodique de période T_p et pour tout $e_{k,l}$, $t_\sigma(e_{k,l})$ est également clairement 1-périodique de période T_p. ■

Lemme 4.2. *Si M_{K_1,T_p} et M_{K_2,T_p} sont surjectives alors $M_{K_1+K_2,T_p}$ l'est également.*

Démonstration. Soit $(\tilde{t}_\pi, \tilde{t}_\sigma)$ un (K_1+K_2)-motif de longueur T_p. Alors en définissant $\tilde{t}^{(1)}_\pi$ de $V_A \times [\![1, K_1]\!]$ dans $[0, T_p[$ par $\tilde{t}^{(1)}_\pi(T_k, n) = \tilde{t}_\pi(T_k, n)$ et $\tilde{t}^{(1)}_\sigma$ de $(E_A \times [\![1, K]\!]) \times E_P$ dans $[0, T_p[$ par $\tilde{t}^{(1)}_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma((e_{k,l}, n), P_i \to P_j)$, $(\tilde{t}^{(1)}_\pi, \tilde{t}^{(1)}_\sigma)$ est un K_1-motif de longueur T_p. De même, en définissant $\tilde{t}^{(2)}_\pi$ de $V_A \times [\![1, K_2]\!]$ dans $[0, T_p[$ par $\tilde{t}^{(2)}_\pi(T_k, n) = \tilde{t}_\pi(T_k, K_1+n)$ et $\tilde{t}^{(2)}_\sigma$ de $(E_A \times [\![1, K_2]\!]) \times E_P$ dans $[0, T_p[$ par $\tilde{t}^{(2)}_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma((e_{k,l}, K_2+n), P_i \to P_j)$, $(\tilde{t}^{(2)}_\pi, \tilde{t}^{(2)}_\sigma)$ est un K_2-motif de longueur T_p.

M_{K_1,T_p} et M_{K_2,T_p} étant surjectives, soit $(t^{(1)}_\pi, t^{(1)}_\sigma)$ (resp. $(t^{(2)}_\pi, t^{(2)}_\sigma)$) de motif $(\tilde{t}^{(1)}_\pi, \tilde{t}^{(1)}_\sigma)$ (resp. $(\tilde{t}^{(2)}_\pi, \tilde{t}^{(2)}_\sigma)$). Alors t_π en définissant t_π de $V_A \times \mathbb{N}$ dans $\mathbb{R}$ par

$$t_\pi(T_k, n) = \begin{cases} t^{(1)}_\pi(T_k, n) & \text{si } n \in [\![1, K_1]\!] + (K_1 + K_2)\mathbb{Z} \\ t^{(2)}_\pi(T_k, n) & \text{sinon} \end{cases}$$

et t_σ de $(E_A \times \mathbb{N}) \times E_P$ dans $\mathbb{R}$ par

$$t_\sigma((e_{k,l}, n), P_i \to P_j) = \begin{cases} t^{(1)}_\sigma((e_{k,l}, n), P_i \to P_j) & \text{si } n \in [\![1, K_1]\!] + (K_1 + K_2)\mathbb{Z} \\ t^{(2)}_\sigma((e_{k,l}, n), P_i \to P_j) & \text{sinon} \end{cases},$$

on obtient un ordonnancement $(K_1 + K_2)$-cyclique de période T_p et de motif $(\tilde{t}_\pi, \tilde{t}_\sigma)$. ■

Théorème 4.1. *Pour tout $K \in \mathbb{N}^*$, M_{K,T_p} est surjective.*

Démonstration. Récurence triviale en utilisant les deux lemmes précédents. ■

4.3 Régime permanent sur une plate-forme quelconque

Cette section montre comment obtenir un ordonnancement périodique de fréquence maximale. Cette construction s'effectue en deux temps. Dans les sections 4.3.1 à 4.3.4, on écrit sous forme d'un programme linéaire les équations du régime permanent afin de calculer une borne supérieure de la fréquence des ordonnancements cycliques. Nous montrons ensuite en section 4.3.5 comment utiliser ce programme linéaire pour construire un motif valide, et donc un ordonnancement périodique, atteignant cette fréquence. Un tel ordonnancement est donc de fréquence maximale parmi l'ensemble des ordonnancements cycliques.

Nous montrerons de plus en section 4.5.1 que l'ordonnancement périodique ainsi construit est asymptotiquement optimal, c'est-à-dire qu'en temps T, il n'est possible d'exécuter qu'un nombre constant (indépendant de T) de tâches de plus.

4.3.1 Définitions

Pour toute arête $e_{k,l} : T_k \to T_l$ du graphe de tâches et pour chaque paire de processeurs $P_i \to P_j$, on note $s(P_i \to P_j, e_{k,l})$ le temps moyen passé par P_i à envoyer des données de type $e_{k,l}$ au processeur P_j. Le nombre $s(P_i \to P_j, e_{k,l})$ est donc un rationnel positif. On note également $sent(P_i \to P_j, e_{k,l})$ le nombre moyen de fichiers de type $e_{k,l}$ envoyés de P_i à P_j par unité de temps. On a donc la relation

$$s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}) \times (data_{k,l} \times c_{i,j}) , \tag{4.1}$$

ce qui signifie simplement que la fraction de temps passée à transférer de tels fichiers est égal au nombre de fichiers de ce type transférés par unité de temps multiplié par le temps nécessaire à en transférer un.

Pour chaque tâche T_k et chaque processeur P_i, on note $\alpha(P_i, T_k)$ le temps moyen passé à traiter des tâches de type T_k sur le processeur P_i et $cons(P_i, T_k)$ le nombre moyen de tâches de type T_k traitées par unité de temps sur le processeur P_i. Comme précédemment, $\alpha(P_i, T_k)$ et $cons(P_i, T_k)$ sont des nombres rationnels positif et on a la relation suivante :

$$\alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k} \tag{4.2}$$

4.3.2 Équations du régime permanent

Nous cherchons donc des valeurs rationnelles des variables $s(P_i \to P_j, e_{k,l})$, $sent(P_i \to P_j, e_{k,l})$, $\alpha(P_i, T_k)$ et $cons(P_i, T_k)$. En régime permanent, ces quantités sont soumises aux contraintes suivantes :

Activité sur une période unitaire Les fractions de temps passées par les différents processeurs à faire quelque chose, que ce soit calculer ou communiquer, doivent appartenir à l'intervalle $[0,1]$ puisque ces quantités correspondent à une activité moyenne pendant une unité de temps :

$$\forall P_i, \forall T_k \in V_A, 0 \leqslant \alpha(P_i, T_k) \leqslant 1 \tag{4.3}$$

$$\forall P_i, P_j, \forall e_{k,l} \in E_A, 0 \leqslant s(P_i \to P_j, e_{k,l}) \leqslant 1 \tag{4.4}$$

Modèle 1-port en sortie Les émissions du processeur P_i devant être effectuées séquentiellement vers ses voisins, on a l'équation suivante :

$$\forall P_i, \sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1, \tag{4.5}$$

Modèle 1-port en entrée Les réceptions du processeur P_i devant être effectuées séquentiellement, on a l'équation suivante :

$$\forall P_i, \sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) \leqslant 1 \tag{4.6}$$

Recouvrement des calculs et des communications En raison des hypothèses faites sur le recouvrement, il n'y a pas d'autres contraintes sur $\alpha(P_i, T_k)$ que

$$\forall P_i, \sum_{T_k \in V_A} \alpha(P_i, T_k) \leqslant 1 \tag{4.7}$$

4.3.3 Loi de conservation

Les dernières contraintes sont des *lois de conservation* Soit P_i un processeur et $e_{k,l}$ une arête du graphe de tâches. En une unité de temps, P_i reçoit de ses voisins un certain nombre de fichiers de type $e_{k,l}$: $\sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l})$ exactement. Le processeur P_i exécute certaines tâches T_k lui même et génère donc autant de fichiers de type $e_{k,l}$. Qu'arrive-t-il à ces fichiers ? Certains sont envoyés aux voisins de P_i et d'autres sont utilisés par P_i pour traiter de tâches de type T_l. On en déduit l'équation suivante :

$$\begin{aligned} &\forall P_i, \forall e_{k,l} : T_k \to T_l \in E_A, \\ &\sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k) = \sum_{P_i \to P_j} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l) \end{aligned} \tag{4.8}$$

Cette équation permet de traiter le problème dans toute sa généralité (plusieurs maîtres, rapatriement des fichiers de sortie en des endroits bien définis, etc.) mais peut se formuler plus simplement si on suppose, par exemple, qu'il n'y a qu'un seul maître et que les fichiers de sorties n'ont pas besoin d'être rapatriés. On peut alors simplement noter $sent(P_i \to P_j)$ le nombre de fichiers envoyés par P_i à P_j en une unité de temps et

$cons(P_i)$ le nombre de tâches consommées par P_i en une unité de temps. On a alors une relation beaucoup plus simple :

$$\forall P_i, \sum_{P_j \to P_i} sent(P_j \to P_i) = cons(P_i) + \sum_{P_i \to P_j} sent(P_i \to P_j)$$

Autrement dit, toute tâche reçue est soit consommée, soit déléguée à quelqu'un d'autre. Il est important de comprendre que ces équations ne s'appliquent qu'au régime permanent. Au début de la mise en marche de la plate-forme, seuls les fichiers d'entrée sont disponibles et peuvent être envoyés à d'autres machines. Puis viennent des calculs, rendus possibles par la disponibilité de fichiers d'entrée, qui génèrent d'autres fichiers permettant à d'autres tâches d'être exécutées, et ainsi de suite. À la fin d'une phase d'initialisation, on entre en régime permanent et à chaque période, chaque processeur peut en même temps envoyer des tâches, recevoir ou envoyer des données correspondant à d'autres tâches.

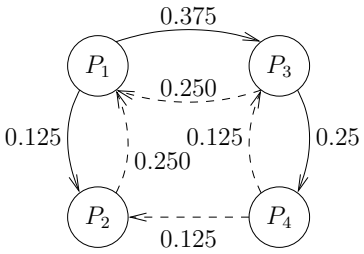
4.3.4 Calcul du régime permanent optimal

Les équations précédentes forment un programme linéaire dont l'objectif est de maximiser le débit de la plate-forme, c'est-à-dire le nombre de tâches T_{end} consommées par unité de temps :

$$\begin{array}{l}
\text{MAXIMISER } \rho = \sum_{i=1}^{p} cons(P_i, T_{end}), \\
\text{SOUS LES CONTRAINTES} \\
\left\{ \begin{array}{ll}
\text{(4.9a)} & \forall P_i, \forall T_k \in V_A, 0 \leqslant \alpha(P_i, T_k) \leqslant 1 \\
\text{(4.9b)} & \forall P_i \to P_j, \forall e_{k,l} \in E_A, 0 \leqslant s(P_i \to P_j, e_{k,l}) \leqslant 1 \\
\text{(4.9c)} & \forall P_i \to P_j, \forall e_{k,l} \in E_A, s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}) \times (data_{k,l} \times c_{i,j}) \\
\text{(4.9d)} & \forall P_i, \forall T_k \in V_A, \alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k} \\
\text{(4.9e)} & \forall P_i, \sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) \leqslant 1 \\
\text{(4.9f)} & \forall P_i, \sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1 \\
\text{(4.9g)} & \forall P_i, \sum_{T_k \in V_A} \alpha(P_i, T_k) \leqslant 1 \\
\text{(4.9h)} & \forall P_i, \forall e_{k,l} \in E_A : T_k \to T_l, \\
 & \qquad \sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k) = \\
 & \qquad\qquad \sum_{P_i \to P_j} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l)
\end{array} \right.
\end{array} \tag{4.9}$$

	$\alpha(P_i, T_1)$	$cons(P_i, T_1)$
P_1	100%	0.025
P_2	100%	0.125
P_3	100%	0.125
P_4	100%	0.250
Total	21 tâches toutes les 40 secondes	

(a) Répartition des calculs



(b) Communications

FIG. 4.2 – Solution du programme linéaire : le graphe de plate-forme est annoté avec les valeurs non-nulles des $sent(P_i \to P_j, e_{k,l})$; les flèches continues représentent des transferts de type $e_{begin,1}$ et les flèches pointillées des transferts de type $e_{1,end}$.

Nous montrons dans la section suivante comment construire un ordonnancement cyclique qui atteint la fréquence calculée par le programme linéaire.

4.3.5 Ordonnancement

Dans cette section, nous montrons comment obtenir, à partir de la solution du programme linéaire (4.9), un ordonnancement asymptotiquement optimal. Considérons l'exemple de la figure 4.1. On suppose que les fichiers d'entrée $e_{begin,1}$ sont à l'origine sur P_1 et que tous les fichiers de sortie $e_{1,end}$ doivent être rassemblés sur P_1. Ces conditions imposent donc que ni T_{begin}, ni T_{end} ne peuvent être exécutées ailleurs que sur P_1.

En résolvant le programme linéaire (4.9), on obtient les valeurs résumées sur la figure 4.2. Ce régime permanent, de débit $\rho = 0.525$, pourra être atteint si on arrive a exhiber un K-motif de longueur T_p tel que $\rho = K/T_p$. La construction de ce motif s'effectue en deux temps. On commence par décomposer la solution du programme linéaire en une somme pondérée d'allocations $(\alpha_i, \mathcal{A}_i)$ (voir figure 4.3), puis on montre qu'il est possible de rendre ces allocations compatibles entre elle pour construire le motif.

La motivation principale pour se placer en régime permanent est que plusieurs allocations sont mélangées pour tirer parti au mieux de la plate-forme. Dans le cas où G_A est simplement constitué d'une tâche principale, d'une tâche T_{begin} et d'une tâche T_{end}, la décomposition se fait simplement en «épluchant» le graphe (voir algorithme 4.1).

L'algorithme 4.1, parcoure G_A en profondeur et sélectionne gloutonnement les processeurs capables d'effectuer les différentes tâches. Les équations de conservation nous garantissent qu'une telle allocation peut être trouvée puisque si une tâche est consommée, elle produit un fichier de sortie qui est soit utilisé directement sur place, soit transmis à un autre processeur. Une fois une allocation valide déterminée, on détermine son poids en prenant le minimum des différentes quantités $cons(P_i, T_k)$ et $sent(P_i \to P_j, e_{k,l})$ impliquées dans l'allocation. En soustrayant ce poids à ces différentes quantités, on obtient des

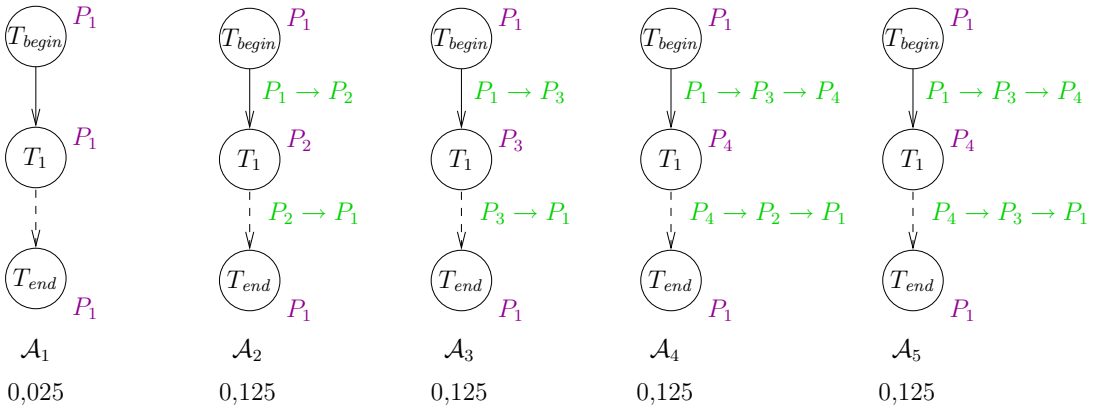
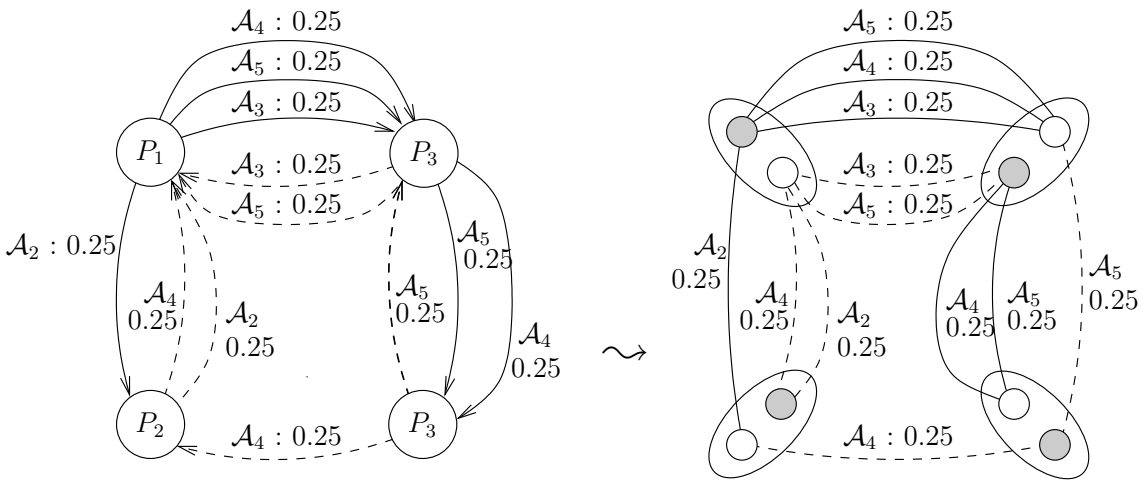


FIG. 4.3 – Décomposition de la solution du programme linéaire en 5 allocations $\mathcal{A}_1$,...,$\mathcal{A}_5$. Chacune de ces allocations participe pour une certaine fraction au régime permanant et la somme de leur contribution $(0.025 + 0.125 + 0.125 + 0.125 + 0.125)$ est égale au débit total $(0.525 = \frac{21}{40})$.

FIND_A_SCHEDULE()
1: Soit P_i tel que $cons(P_i, T_{begin}) = 0$
2: $\pi(T_{begin}) \leftarrow P_i$.
3: $src \leftarrow i$
4: $\gamma_1 \leftarrow \emptyset$
5: **Tant que** $cons(P_{src}, T_1) = 0$ **:**
6: Soit P_j tel que $sent(P_{src} \to P_j, e_{begin,1}) > 0$
7: $\gamma_1 \leftarrow \gamma_1 \cup (P_{src} \to P_j)$
8: $src \leftarrow j$
9: $\sigma(e_{begin,1}) \leftarrow \gamma_1$
10: $\pi(T_1) \leftarrow P_{src}$
11: $\gamma_2 \leftarrow \emptyset$
12: **Tant que** $cons(P_{src}, T_{end}) = 0$ **:**
13: Soit P_j tel que $sent(P_{src} \to P_j, e_{1,end}) > 0$
14: $\gamma_2 \leftarrow \gamma_2 \cup (P_{src} \to P_j)$
15: $src \leftarrow j$
16: $\sigma(e_{1,end}) \leftarrow \gamma_2$
17: $\pi(T_{end}) \leftarrow P_{src}$
18: $\alpha \leftarrow \min\big(cons(P_{begin}, T_{\pi(T_{begin})}), cons(P_1, T_{\pi(T_1)}), cons(P_{end}, T_{\pi(T_{end})}),$
$\{sent(P_i \to P_j, e_{begin,1}) | P_i \to P_j \in \sigma(e_{begin,1})\},$
$\{sent(P_i \to P_j, e_{1,end}) | P_i \to P_j \in \sigma(e_{1,end})\}\big)$
19: **Renvoyer**(α, π, σ)

Algorithme 4.1: Algorithme d'extraction d'une allocation



(a) Solution du programme linéaire (b) Graphe biparti associé

FIG. 4.4 – Graphe des communications et graphe biparti associé.

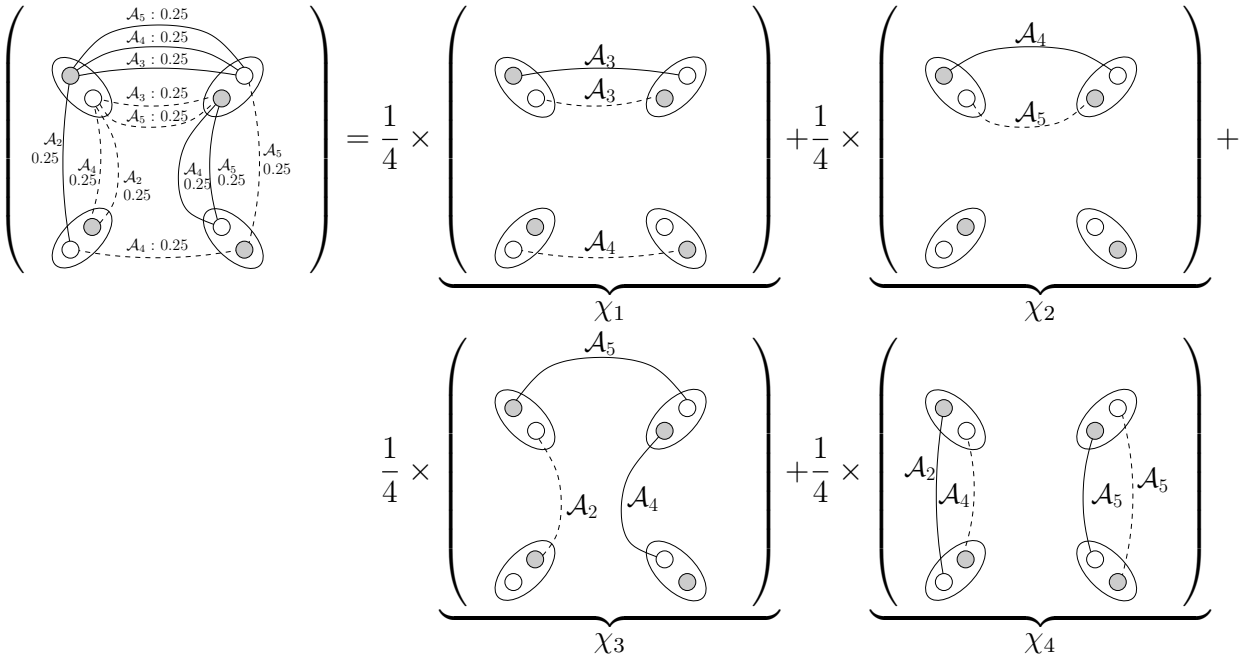
variables qui vérifient toujours les contraintes (4.9) et on peut donc recommencer jusqu'à ce que plus aucune tâche ne soit consommée.

Ces valeurs nous permettent de construire le *graphe de communications* représenté figure 4.4(a). Chaque sommet correspond à un processeur de la plate-forme. Les flèches continues représentent des transferts de type $e_{begin,1}$ d'un processeur à l'autre et les flèches pointillées des transferts de type $e_{1,end}$. Chaque arête est étiquetée par un numéro d'allocation et par un poids qui représente la fraction de temps passée au transfert de ce type de fichier entre les deux machines pour une allocation donnée. Ainsi, l'arête $(\mathcal{A}_3, 0.25)$ en trait continu qui relie P_1 à P_2 représente le fait qu'en une unité de temps, 0.25 unité de temps sont passées à faire transiter de P_1 vers P_2 des arêtes de type $e_{begin,1}$ qui seront utilisées pour l'allocation $\mathcal{A}_3$. Le fait qu'il y ait plusieurs arêtes d'un processeur à un autre permet donc de modéliser les contraintes de ressources.

Pour reconstruire un ordonnancement à partir de cette description, nous commençons pas transformer ce graphe en un graphe biparti pondéré (voir figure 4.4(b)) en divisant chaque sommet en deux : un pour les émissions (en gris) et un pour les réceptions (en blanc).

Comme nous nous sommes mis dans le cas d'un recouvrement complet des communications et que les processeurs ne sont capable de recevoir des données que d'une seul personne à la fois (1 port en entrée) et de n'envoyer des données qu'à une seule personne à la fois (1 port en sortie), à chaque instant il y a au plus deux communications pour un processeur donné : une pour les émissions, et une pour les réceptions. Ainsi, à un instant donné, seules les communications correspondant à un couplage dans le graphe biparti peuvent avoir lieu. Cette transformation nous permet donc de modéliser les contraintes 1-port. Nous devons donc décomposer notre graphe biparti pondéré en une somme pon-

dérée de couplages telle que la somme des coefficients est inférieure à 1 (de façon à abtenir des communications réalisables en une unité de temps). Sur notre exemple, il est possible d'utiliser la décomposition suivante :



(4.10)

L'obtention d'une telle décomposition n'est pas triviale mais elle existe néanmoins toujours et peut être construite en temps $O(m^2)$, où m est le nombre d'arêtes du graphe [96, vol.A chapitre 20]. Le nombre de couplage est borné par m et, à partir d'une telle décomposition, il est aisé de construire un ordonnancement qui atteint le régime permanent optimal (voir figure 4.5) puisqu'on a alors un motif qui respecte les contraintes de ressource et les contraintes 1-port.

Pour résumer, la construction d'un tel ordonnancement se fait de la façon suivante :

- Résoudre le programme linéaire.
- Décomposer la solution en une somme d'allocation et construire le graphe des communications induit par les $s(P_i \to P_j, e_{k,l})$.
- Transformer le graphe de communications en un graphe pondéré biparti B.
- Décomposer le graphe B en une somme pondérée de couplages $B = \sum \alpha_c \chi_c$ telle que $\sum \alpha_c \leqslant 1$.
- En notant, $\alpha(P_i, T_1) = \frac{a_i}{b_i}$ et $\alpha_c = \frac{p_c}{q_c}$, on peut définir la période T_p permettant d'obtenir le régime permanent comme le plus petit commun multiple des $b_i \times w_{i,k}$ et des $q_c \times data_{k,l} \times c_{i,j}$ si le transfert d'un fichier de type $e_{k,l}$ de P_i à P_j appartient au couplage χ_c. Ainsi, un nombre entier de tâches de chaque type est consommé à chaque période T_p (grâce aux $\alpha(P_i, T_1)$) et un nombre entier de fichiers est transféré durant chacun des moments correspondant aux couplages. Dans le cas de l'exemple

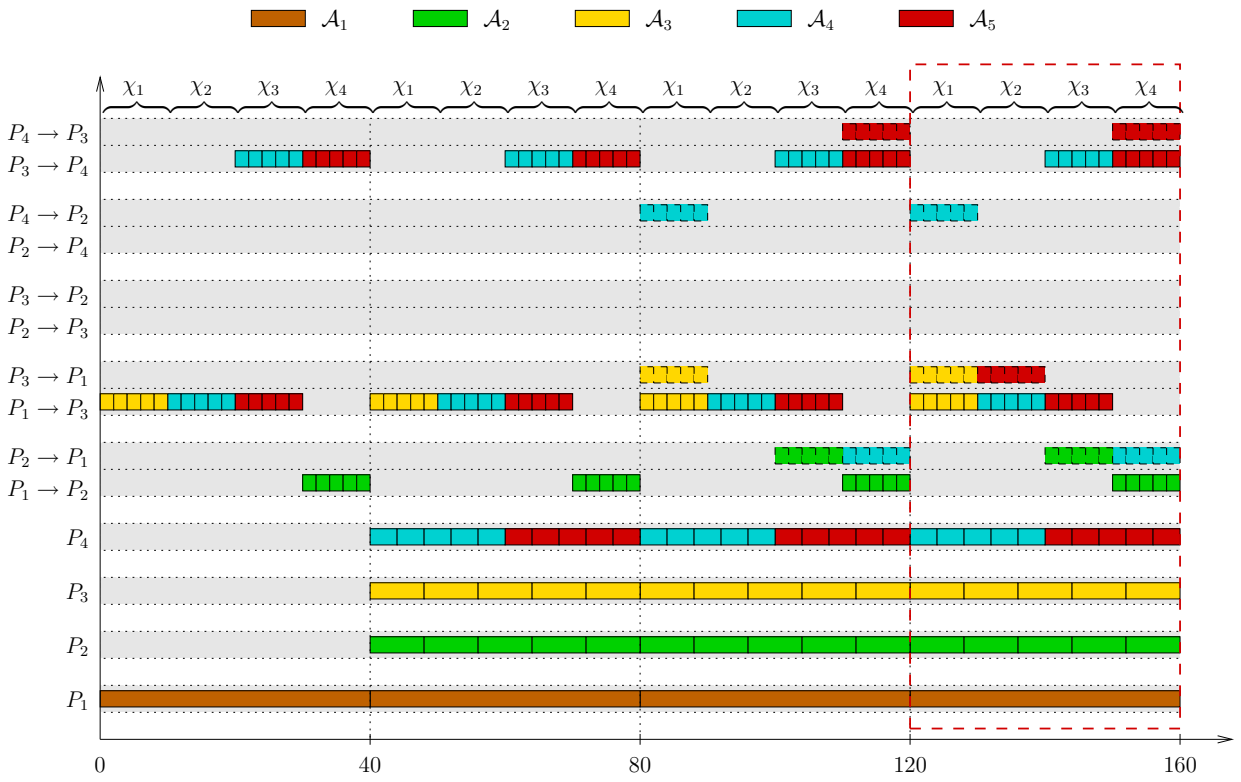


FIG. 4.5 – Ordonnancement atteignant le régime permanent optimal

précédent, T_p était le plus petit commun multiple de $10, 2, 2, 1$ (pour P_1, P_2, P_3, P_4) et de $4 \times 2 = 8$ pour les différents couplages (ici, tous les temps de transferts et les poids des couplages sont identiques). T_p est donc égal à 40 sur notre exemple.

– Par construction de la composition, chaque couplage χ_i est un ensemble de communications compatibles de $e_{begin,1}$ ou de $e_{1,end}$ d'un P_i vers un P_j et correspondant à une allocation donnée. On a donc construit un K-motif (ici, $K = 1+5+5+5+5 = 21$) de longueur T_p avec K/T_p égal au débit optimal calculé par le programme linéaire. On en déduit donc un ordonnancement périodique de débit optimal.
Notons que la description de l'ordonnancement est bien polynomiale en les entrées du problème. En effet, le nombre de variables du programme linéaire est un $O(n^2p^2)$ (à cause des $s(P_i \to P_j, e_{k,l})$), le graphe pondéré biparti est donc bien polynomial en n et p. Le nombre d'allocations est donc borné par n^2p^2 et le graphe de communications a donc $O(n^4p^4)$ arêtes. Lors de la décomposition en couplage, le nombre de couplages est donc borné par n^4p^4 et la description de la somme pondérée est donc polynomiale. La description de l'ordonnancement est donc bien polynomiale et la durée de la période peut évidemment être codée de façon polynomiale puisqu'elle est obtenue comme résultat du plus petit commun multiple des $b_i \times w_{i,k}$ et des $q_c \times data_{k,l} \times c_{i,j}$.
En revanche, si cette durée peut être codée de façon polynomiale, elle ne l'est pas forcément elle-même. En décrivant ce que fait chaque processeur dans chacune des périodes induites par les couplages, on arrive à une description qui est bien

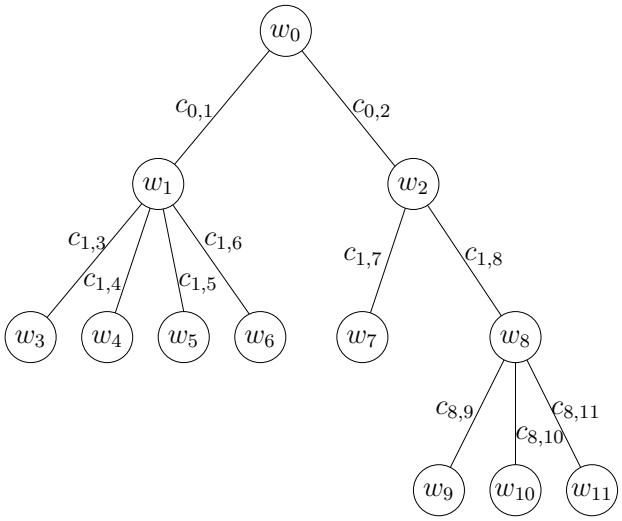


FIG. 4.6 – Plate-Forme en arbre.

polynomiale. En revanche, une description instant par instant des différentes actions de chaque processeur pourrait ne pas être polynomiale.

Hormis quelques périodes au départ et à la fin, le reste de l'ordonnancement est parfait. Nous montrerons en section 4.5.1 que l'ordonnancement cyclique ainsi construit est asymptotiquement optimal, c'est-à-dire qu'en temps T, il n'est possible d'exécuter qu'un nombre constant (indépendant de T) de tâches de plus.

4.4 Régime permanent sur une arborescence

Dans cette section, nous nous intéressons plus particulièrement au cas des plates-formes représentées par une arborescence (voir figure 4.6). Nous supposons également que les fichiers de sortie doivent rester sur place. En effet, dans le cas d'une arborescence, si les fichiers de résultat devaient être renvoyés au maître, ils prendraient le même chemin que les fichiers d'entrée et, dans le cadre du régime permanent, il suffirait donc de considérer des fichiers d'entrée un peu plus gros pour les prendre en compte. Dans ces conditions, nous pouvons reformuler les équations du programme linéaire (4.9) et les résoudre de façon explicite.

4.4.1 Équations

Dans cette section, α_i représente la fraction de temps passée par P_i à calculer. $s_{i,j}$ représente la fraction de temps passée par P_i à envoyer des données à P_j. En utilisant ces

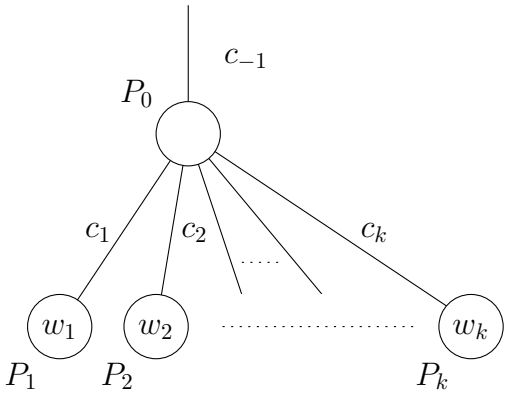


FIG. 4.7 – Graphe en étoile.

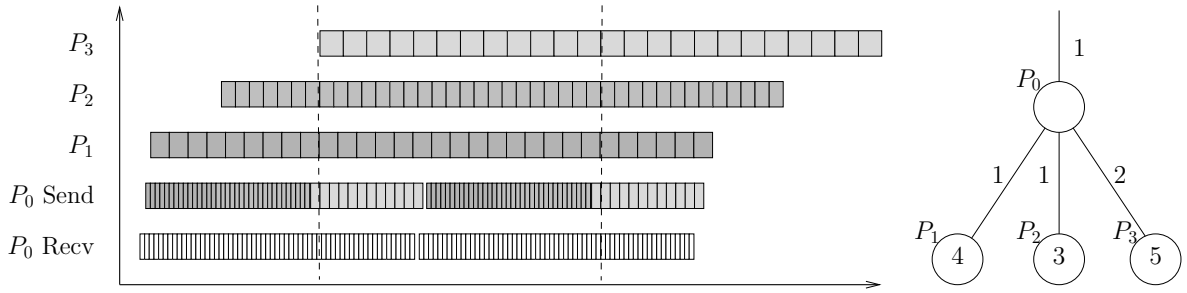
notations, le programme linéaire (4.9) devient :

$$\begin{array}{l}\text{MAXIMISER } \rho = \sum_{i=1}^{p} \frac{\alpha_i}{w_i}, \\ \text{SOUS LES CONTRAINTES} \\ \left\{ \begin{array}{lll} \text{(4.11a)} & \forall i, 0 \leqslant \alpha_i \leqslant 1 & \\ \text{(4.11b)} & \forall P_i \to P_j, 0 \leqslant s_{i,j} \leqslant 1 & \\ \text{(4.11c)} & \forall i, \sum_{P_i \to P_j} s_{i,j} \leqslant 1 & \text{(modèle 1-port)} \\ \text{(4.11d)} & \forall P_i \text{ différent du maître, } \frac{s_{father(i),i}}{c_{father(i),i}} = \frac{\alpha_i}{w_i} + \sum_{P_i \to P_j} \frac{s_{i,j}}{c_{i,j}} & \text{(loi de conservation)} \end{array} \right. \end{array} \tag{4.11}$$

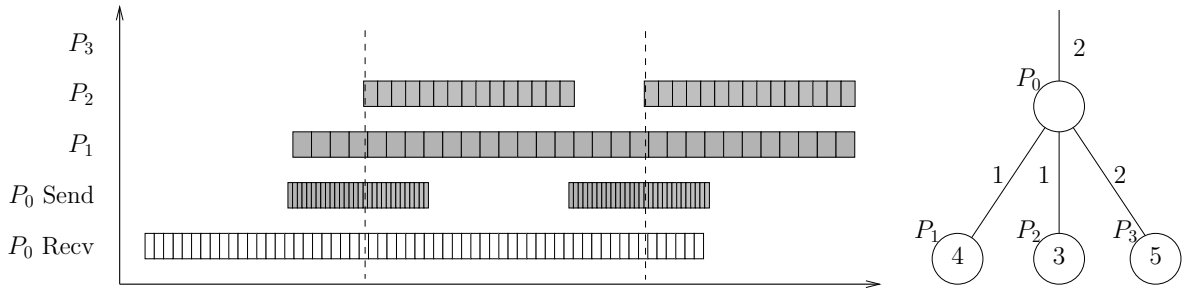
4.4.2 Solution sur une étoile

On commence par résoudre le problème 4.11 dans le cas où la plate-forme est une étoile (voir figure 4.7) constituée d'un nœud P_0 et de k esclaves $P_1, \ldots, P_k$. P_0 a besoin de c_i unités de temps pour envoyer une tâche au processeur P_i et peut recevoir en même temps des tâches de son propre père (disons P_{-1}) en c_{-1} unité de temps. Nous donnons trois exemples dans les figures 4.8(a), 4.8(b) et 4.8(c). Dans le premier tous les esclaves travaillent à temps plein et dans les deux derniers, le réseau est le facteur limitant.

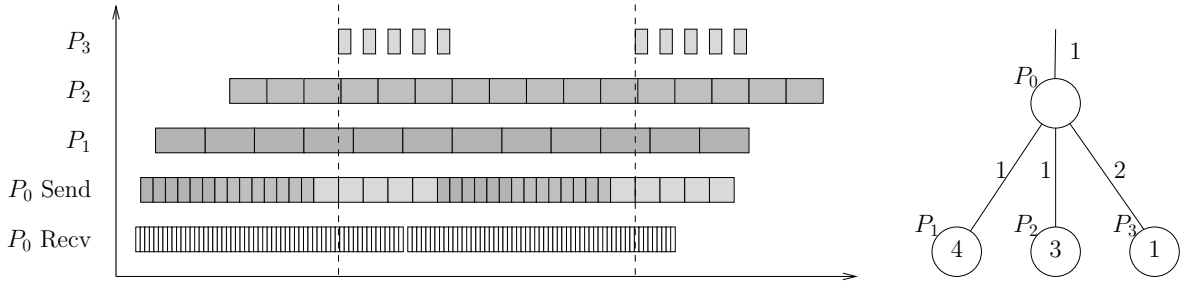
On peut donc simplifier encore un peu les notations par rapport aux sections précédentes. α_i représente toujours la fraction de temps passée par le processeur P_i à calculer mais on note s_i (et pas $s_{0,i}$) la fraction de temps passée à envoyer des données à P_i. Enfin, on note r_{-1} le temps passé par P_0 à recevoir des données de son père.



(a) Premier exemple, sans saturation de la bande passante : il est possible de garder tous les esclaves en activité



(b) Deuxième exemple, avec saturation de la bande passante : certains esclaves doivent demeurer inactifs en raison de la faible bande-passante entre P_0 et son père



(c) Troisième exemple, avec saturation de la bande passante : l'esclave P_3 reste partiellement inactif en raison de la valeur élevée de sa puissance de calcul

FIG. 4.8 – Quelques exemples

On s'est donc ramené au problème suivant :

$$
\begin{array}{l}
\text{MAXIMISER } \rho = \frac{\alpha_0}{w_0} + \sum_{i=1}^{k} \frac{\alpha_i}{w_i}, \\
\text{SOUS LES CONTRAINTES} \\
\left\{
\begin{array}{lll}
(4.12a) & \forall i \in [\![0, k]\!], 0 \leqslant \alpha_i \leqslant 1 & \\
(4.12b) & \forall i \in [\![1, k]\!], 0 \leqslant s_i \leqslant 1 \text{ et } 0 \leqslant r_{-1} \leqslant 1 & \\
(4.12c) & \displaystyle\sum_{i=1}^{k} s_i \leqslant 1 & \text{(modèle 1-port)} \\
(4.12d) & \displaystyle\frac{r_{-1}}{c_{-1}} = \frac{\alpha_0}{w_0} + \sum_{i=1}^{k} \frac{s_i}{c_i} & \text{(loi de conservation pour le maître)} \\
(4.12e) & \forall i \in [\![1, k]\!], \displaystyle\frac{s_i}{c_i} = \frac{\alpha_i}{w_i} & \text{(loi de conservation pour les esclaves)}
\end{array}
\right.
\end{array}
\tag{4.12}
$$

Lemme 4.3. *En utilisant les notations précédentes, le rendement $\rho = \sum_{i=0}^{k} \frac{\alpha_i}{w_i}$ pour une étoile peut être obtenu de la façon suivante :*

1. *Trier les esclaves par temps de communication croissant et les numéroter de façon à ce que $c_1 \leqslant c_2 \ldots \leqslant c_k$.*
2. *Soit q le plus grand indice tel que $\sum_{i=1}^{q} \frac{c_i}{w_i} \leqslant 1$. Si $q < k$ définissons $\varepsilon = 1 - \sum_{i=1}^{q} \frac{c_i}{w_i}$ et sinon posons $\varepsilon = 0$.*
3. *Alors*

$$\rho = \min\left(\frac{1}{c_{-1}}, \frac{1}{w_0} + \sum_{i=1}^{q} \frac{1}{w_i} + \frac{\varepsilon}{c_{q+1}}\right) \tag{4.13}$$

Intuitivement, les processeurs ne peuvent pas consommer plus de tâches que P_{-1} n'en envoie, d'où le premier terme de l'équation (4.13). Pour ce qui est du second terme, quand $k = q$, cela signifie simplement que l'on peut envoyer des tâches aux esclaves suffisamment rapidement pour qu'ils travaillent en permanence. Si $q < k$, cela signifie que certains esclaves ne seront pas alimentés, que ce sont ceux dont le lien de communication est lent, et ce quelque soit leur puissance de calcul. En d'autre termes, un processeur lent avec un lien de communication rapide est préférable à un processeur rapide doté d'un lien de communication lent.

Démonstration. Introduisons la notation R_i, pour $i \in [\![1, k]\!]$ qui représente le nombre de tâches consommées par secondes par P_i (et donc aussi le nombre de tâches envoyées par seconde par P_0 au processeur P_i). R_{-1} représente le nombre de tâches envoyées par

seconde à P_0. Notre problème est donc équivalent au problème suivant :

$$\begin{array}{l}\text{MAXIMISER } \rho = \sum_{i=0}^{k} R_i, \\ \text{SOUS LES CONTRAINTES} \\ \begin{cases} \text{(4.14a)} & R_{-1} = \displaystyle\sum_{i=0}^{k} R_i \\ \text{(4.14b)} & R_{-1}c_{-1} \leqslant 1 \\ \text{(4.14c)} & \forall i \in [\![1,k]\!], R_i w_i \leqslant 1 \\ \text{(4.14d)} & \displaystyle\sum_{i=1}^{k} R_i c_i \leqslant 1 \end{cases} \end{array} \tag{4.14}$$

Lemme 4.4. *Si on note $R = (\rho, R_{-1}, R_0, \ldots, R_k)$ une solution du problème précédant alors $\rho = \min\left(\frac{1}{c_{-1}}, \frac{1}{w_0} + S\right)$ où S est la solution du programme linéaire suivant :*

$$\begin{array}{l}\text{MAXIMISER } S = \sum_{i=1}^{k} R_i, \\ \text{SOUS LES CONTRAINTES} \\ \begin{cases} \text{(4.15a)} & \forall i \in [\![1,k]\!], R_i w_i \leqslant 1 \\ \text{(4.15b)} & \displaystyle\sum_{i=1}^{k} R_i c_i \leqslant 1 \end{cases} \end{array} \tag{4.15}$$

Démonstration. Ce programme auxiliaire étant moins contraint que le programme original on a clairement $\frac{1}{w_0} + S \geqslant \rho$. Grâce aux contraintes 4.14a et 4.14b, on a également $\frac{1}{w_{-1}} \geqslant \rho$ et donc $\rho \leqslant \min\left(\frac{1}{c_{-1}}, \frac{1}{w_0} + S\right)$.
Montrons la réciproque. Supposons d'abord que $\min\left(\frac{1}{c_{-1}}, \frac{1}{w_0} + S\right) = \frac{1}{c_{-1}}$. Alors, soit $(R'_1, \ldots, R'_k)$ une solution du programme auxiliaire. On a donc $S = \sum_{i=1}^{k} R_i$ et $\frac{1}{w_0} + S \geqslant \frac{1}{c_{-1}}$. Donc, en posant $\alpha = \frac{\frac{1}{c_{-1}}}{\frac{1}{w_0}+S} \leqslant 1$, $(\frac{1}{c_{-1}}, \frac{\alpha}{w_0}, \alpha R'_1, \ldots, \alpha R'_k)$ est une solution du problème initial dont la valeur de la fonction objectif est $\frac{1}{c_{-1}}$, et on a donc $\frac{1}{c_{-1}} \leqslant \rho$. Supposons maintenant que $\min\left(\frac{1}{c_{-1}}, \frac{1}{w_0} + S\right) = \frac{1}{w_0} + S$. Soit $(R'_1, \ldots, R'_k)$ une solution optimale du programme auxiliaire. Alors $(\frac{1}{w_0} + \sum_{i=1}^{k} R'_i, \frac{1}{w_0}, R'_1, \ldots, R'_k)$ est une solution du problème initial dont la valeur de la fonction objectif est $\frac{1}{w_0} + S$. On a donc $\frac{1}{w_0} + S \leqslant \rho$, ce qui permet de ce concentrer sur la résolution du problème auxiliaire. ■

Le programme linéaire (4.15) peux s'interprèter comme un simple problème de sac à dos fractionnaire [40, Chapitre 17]. $\frac{1}{w_i}$ représente la quantité maximale de objet de

type i que l'on puisse sélectionner et c_i représente son volume. On souhaite prendre le plus d'objets possible. Il est donc naturel de choisir les «objets» de plus petit volume en priorité et d'en prendre le plus possible, c'est-à-dire de choisir en priorité les processeurs avec un c_i faible.

La solution optimale au problème (4.15) est donc $S = \sum_{i=1}^{k} R_i = \sum_{i=1}^{q} \frac{1}{w_i} + \frac{\varepsilon}{c_{q+1}}$, ce qui établit notre résultat. ■

Remarque. *Les c_i et les w_i appartiennent à $\mathbb{R}^+ \cup \{+\infty\}$. On fait l'hypothèse que $1/\infty = 0$, que $1/0 = \infty$ et que $0/0 = 0$. Nous détaillons quelques cas de figure :*

- *Si $c_{-1} = 0$, le premier terme de l'équation (4.13) est alors infini. Le maître n'étant pas limité par son lien de communication avec son père, seul les esclaves déterminent le débit de la plate-forme.*
- *Si $c_{-1} = \infty$, alors le maître ne peut recevoir de tâches et le débit est alors bien égal à 0 puisque le premier terme de l'équation (4.13) est nul.*
- *Si un des c_i (pour $i \geqslant 1$) est nul on n'effectue de division par c_i que si $i = q+1$, ce qui signifie que soit $\varepsilon = 0$, soit $\sum_{j=1}^{i} c_j/w_j > 1$. Ce dernier cas n'est pas possible puisque comme les processeurs sont triés par c_j croissant, la somme en question est nulle. Cette division par 0 n'en est donc pas une...*
- *Si $w_0 = 0$, le processeur P_0 est capable de traiter toutes les tâches qu'il reçoit. Le second terme de l'équation (4.13) étant alors infini, le débit est uniquement limité par le lien de communication reliant P_0 à son père.*

Du point de vue de l'analogie avec un problème de sac à dos, $c_i = 0$ ou $c_i = \infty$ signifie simplement que le volume d'un objet peut être nul (auquel cas, le prendre ne coûte rien) ou infini (auquel cas il est impossible de le prendre). $w_i = \infty$ ou $w_i = 0$ signifie simplement qu'un objet donné n'est pas disponible du tout ou au contraire qu'il y en a une quantité illimitée.

4.4.3 Solution sur une arborescence quelconque

On peut calculer le régime permanent pour une arborescence en remontant l'arborescence à partir des feuilles et en utilisant le lemme 4.3.

Théorème 4.2. *Le régime permanent $\rho_{opt}(T)$ d'une arborescence T arbitraire peut être calculé de la façon suivante :*

1. *Pour toute sous-arborescence F de T de profondeur 1, remplacer cette sous-arborescence par un nœud de poids $w = \frac{1}{\rho_{opt}(F)}$ où $\rho_{opt}(F)$ est calculé à l'aide du lemme 4.3.*
2. *Itérer le processus jusqu'à ce qu'il ne reste plus qu'un seul nœud.*

Le débit $\rho_{opt}(T)$ de la plate-forme est égal à l'inverse du poids du nœud restant.

Démonstration. Ceci vient du fait qu'en régime permanent, une étoile F constituée d'un père et de plusieurs esclaves a le même comportement qu'un seul nœud de poids $w =$

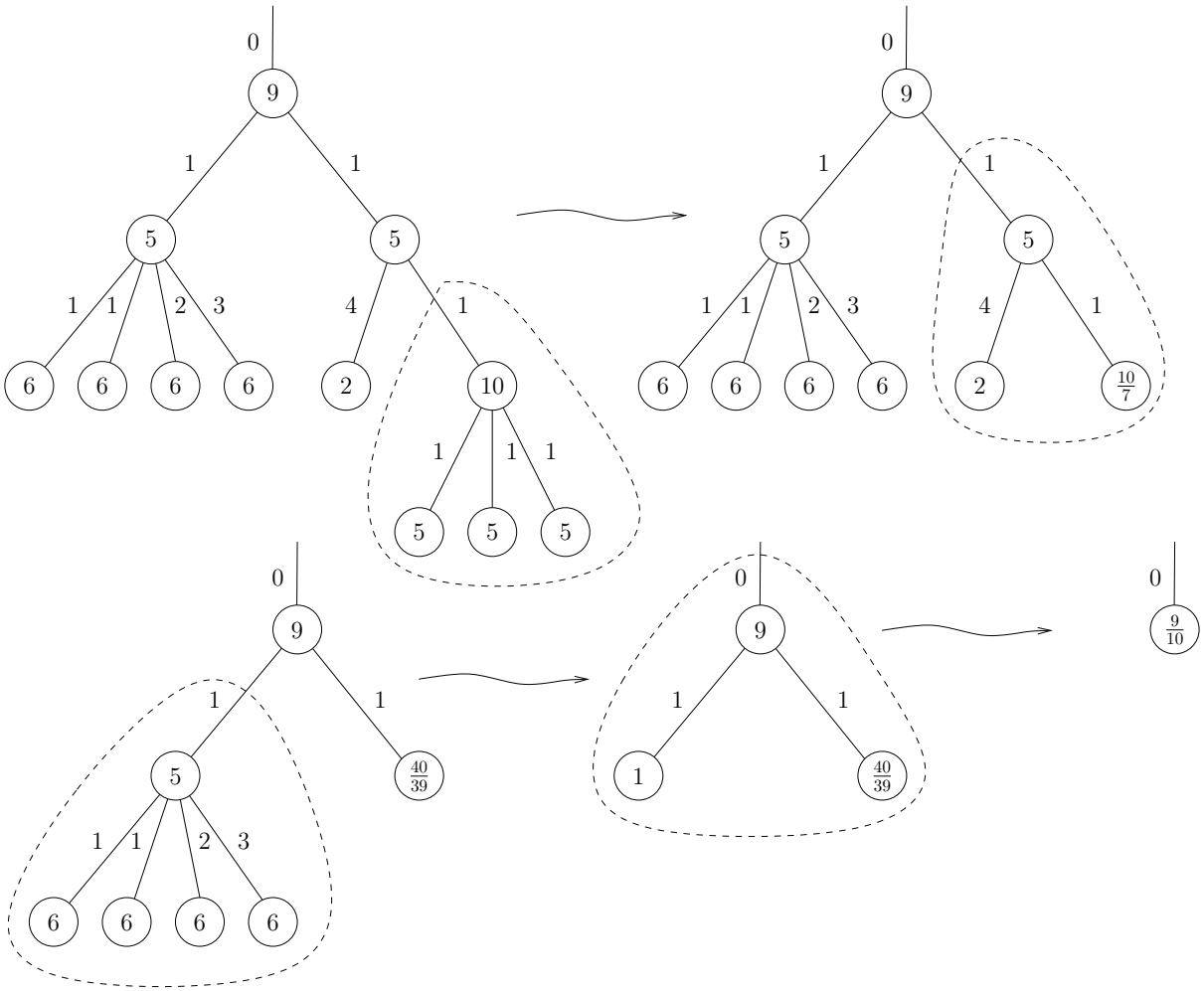


FIG. 4.9 – Calcul du régime permanent optimal à l'aide du théorème 4.2.

$\frac{1}{\rho_{\text{opt}}(F)}$ où $\rho_{\text{opt}}(F)$ est calculé à l'aide du lemme 4.3. En supposant que la racine a un parent virtuel de capacité infinie (c'est-à-dire telle que $c_{-1} = 0$), on obtient le résultat. ■

La figure 4.9 donne une illustration de l'application de ce théorème.

4.4.4 Ordonnancement

À la différence de la section 4.3.5, nul besoin de recourir à une décomposition en couplage pour construire un ordonnancement valide. En effet, chaque processeur gère lui-même ses émissions avec ses esclaves et ne peut rentrer en conflit (c'est-à-dire essayer de communiquer avec un esclave qui est déjà en train de recevoir des données d'une autre personne) avec qui que ce soit. Si on note $\alpha_i = \frac{a_i}{b_i}$ et $s_{i,j} = \frac{p_{i,j}}{q_{i,j}}$, il suffit donc de prendre le plus petit commun multiple des dénominateurs des $b_i \times w_i$ et des $q_{i,j} \times c_{i,j}$ pour obtenir

la durée de la période et en déduire un ordonnancement réalisant le régime permanent ainsi calculé.

4.5 Résultats de complexité

Dans cette section, nous établissons un certain nombre de résultats de complexité. Dans la section 4.5.1, nous montrons que les ordonnancements basé sur le calcul du régime permanent sont asymptotiquement optimaux. En section 4.5.2, nous expliquons comment modifier les équations du programme linéaire 4.9 pour prendre en compte d'autres modèles de communication. Enfin, en section 4.5.3, nous montrons que l'extraction de la meilleur arborescence couvrante est NP complet et inapproximable.

4.5.1 Optimalité asymptotique

Dans cette section, nous montrons l'optimalité asymptotique des ordonnancements présentés en section 4.3 et 4.4. Soit $G = (V, E, w, c)$ un graphe de plate-forme et une borne de temps K. Notons $n_{\text{opt}}(G, K)$ le nombre optimal de tâches traitées en un temps K sur la plate-forme G. On a le résultat suivant :

Lemme 4.5. $n_{opt}(G, K) \leqslant \rho_{opt} \times K$

Démonstration. Considérons un ordonnancement optimal. Notons $sent^\star(P_i \to P_j, e_{k,l})$ le nombre de fichiers de type $e_{k,l}$ que P_i envoie à P_j et $cons^\star(P_i, T_k)$ le nombre de tâches de type T_k calculées par P_i dans cet ordonnancement. De même, on définit $s^\star(P_i \to P_j, e_{k,l})$ le temps passé par P_i à envoyer des fichiers de type $e_{k,l}$ vers P_j et $\alpha^\star(P_i, T_k)$ le temps passé par P_i à calculer des tâches de type T_k. On a donc les équations suivantes :

- le processeur P_i traite $cons^\star(P_i, T_k)$ tâches de type T_k en temps K :

$$\alpha^\star(P_i, T_k) = cons^\star(P_i, T_k) \times w_{i,k} \leqslant K$$

- le processeur P_i reçoit $sent^\star(P_i \to P_j, e_{k,l})$ fichiers de type $e_{k,l}$ de la part de P_j et y passe un temps $s^\star(P_i \to P_j, e_{k,l})$:

$$s^\star(P_i \to P_j, e_{k,l}) = sent^\star(P_i \to P_j, e_{k,l}) \times data_{k,l} \times c_{i,j}$$

- Pour tout P_i les émissions sont séquentielles :

$$\sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s^\star(P_i \to P_j, e_{k,l}) \leqslant K$$

- Pour tout P_i les réceptions sont séquentielles :

$$\sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s^\star(P_j \to P_i, e_{k,l}) \leqslant K$$

– Les fichiers n'apparaissent et ne disparaissent pas dans la nature. On a donc pour tout P_i et tout type de fichier $e_{k,l} : T_k \to T_l$:

$$\sum_{P_j \to P_i} sent^\star(P_j \to P_i, e_{k,l}) + cons^\star(P_i, T_k) = \sum_{P_i \to P_j} sent^\star(P_i \to P_j, e_{k,l}) + cons^\star(P_i, T_l)$$

En définissant

$$\begin{cases} \alpha(P_i, T_k) = \alpha^\star(P_i, T_k)/K \\ cons(P_i, T_k) = cons^\star(P_i, T_k)/K \\ s(P_i \to P_j, e_{k,l}) = s^\star(P_i \to P_j, e_{k,l})/K \\ sent(P_i \to P_j, e_{k,l}) = sent^\star(P_i \to P_j, e_{k,l})/K \end{cases},$$

on obtient des variables vérifiant les contraintes du programme linéaire 4.9. On a donc $\sum_{i=1}^{p} cons(P_i, T_{end}) \leqslant \rho_{\text{opt}}$ et donc

$$n_{\text{opt}}(G, K) = \sum_{i=1}^{p} cons^\star(P_i, T_{end}) \leqslant \rho_{\text{opt}} \times K \qquad \blacksquare$$

Le lemme 4.5 dit simplement qu'aucun ordonnancement ne peut exécuter plus de tâches que le régime permanent. On ne peut pas se placer tout de suite en régime permanent car tous les fichiers ne sont pas nécessairement disponibles. Il y a donc une période d'initialisation avant de rentrer en régime permanent. Notons qu'à partir d'un certain moment, il ne sert à rien de commencer le traitement de certaines tâches car il ne sera pas possible de les terminer à temps. On a donc une phase d'initialisation et une phase de nettoyage. L'initialisation de l'algorithme avant d'arriver en régime permanent et proposée en section 4.3.5 est indépendante de K et vaut donc une certaine valeur I, indépendante de T et ne dépendant que de la plate-forme G et de T_p. En effet, le temps nécessaire à l'envoi à chaque processeur P_i de $cons(P_i, T_k)T_p$ tâches T_k ne dépend que des caractéristiques de G et de la valeur de T_p. De même le temps de la phase de nettoyage ne dépend que de T_p et des caractéristiques de la plate-forme et vaut une certaine valeur J. Le reste du temps peut donc se passer en régime permanent et on a donc

$$\begin{aligned} n_{\text{steady-state}}(G, K) &\geqslant \rho_{\text{opt}} T_p \left\lfloor \frac{K - I - J}{T_p} \right\rfloor \\ &\geqslant \rho_{\text{opt}} T_p \left(\frac{K - I - J}{T_p} - 1 \right) \\ &\geqslant \rho_{\text{opt}} K - \rho_{\text{opt}}(I + J + T_p) \\ &\geqslant n_{\text{opt}}(G, K) - \rho_{\text{opt}}(I + J + T_p), \end{aligned}$$

où $n_{\text{steady-state}}$ est le nombre de tâches exécutées par l'ordonnancement de la section 4.3.5 en temps K. En temps T, l'ordonnancement de la section 4.3.5 basé sur le régime permanent optimal n'exécute donc qu'un nombre borné (indépendant de T) de tâches de moins que l'optimal. On peut en déduire le résultat suivant :

Théorème 4.3. *L'ordonnancement de la section 4.3.5 basé sur le régime permanent optimal est asymptotiquement optimal :*

$$\lim_{K\to+\infty} \frac{n_{steady\text{-}state}(G,K)}{n_{opt}(G,K)} = 1.$$

Démonstration. En effet, on a

$$\begin{aligned}\frac{n_{\text{steady-state}}(G,K)}{n_{\text{opt}}(G,K)} &\geqslant \rho_{\text{opt}} T_p \left\lfloor \frac{K-I-J}{T_p} \right\rfloor \frac{1}{n_{\text{opt}}(G,K)} \\ &\geqslant \rho_{\text{opt}} T_p \left(\frac{K-I-J}{T_p} - 1 \right) \left(\frac{1}{\rho_{\text{opt}} K} \right) \xrightarrow[K\to+\infty]{} 1,\end{aligned}$$

ce qui établit le résultat. ■

4.5.2 Extension à d'autres modèles

Dans cette section, nous proposons d'autres modèles de communication que le modèle «1-port en entrée, 1-port en sortie et recouvrement» que nous utilisons dans ce chapitre. Pour chacun de ces modèles, nous expliquons comment modifier les équations du programme (4.9) et comment adapter le lemme 4.3.

Nous commençons par énumérer un certain nombre de modèles, en partant des machines aux meilleurs capacités et en finissant avec les machines purement séquentielles.

$\mathcal{M}(r^*\|s^*\|w)$ **: Multi-port avec recouvrement** Dans ce modèle, un processeur peut simultanément recevoir des données de tous ses voisins, effectuer des calculs (indépendants des données qu'il est en train de recevoir), et envoyer des données à chacun de ses voisins. Ce modèle n'est évidemment plus réaliste dès que le nombre de voisins devient trop important.

$\mathcal{M}(r\|s\|w)$ **: 1-port en émission et 1 port en réception, avec recouvrement** Dans ce deuxième modèle, un processeur peut à la fois recevoir des données d'un de ses voisins, envoyer des données à un voisin et effectuer des calculs. À chaque étape, chaque processeur est donc impliqué dans au plus deux communication, une émission et une réception. C'est le modèle de base que nous utilisons dans cette thèse. Il est relativement représentatif d'un certain nombre de machines modernes et il n'est pas difficile de réduire les autres modèles à celui-ci.

$\mathcal{M}(w\|r,s)$ **: Travail en parallèle, 1-port** Dans ce troisième modèle, comme dans les deux prochains, un processeur n'a qu'un seul niveau de parallélisme : un processeur peut effectuer des calculs tout en effectuant soit une émission, soit une réception. Ce modèle est également assez représentatif des capacités de certaines machines actuelles.

$\mathcal{M}(r\|s,w)$ **: Réception en parallèle, 1-port** Dans ce quatrième modèle, un processeur peut recevoir des données d'un voisin tout en effectuant soit un calcul, soit une émission de données vers un processeur.

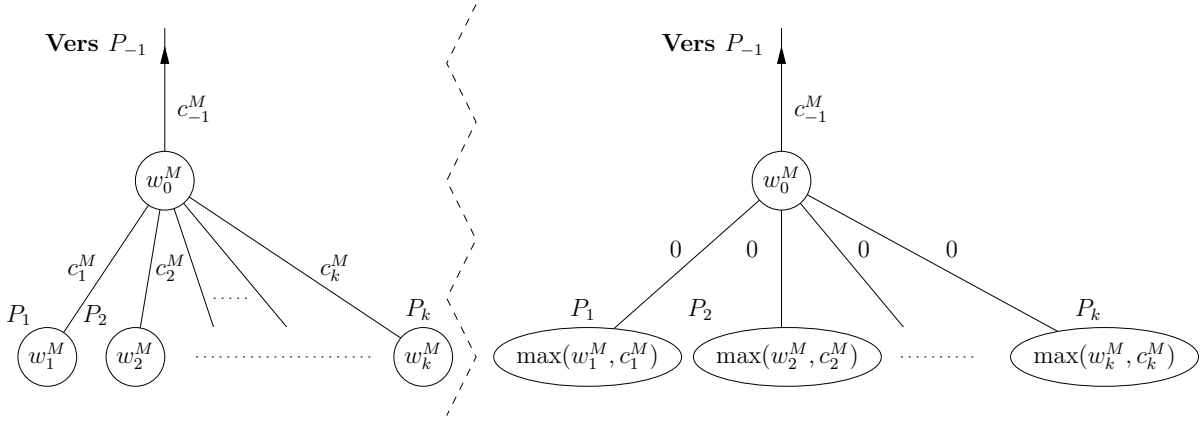


FIG. 4.10 – Réduction pour les arborescences du modèle multiport $\mathcal{M}(r^*\|s^*\|w)$ au modèle de base

$\mathcal{M}(s\|r,w)$ **: Émission en parallèle, 1-port** Dans ce cinquième modèle, un processeur peut envoyer des données à un voisin tout en effectuant soit un calcul, soit une réception de données.

$\mathcal{M}(r,s,w)$ **: Pas de parallélisme** Dans ce dernier modèle, un processeur ne peut faire qu'une seule chose à la fois : soit recevoir des données, soit effectuer un calcul, soit envoyer des données.

4.5.2.1 Réduction de $\mathcal{M}(r^*\|s^*\|w)$ au modèle de base

Dans ce modèle, un nombre illimité de communications peut avoir lieu en même temps. Ce modèle est assez simple à prendre en compte puisqu'il suffit de supprimer les contraintes (4.5) et (4.6) dans le programme linéaire. En effet, les contraintes (4.4) suffisent alors pour caractériser l'activité du processeur. Une fois le programme linéaire résolu, il n'y a plus besoin de passer par une décomposition en couplage comme dans la section 4.3.5 pour obtenir un ordonnancement puisqu'il n'y a plus aucun conflit de communications.

Proposition 4.1. *Dans le cas où on se restreint aux arborescences (comme dans la section 4.4), il est toujours possible d'utiliser le lemme 4.3 si on effectue préalablement la transformation représentée figure 4.10.*

Démonstration. Rappelons que les contraintes dans le modèle de base s'écrivaient (voir

contraintes (4.14)) :

$$\begin{array}{l}\text{MAXIMISER } \rho = \sum_{i=0}^{k} R_i, \\ \text{SOUS LES CONTRAINTES} \\ \begin{cases} \text{(4.16a)} & R_{-1} = \displaystyle\sum_{i=0}^{k} R_i \\ \text{(4.16b)} & R_{-1} c_{-1}^B \leqslant 1 \\ \text{(4.16c)} & \forall i \in [\![0,k]\!], R_i w_i^B \leqslant 1 \\ \text{(4.16d)} & \displaystyle\sum_{i=1}^{k} R_i c_i^B \leqslant 1 \end{cases} \end{array} \tag{4.16}$$

Dans le modèle $\mathcal{M}(r^*\|s^*\|w)$, les contraintes s'écrivent

$$\begin{array}{l}\text{MAXIMISER } \rho = \sum_{i=0}^{k} R_i, \\ \text{SOUS LES CONTRAINTES} \\ \begin{cases} \text{(4.17a)} & R_{-1} = \displaystyle\sum_{i=0}^{k} R_i \\ \text{(4.17b)} & R_{-1} c_{-1}^M \leqslant 1 \\ \text{(4.17c)} & \forall i \in [\![0,k]\!], R_i w_i^M \leqslant 1 \\ \text{(4.17d)} & \forall i \in [\![1,k]\!], R_i c_i^M \leqslant 1 \end{cases} \end{array} \tag{4.17}$$

Si on dispose d'un problème énoncé dans le modèle $\mathcal{M}(r^*\|s^*\|w)$, en définissant $c_{-1}^B = c_{-1}^M$, $w_0^B = w_0^M$, et pour $i \in [\![1,k]\!]$, $c_i^B = 0$ et $w_i^B = \max(w_i^M, c_i^M)$ (voir figure 4.10), on va montrer que les deux programmes linéaires sont équivalents.

En effet, l'équation (4.16d) est tout le temps vraie et les contraintes (4.17c) et (4.17d) peuvent alors s'écrire $R_i \max(w_i^M, c_i^M) \leqslant 1$, soit $R_i w_i^B \leqslant 1$. Les contraintes (4.17c) et (4.17d) sont donc équivalentes à la contrainte (4.16c). On peut enfin remarquer que (4.17a) et (4.17b) sont clairement équivalentes à (4.16a) et (4.16b), ce qui montre bien l'équivalence entre ces deux systèmes. ■

4.5.2.2 Réduction de $\mathcal{M}(w\|r,s)$ au modèle de base

Dans ce modèle, chaque processeur peut calculer en parallèle avec ses communications mais au plus une communication (que ce soit en émission ou en réception) peut avoir lieu à un instant donné. Pour prendre ces nouvelles contraintes en compte, il est donc nécessaire de remplacer les contraintes (4.5) et (4.6) par la suivante :

$$\forall P_i, \sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) + \sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1$$

Nous verrons plus tard que cette substitution n'est hélas pas suffisante pour calculer le régime permanent optimal sur une plate-forme générale et qu'elle ne nous permet que

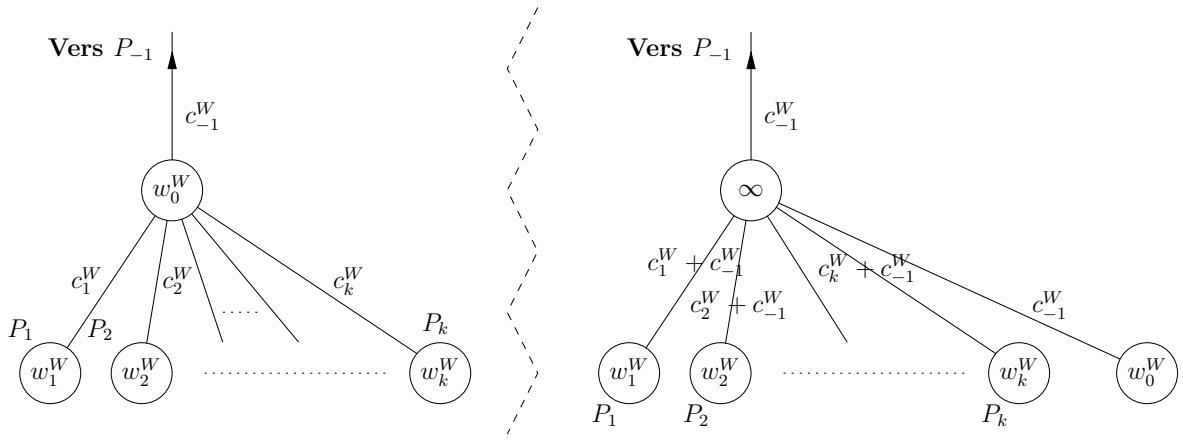


FIG. 4.11 – Réduction pour les arborescences du modèle $\mathcal{M}(w\|r,s)$ au modèle de base

d'obtenir une borne supérieure de ce débit. En revanche, sur une arborescence, cette substitution est suffisante :

Proposition 4.2. *Dans le cas où on se restreint aux arborescences (comme dans la section 4.4), il est toujours possible d'utiliser le lemme 4.3 si on effectue préalablement la transformation représentée figure 4.11.*

Démonstration. Dans le modèle $\mathcal{M}(w\|r,s)$, le programme linéaire (4.16) définissant le régime permanent devient :

$$\begin{array}{l}\text{MAXIMISER } \rho = \sum_{i=0}^{k} R_i^W, \\ \text{SOUS LES CONTRAINTES} \\ \left\{\begin{array}{ll} \text{(4.18a)} & R_{-1}^W = \displaystyle\sum_{i=0}^{k} R_i^W \\ \text{(4.18b)} & R_{-1}^W c_{-1}^W \leqslant 1 \\ \text{(4.18c)} & \forall i \in [\![0,k]\!], R_i^W w_i^W \leqslant 1 \\ \text{(4.18d)} & R_{-1}^W c_{-1}^W + \displaystyle\sum_{i=1}^{k} R_i^W c_i^W \leqslant 1 \end{array}\right. \end{array} \tag{4.18}$$

Si on dispose d'un problème énoncé dans le modèle $\mathcal{M}(w\|r,s)$, il suffit de définir l'instance suivante dans le modèle de base (voir figure 4.11) :

$$\begin{cases} c_{-1}^B = c_{-1}^W & \text{et } w_0^B = \infty \\ c_{k+1}^B = c_{-1}^W & \text{et } w_{k+1}^B = w_0^W \\ c_i^B = c_i^W + c_{-1}^W & \text{et } w_i^B = w_i^W \text{ pour } i \in [\![1,k]\!] \end{cases}$$

On peut alors montrer que l'on peut passer d'un système d'équations à l'autre en posant $R_i = R_i^W$ pour $i \in [\![1,k]\!]$ et $R_{k+1} = R_0^W$. R_0 est une variable dans le modèle

de base qui est contrainte par $R_0 \leqslant \frac{1}{w_0^B}$, ce qui signifie que $R_0 = 0$. Elle n'a donc pas d'importance pour passer d'un système à l'autre ni d'impact sur la fonction objectif.

En substituant ces valeurs dans les contraintes (4.18b) et (4.18c) on obtient :

$$R_{-1} = \frac{1}{c_{-1}^B}$$
$$R_i \leqslant \frac{1}{w_i^B} \text{ pour } i \in [\![1, k+1]\!]$$

Ces contraintes, sont donc exactement les contraintes (4.16b) et (4.16c) (on a vu que R_0 n'avait pas d'importance). Pour obtenir (4.16d), il suffit de réécrire la partie gauche de (4.18d) :

$$\begin{aligned} R_{-1}^W c_{-1}^W + \sum_{i=1}^{k} R_i^W c_i^W &= \sum_{i=0}^{k} R_i^W c_{-1}^B + \sum_{i=1}^{k} R_i^W \underbrace{(c_i^B - c_{-1}^B)}_{c_i^W} \\ &= \sum_{i=1}^{k} R_i^W c_i^B + R_0^W c_{-1}^B \\ &= \sum_{i=1}^{k} R_i^W c_i^B + R_{k+1} c_{k+1}^B \\ &= \sum_{i=0}^{k+1} R_i c_i^B. \end{aligned}$$

Enfin (4.18a) se réécrit

$$R_{-1}^W = \sum_{i=0}^{k} R_i^W = \sum_{i=0}^{k+1} R_i = R_{-1},$$

ce qui nous donne bien (4.16a), et montre ainsi l'équivalence entre ces deux programmes linéaires. ■

Intuitivement, la proposition précédante est vraie car le problème de la résolution des conflits présenté en section 4.3.5 n'a lieu d'être que si un processeur peut recevoir des données de différentes personnes, ce qui n'est pas le cas dans une arborescence.

Dans le cas général, il n'est en revanche pas toujours possible de transformer le graphe des communications en un graphe biparti, notamment dans les modèles $\mathcal{M}(s\|r, w)$, $\mathcal{M}(r\|s, w)$,$\mathcal{M}(w\|r, s)$ et $\mathcal{M}(r, s, w)$. En effet, en dédoublant les processeurs en une partie émettrice et une partie réceptrice, nous modélisions le fait que les émissions et les réceptions n'interféraient pas. Dans le cas contraire, le graphe n'est plus biparti et on n'a

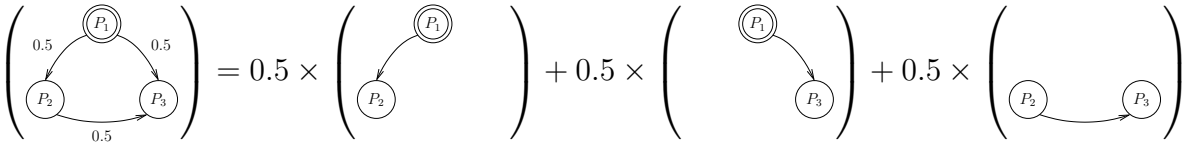


FIG. 4.12 – Une solution non atteignable

pas l'assurance de pouvoir le décomposer et de pouvoir reconstruire un ordonnancement atteignant effectivement le régime permanent.

Supposons par exemple que le graphe de communications retourné par le programme linéaire dans le modèle $\mathcal{M}(w\|r,s)$ (c'est-à-dire qu'il est possible de recouvrir les communications par du calcul mais que les communications sont toutes séquentielles) est celui représenté en figure 4.12. La somme des poids des arêtes entrantes et sortantes est égale à 1 en chaque nœud (c'est donc bien un graphe de communications valide) et il est impossible de le décomposer en une somme pondérée de couplage telle que la somme des poids est inférieure ou égale à 1. En effet, les trois arêtes doivent chacune être dans un couplage différent puisqu'elles ont toutes au moins un sommet en commun. La somme des poids de toute décomposition est donc necéssairement égale à 1.5, ce qui ne permet pas de reconstituer un motif valide.

On peut remarquer que l'on sait décomposer le graphe de communications si et seulement si on arrive à trouver un ordonnancement valide associé aux allocations déduites du programme linéaire. Cependant rien n'empêche, *a priori*, que d'autres allocations conduisent à un graphe de communications décomposable et donc permettant d'atteindre le régime permanant ainsi calculé.

Ce résultat peut paraître négatif puisque pour un certain nombre de modèles (c'est-à-dire tous ceux où les réceptions ne s'effectuent pas en même temps que les émissions), il n'est pas toujours possible d'exhiber un ordonnancement réalisant le régime permanent calculé en section 4.3 mais uniquement de donner une borne supérieure. Avec ce modèle, le calcul du régime permanent optimal reste donc une question ouverte. Nous reviendrons sur cette limitation en section 4.5.2.6.

4.5.2.3 Réduction de $\mathcal{M}(r\|s,w)$ au modèle de base

Ce modèle est moins puissant que le modèle de base : chaque processeur peut simultanément recevoir une tâche d'un de ses voisins et, soit calculer, soit envoyer des données à un voisin. Pour prendre ces nouvelles contraintes en compte, il suffit de remplacer les contraintes (4.5) et (4.7) par la suivante :

$$\forall P_i, \sum_{T_k \in V_A} \alpha(P_i, T_k) + \sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1$$

Proposition 4.3. *Dans le cas où on se restreint aux arborescences (comme dans la section 4.4), il est toujours possible d'utiliser le lemme 4.3 si on effectue préalablement la transformation représentée figure 4.13.*

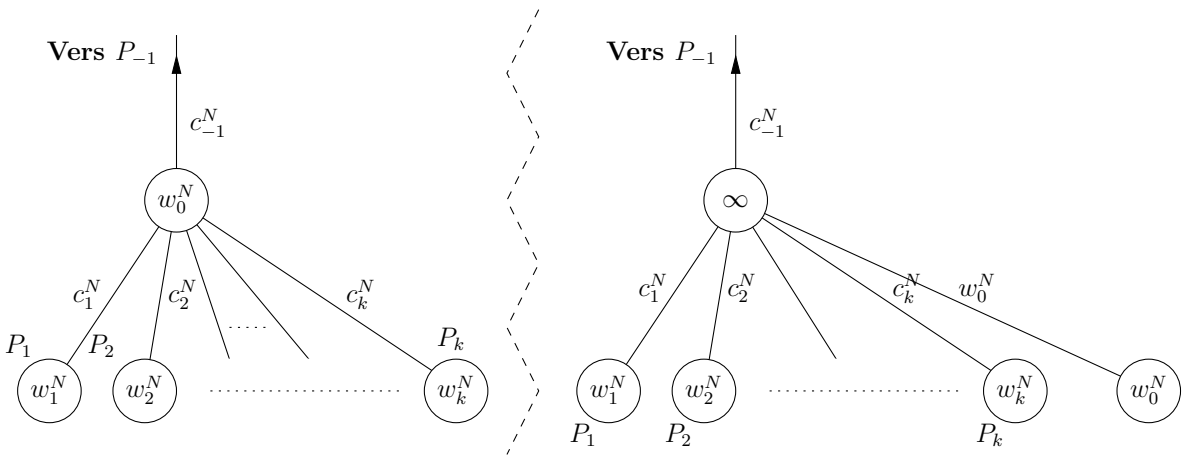


FIG. 4.13 – Réduction pour les arborescences du modèle $\mathcal{M}(r\|s,w)$ ou $\mathcal{M}(r,s,w)$ au modèle de base

Démonstration. Dans le modèle $\mathcal{M}(r\|s,w)$, le programme linéaire (4.16) définissant le régime permanent devient :

$$\begin{array}{l}\text{MAXIMISER } \rho = \sum_{i=0}^{k} R_i^N, \\ \text{SOUS LES CONTRAINTES} \\ \left\{\begin{array}{ll} \text{(4.19a)} & R_{-1}^N = \displaystyle\sum_{i=0}^{k} R_i^N \\ \text{(4.19b)} & R_{-1}^N c_{-1}^N \leqslant 1 \\ \text{(4.19c)} & \forall i \in [\![0,k]\!], R_i^N w_i^N \leqslant 1 \\ \text{(4.19d)} & \displaystyle\sum_{i=1}^{k} R_i^N c_i^N + R_0^N w_0^N \leqslant 1 \end{array}\right. \end{array} \tag{4.19}$$

Si on dispose d'un problème énoncé dans le modèle $\mathcal{M}(r\|s,w)$, il suffit de définir l'instance suivante dans le modèle de base (voir figure 4.13) :

$$\begin{cases} c_{-1}^B = c_{-1}^N & \text{et } w_0^B = \infty \\ c_{k+1}^B = w_0^N & \text{et } w_{k+1}^B = w_0^N \\ c_i^B = c_i^N & \text{et } w_i^B = w_i^N \text{ pour } i \in [\![1,k]\!] \end{cases}$$

On peut alors montrer que l'on peut passer d'un système d'équations à l'autre en posant $R_i = R_i^N$ pour $i \in [\![1,k]\!]$ et $R_{k+1} = R_0^N$. Comme précédemment, R_0 est une variable dans le modèle de base qui est contrainte par $R_0 \leqslant \frac{1}{w_0^B}$, ce qui signifie que $R_0 = 0$ et qu'elle n'a donc pas d'importance pour passer d'un système à l'autre ni d'impact sur la fonction objectif.

Pour obtenir (4.16d), il suffit de réécrire le membre gauche de (4.19d),

$$\sum_{i=1}^{k} R_i^N c_i^N + R_0^N w_0^N = \sum_{i=1}^{k} R_i c_i^B + R_{k+1} w_{k+1}^B = \sum_{i=1}^{k+1} R_i c_i^B$$

Les autres contraintes se transposent immédiatement, ce qui montre bien l'équivalence entre ces deux modèles. ■

Comme dans le modèle précédent, il est aisé de déduire un ordonnancement réalisant effectivement le débit optimal pour ce modèle si la plate-forme est un arbre. Cependant, à la différence du modèle précédant, il est possible d'émettre et de recevoir en même temps et le graphe des communications se transforme donc bien en un graphe biparti. Il est donc possible d'ordonnancer les communications. En ce qui concerne l'interférence avec le calcul, il suffit de ne pas en tenir compte et d'effectuer la décomposition du graphe de communications comme d'habitude. En effet, une fois les communications ordonnancées, les contraintes du programme linéaire nous assurent que chaque processeur aura suffisamment de temps libre pour effectuer les calculs dont il a la charge. On peut donc retrouver un ordonnancement valide asymptotiquement optimal avec ce modèle.

4.5.2.4 Réduction de $\mathcal{M}(s\|r,w)$ au modèle de base

Dans ce modèle, chaque processeur peut simultanément envoyer une tâche à un de ses voisins et, soit calculer, soit recevoir des données d'un voisin. Pour prendre ces nouvelles contraintes en compte, il suffit de remplacer les contraintes (4.6) et (4.7) par la suivante :

$$\forall P_i, \sum_{T_k \in V_A} \alpha(P_i, T_k) + \sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) \leqslant 1$$

Proposition 4.4. *Dans le cas où on se restreint aux arborescences (comme dans la section 4.4), il est toujours possible d'utiliser le lemme 4.3 si on effectue préalablement la transformation représentée figure 4.14.*

Démonstration. Dans le modèle $\mathcal{M}(s\|r,w)$, le programme linéaire (4.16) définissant le

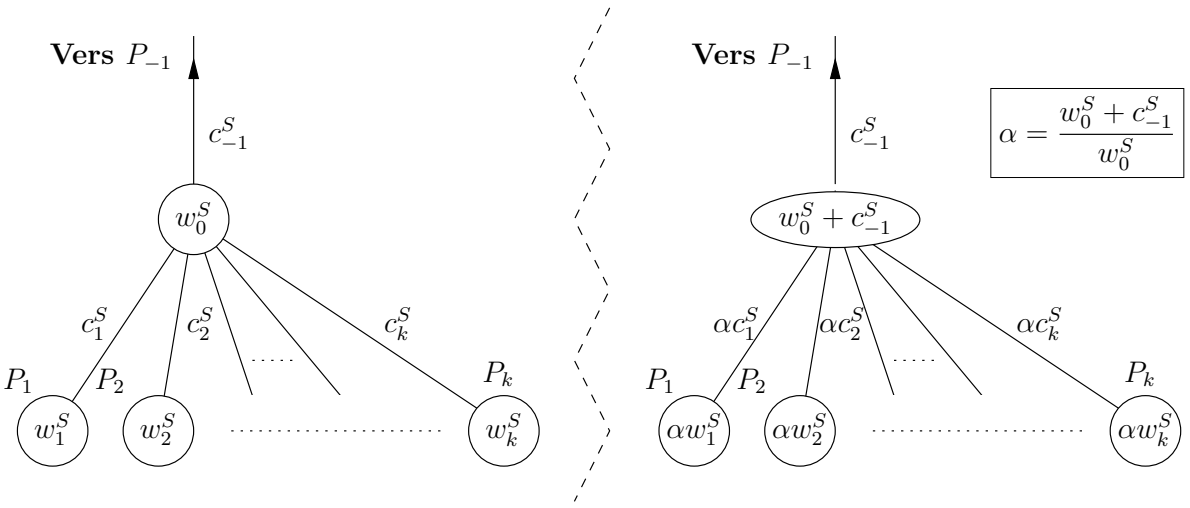


FIG. 4.14 – Réduction pour les arborescences du modèle $\mathcal{M}(s\|r,w)$ au modèle de base

régime permanent devient :

$$\begin{array}{l}\text{MAXIMISER } \rho = \sum_{i=0}^{k} R^S_i, \\ \text{SOUS LES CONTRAINTES} \\ \left\{\begin{array}{ll} \text{(4.20a)} & R^S_{-1} = \displaystyle\sum_{i=0}^{k} R^S_i \\ \text{(4.20b)} & R^S_{-1} c^S_{-1} \leqslant 1 \\ \text{(4.20c)} & \forall i \in [\![1,k]\!], R^S_i w^S_i \leqslant 1 \\ \text{(4.20d)} & \displaystyle\sum_{i=1}^{k} R^S_i c^S_i \leqslant 1 \\ \text{(4.20e)} & R^S_{-1} c^S_{-1} + R^S_0 w^S_0 \leqslant 1 \end{array}\right. \end{array} \tag{4.20}$$

Si on dispose d'un problème énoncé dans le modèle $\mathcal{M}(s\|r,w)$, il suffit de définir l'instance suivante dans le modèle de base (voir figure 4.13) :

$$\begin{cases} \alpha = \frac{w^S_0 + c^S_{-1}}{w^S_0} & \\ c^B_{-1} = c^S_{-1} & \text{et } w^B_0 = w^S_0 + c^S_{-1} \\ c^B_i = \alpha c^S_i & \text{et } w^B_i = \alpha w^S_i \text{ pour } i \in [\![1,k]\!] \end{cases}$$

On peut passer des contraintes (4.20) aux contraintes (4.16) en posant

$$\begin{cases} R_{-1} = R^S_{-1} \\ R_0 = R^S_0 + \left(1 - \frac{1}{\alpha}\right) \sum_{i=1}^{k} R^S_i \\ R_i = \frac{R^S_i}{\alpha} \text{ pour } i \in [\![1,k]\!]. \end{cases}$$

Commençons par remarquer que

$$1 - \frac{1}{\alpha} = 1 - \frac{w_0^S}{w_0^S + c_{-1}^S} = \frac{c_{-1}^S}{w_0^S + c_{-1}^S}$$

Commençons par la contrainte (4.20a). Elle se réécrit :

$$\begin{aligned} R_{-1}^S &= \sum_{i=0}^{k} R_i^S = R_0^S + \sum_{i=1}^{k} R_i^S \\ &= R_0 - \left(1 - \frac{1}{\alpha}\right) \sum_{i=1}^{k} \alpha R_i + \sum_{i=1}^{k} \alpha R_i = R_0 + \sum_{i=1}^{k} R_i, \end{aligned}$$

soit $R_{-1} = \sum_{i=0}^{k} R_i$, ce qui nous donne l'équation (4.16a).

Comme $R_{-1}^S c_{-1}^S = R_1 c_1$, on a équivalence entre la contrainte (4.20b) et la contrainte (4.16b). De même, comme $R_i^S w_i^S = \alpha R_i \frac{1}{\alpha} w_i^B = R_i w_i^B$, la contrainte (4.20c) et la contrainte (4.16c) sont équivalentes pour $i \in [\![1, k]\!]$. La réécriture de la contrainte (4.20e) nous donne le cas manquant.

$$\begin{aligned} R_0^S w_0^S + R_{-1}^S c_{-1}^S &= R_0^S w_0^S + c_{-1}^S \left(\sum_{i=0}^{k} R_i^S \right) \text{ en utilisant (4.20a)} \\ &= (w_0^S + c_{-1}^S) R_0^S + c_{-1}^S \left(\sum_{i=1}^{k} R_i^S \right) \\ &= (w_0^S + c_{-1}^S) R_0^S + (w_0^S + c_{-1}^S) \left(\frac{c_{-1}^S}{w_0^S + c_{-1}^S} \right) \left(\sum_{i=1}^{k} R_i^S \right) \\ &= (w_0^S + c_{-1}^S) \left(R_0^S + \left(1 - \frac{1}{\alpha}\right) \sum_{i=1}^{k} R_i^S \right) \\ &= (w_0^S + c_{-1}^S) R_0 = w_0^B R_0. \end{aligned}$$

Enfin, comme on a vu que pour $i \in [\![1, k]\!]$, on avait $R_i^S c_i^S = R_i c_i^B$, l'équivalence entre (4.20d) et la contrainte (4.16d) est immédiate, ce qui achève la démonstration. ■

Comme dans le modèle précédent, on peut déduire un ordonnancement réalisant effectivement le débit optimal dans tous les cas puisqu'il est possible d'émettre et de recevoir en même temps.

4.5.2.5 Réduction de $\mathcal{M}(r, s, w)$ au modèle de base

Enfin, dans le dernier modèle où il n'y a aucun parallélisme interne, il suffit de remplacer les contraintes (4.5), (4.6) et (4.7) par la suivante :

$$\forall P_i, \sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) + \sum_{T_k \in V_A} \alpha(P_i, T_k) + \sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1$$

Proposition 4.5. *Dans le cas où on se restreint aux arborescences (comme dans la section 4.4), il est toujours possible d'utiliser le lemme 4.3 si on effectue préalablement la transformation représentée figure 4.13.*

Démonstration. La démonstration est similaire à celle de la proposition 4.3, à ceci près que la réduction s'opère vers les équations (4.18) qui décrivent le modèle $\mathcal{M}(w\|r, s)$ et pas directement vers le modèle de base. ■

Comme avec le modèle $\mathcal{M}(w\|r, s)$, il est aisé de déduire un ordonnancement asymptotiquement optimal avec ce modèle si la plate-forme est un arbre mais il n'est pas toujours possible de décomposer le graphe des communications et donc de pouvoir reconstruire un ordonnancement atteignant effectivement ce régime permanent dans le cas d'un graphe général.

4.5.2.6 Application à une plateforme réelle

Nous avons donc vu qu'il n'est pas toujours possible de transformer le graphe des communications en un graphe biparti et que l'on n'a donc pas l'assurance de pouvoir le décomposer et de pouvoir reconstruire un ordonnancement atteignant effectivement le régime permanent. Cependant, ce n'est pas vraiment un problème en pratique car les routeurs sont généralement capables de traiter plusieurs communications en même temps. Si le graphe des communications ne peut être transformé en un graphe biparti, cela signifie qu'il existe dans le réseau un cycle (de longueur impaire) constitué de machines incapables d'effectuer des communications en parallèle. Il est quand même rare de trouver des boucles de *hubs* et les machines traitant séquentiellement leurs communications sont donc généralement des feuilles du graphe d'interconnexion. Il est alors possible de décomposer le graphe des communications.

4.5.3 Arborescence recouvrante

Dans le cas d'un réseau d'interconnexion modélisé par un graphe général, notre programme linéaire peut conduire à une solution utilisant plusieurs chemins pour atteindre un même processeur (ce qui est le cas de l'exemple de la section 4.3.5). Il peut cependant être intéressant de travailler avec un arbre pour différentes raisons. En effet, une vision arborescente de la plate-forme simplifie grandement une mise en œuvre réelle et l'approche

orientée bande-passante de la section 4.4 conduit à des protocoles d'ordonnancement locaux et à la demande qui sont donc très robustes aux variations de charge [13, 31]. En effet, nous avons vu avec le théorème 4.2 qu'il est possible d'effectuer le calcul du régime permanent optimal et de l'ordonnancement qui atteint ce régime permanent en utilisant uniquement des informations locales. Si chaque hôte envoie régulièrement à son père une estimation de la bande passante du lien qui les relie et de sa puissance cumulée (c'est-à-dire de la puissance de son sous-arbre), ce dernier est alors capable de déterminer avec quelles priorités il doit répondre aux requêtes de ses différents fils pour atteindre le régime permanent optimal. De plus, il n'existe que trois classes de priorités selon que les nœuds doivent être utilisés complètement, partiellement ou pas du tout et ces informations de nature qualitative sont robustes aux petites variations des performances de la plate-forme. Il est donc naturel de se demander comment extraire la meilleure arborescence couvrante d'une plate-forme, c'est-à-dire celle permettant d'atteindre le meilleur débit en régime permanent.

Cette section établit des résultats plutôt négatifs : premièrement, l'extraction de la meilleure arborescence couvrante est NP-complet et deuxièmement, quand bien même on l'aurait trouvée, son débit peut être arbitrairement mauvais par rapport à celui de la plate-forme complète.

4.5.3.1 Extraction de la meilleure arborescence

Définissons formellement le problème de l'extraction de la meilleure arborescence couvrante d'une plate-forme quelconque.

Définition 4.12 (Spanning-Tree(G)). *Soit $G = (V, E, w, c)$ un graphe pondéré représentant l'architecture de la plate-forme. Trouver la meilleure arborescence couvrante $T = (V, E', w, c)$ de G telle que le débit $\rho_{opt}(T)$ est maximal parmi l'ensemble des arborescences couvrantes de G.*

Le problème de décision associé à ce problème d'optimisation est le suivant :

Définition 4.13 (Spanning-Tree-Dec(G, α)). *Soit $G = (V, E, w, c)$ un graphe pondéré représentant l'architecture de la plate-forme et α un réel strictement positif. Existe-t-il une arborescence couvrante $T = (V, E', w, c)$ de G dont le débit $\rho_{opt}(T)$ est supérieur à α ?*

Nous allons montrer le résultat suivant

Théorème 4.4. *Spanning-Tree-Dec est NP-complet.*

Démonstration. Spanning-Tree-Dec est clairement dans la classe NP. Nous démontrons sa NP-complétude par réduction à 2-Partition dont la NP-complétude est connue [58]. Considérons l'instance suivante de 2-Partition : soit $a_1, \ldots, a_n$ n entiers positifs. La question posée par 2-Partition est la suivante :

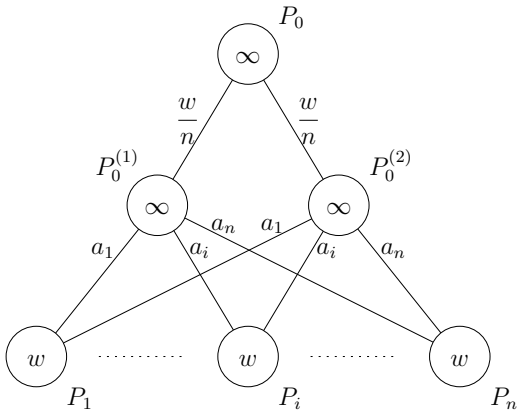


FIG. 4.15 – Instance de Spanning-Tree-Dec utilisée pour la réduction.

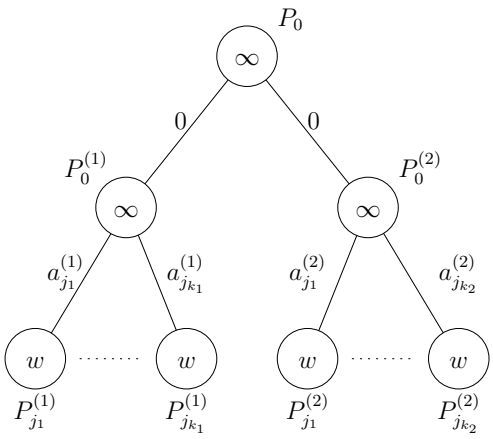


FIG. 4.16 – Description de l'arborescence T extraite de G.

Existe-t-il une partition de $[\![1, n]\!]$ en deux ensembles A_1, A_2 telle que $\sum_{i \in A_1} a_i = \sum_{i \in A_2} a_i$?

L'instance de Spanning-Tree-Dec que nous allons utiliser pour notre démonstration est constituée du graphe G représenté en figure 4.15. Le maître P_0 a deux voisins $P_0^{(1)}$ et $P_0^{(2)}$. Le temps de communication du maître vers ses deux voisins est nul. $P_0^{(1)}$ et $P_0^{(2)}$ ont le même ensemble de voisins $P_1, \ldots, P_n$ qui sont les seuls nœuds capables de traiter des tâches. Leur puissance de calcul w_i est homogène et égale à $w = \frac{1}{2}\sum_{i=1}^{n} a_i$. Le temps de communication de $P_0^{(1)}$ à P_i est identique à celui de $P_0^{(2)}$ à P_i et est égal à a_i. Enfin, on utilisera $\alpha = \frac{n}{w}$ comme borne. La taille de cette instance est clairement polynomiale (et même linéaire) en celle de l'instance originale de 2-Partition.

Montrons maintenant qu'il existe une solution à notre instance originale de 2-Partition si et seulement s'il existe une solution à notre instance de Spanning-Tree-Dec.

$\boxed{\Leftarrow}$ Supposons qu'il existe une solution T de Spanning-Tree-Dec telle que $\rho_{\text{opt}}(T) \geqslant \alpha$. Toute arborescence couvrante de G peut être représentée par une partition des feuilles P_i en fonction du père pour lequel on garde la connexion. On note donc A_1 (resp. A_2) les indices des esclaves qui sont connectés à $P_0^{(1)}$ (resp. $P_0^{(2)}$) dans T (voir figure 4.16).

En utilisant les résultats de la section 4.4, on peut calculer $\rho_{\text{opt}}(T)$. On note $j_1^{(i)} \leqslant \ldots \leqslant j_{k_i}^{(i)}$ l'ensemble des indices de A_i et on suppose que $a_{j_1^{(i)}} \leqslant \ldots \leqslant a_{j_{k_i}^{(i)}}$. Notons k_i' le plus grand indice de A_i, s'il existe, tel que

$$\sum_{k=1}^{k_i'} \frac{a_{j_k^{(i)}}}{w} \leqslant 1 \text{ et } \sum_{k=1}^{k_i'+1} \frac{a_{j_k^{(i)}}}{w} > 1.$$

Si un tel k_i' n'existe pas, on pose $k_i' = k_i$.

Alors le débit de la sous-arborescence T_i enracinée en $P_0^{(i)}$ pour $i \in \{1, 2\}$ est :

– si $k'_i = k_i$ alors $\rho_{\text{opt}}(T_i) = \frac{k_i}{w}$ (en utilisant le lemme 4.3).
– si $k'_i < k_i$ alors on a

$$\rho_{\text{opt}}(T_i) = \sum_{k=1}^{k'_i} \frac{1}{w} + \frac{1 - \sum_{k=1}^{k'_i} \frac{a_{j_k^{(i)}}}{w}}{a_{j_{k'_i+1}^{(i)}}},$$

et comme $\frac{a_{j_{k'_i+1}^{(i)}}}{w} + \sum_{k=1}^{k'_i} \frac{a_{j_k^{(i)}}}{w} > 1$, on a

$$\frac{1 - \sum_{k=1}^{k'_i} \frac{a_{j_k^{(i)}}}{w}}{a_{j_{k'_i+1}^{(i)}}} < \frac{1}{w}, \text{ et donc } \rho_{\text{opt}}(T_i) < \frac{k_i}{w}$$

On a donc toujours $\rho_{\text{opt}}(T_i) \leqslant \frac{k_i}{w}$ et $\rho_{\text{opt}}(T_i) = \frac{k_i}{w}$ si et seulement si $\sum_{k=1}^{k_i} \frac{a_{j_k^{(i)}}}{w} \leqslant 1$. Ainsi, comme $\rho_{\text{opt}}(T) = \rho_{\text{opt}}(T_1) + \rho_{\text{opt}}(T_2)$, on a $\rho_{\text{opt}}(T) \leqslant \frac{k_1}{w} + \frac{k_2}{w} = \frac{n}{w}$ et

$$\rho_{\text{opt}}(T) = \frac{n}{w} \text{ ssi } \left(\sum_{i \in A_1} a_i \leqslant w \text{ et } \sum_{i \in A_2} a_i \leqslant w \right)$$

Comme $\sum_{i \in A_1} a_i + \sum_{i \in A_2} a_i = 2w$ (par définition de w), on a

$$\rho_{\text{opt}}(T) = \frac{n}{w} \text{ ssi } \sum_{i \in A_1} a_i = \sum_{i \in A_2} a_i = w$$

On a donc montré que s'il existe une solution à notre instance de Spanning-Tree-Dec alors il existe une solution à l'instance originale de 2-Partition.

$\boxed{\Rightarrow}$ Pour la réciproque, il suffit vérifier que l'arborescence de la figure 4.16, définie à partir de la solution de 2-Partition, a un bien un débit ρ_{opt} de n/w. ■

Lors de la relecture de cette thèse, Claire Hanen a suggéré de réduire par rapport à 3-Partition, ce qui permet de montrer un résultat plus fort que le précédent :

Théorème 4.5. *Spanning-Tree-Dec est NP-complet au sens fort.*

Démonstration. Nous démontrons la NP-complétude de Spanning-Tree-Dec par réduction à 3-Partition dont la NP-complétude au sens fort est connue [58]. Considérons l'instance suivante de 3-Partition : soit $A = \{a_1, \ldots, a_{3n}\}$ et $B \in \mathbb{N}$. La question posée par 3-Partition est la suivante :

Existe-t-il des triplets partitionnant A et tels que la somme des éléments de chaque triplet fasse exactement B ?

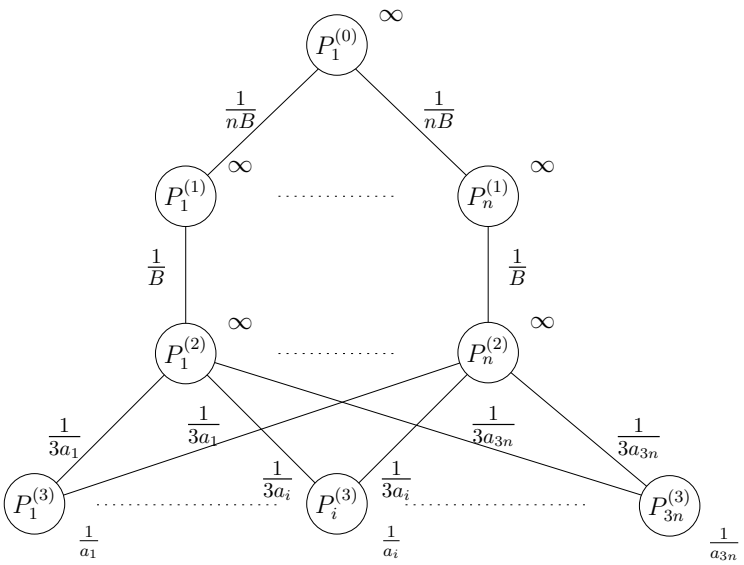


FIG. 4.17 – Instance de Spanning-Tree-Dec utilisée pour la réduction à 3-Partition

L'instance de Spanning-Tree-Dec que nous allons utiliser pour notre démonstration est constituée du graphe G représenté en figure 4.17.

L'objectif est de trouver une arborescence de débit nB. L'idée, c'est qu'un nœud de niveau 2 est attaché à exactement 3 nœud de niveau 3. En effet, ils ont tous un $\frac{c_i}{w_i} = \frac{1}{3}$. Donc si à un endroit on n'en prends que 2, il y a un endroit où il faut qu'il y en ait 4 et on ne pourra pas utiliser le quatrième nœud à pleine puissance, ce qui est indispensable pour atteindre le débit souhaité.

$\boxed{\Rightarrow}$ Supposons qu'il existe une 3-partition $((x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)))$ et montrons que quand $P_i^{(2)}$ est connecté à $P_{x_i}^{(3)}$, $P_{y_i}^{(3)}$ et $P_{z_i}^{(3)}$, on obtient une arborescence de débit nB.

Chaque nœud de niveau 2 peut exploiter ses trois fils à plein régime et a donc un débit débit égal à $\min(\frac{1}{c_{-1}}, a_{x_i} + a_{y_i} + a_{z_i}) = \min(B, B) = B$. Les nœuds de niveau 1 sont donc équivalent à un nœud de puissance de calcul $\frac{1}{B}$. La contrainte de bande-passante au niveau du processeur de niveau 0 s'écrit donc $\sum \frac{\frac{1}{nB}}{\frac{1}{B}} = 1$ le nœud de niveau 0 a donc un rendement égal à $\sum_{i=1}^{n} B = nB$.

$\boxed{\Leftarrow}$ Le débit de toute arborescence est d'au plus nB (la puissance de tous les processeurs en même temps quand on n'est pas limité par les communications). Dans une arborescence de débit nB tous les processeurs doivent donc travailler à temps plein. Comme chaque $P_i^{(3)}$ a un $\frac{c_i}{w_i}$ égal à $\frac{1}{3}$, chaque $P_i^{(2)}$ doit être connecté à exactement 3 nœuds de niveau 3. Le débit d'un $P_i^{(2)}$ est de rendement au plus B à cause de la liaison entre le niveau 1 et le niveau 2. Le rendement des nœuds de niveau 1 est donc d'au plus B également et il faut donc que les nœuds de niveau 2 soient de rendement exactement B si on veut avoir un rendement globale de nB. Chaque $P_i^{(2)}$ est donc relié à exactement trois processeurs de niveau 3 $P_{x_i}^{(3)}$, $P_{y_i}^{(3)}$ et $P_{z_i}^{(3)}$, qui

vérifient $a_{x_i} + a_{y_i} + a_{z_i} = B$. On a donc bien une trois-partition de A. ■

4.5.3.2 Résultat d'inapproximabilité

Une autre question naturelle que l'on peut se poser est la suivante : quelle est la perte engendrée par le choix d'une arborescence couvrante par rapport à l'utilisation de la plate-forme complète ? Le théorème que nous montrons dans cette section établit l'inapproximabilité d'un graphe par une arborescence, pour ce qui est du débit.

Théorème 4.6. *Pour tout entier K, il existe un graphe G tel que toute arborescence couvrante T de G a un débit tel que*

$$\frac{\rho_{opt}(G)}{\rho_{opt}(T)} \geqslant K.$$

Démonstration. Soit K un entier positif. Considérons le graphe représenté en figure 4.18. On vérifie aisément qu'en utilisant toutes le ressources de communications, il est possible

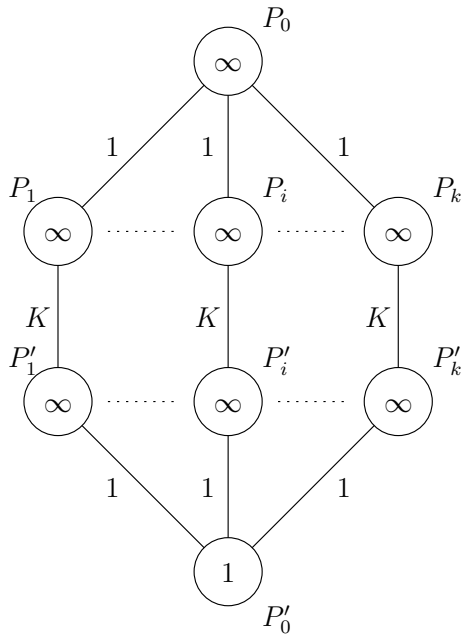


FIG. 4.18 – Graphe G utilisé pour démontrer l'inapproximabilité par une arborescence.

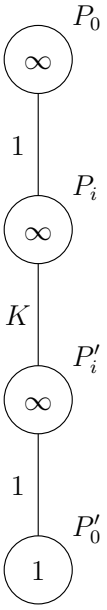


FIG. 4.19 – Description de l'arborescence T extraite de G.

de traiter une tâche par unité de temps, c'est-à-dire que $\rho_{\text{opt}}(G) = 1$. Cependant, toute arborescence couvrante G est de la forme de la chaîne représentée figure 4.19. En raison de la vitesse du lien entre P_i et P'_i, le nombre de tâches traitées par unité de temps est borné par $1/K$, d'où le résultat. ■

4.6 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la distribution de tâches indépendantes et de même taille sur des plates-formes hétérogènes. Nous avons développé une technique de résolution à base de programmation linéaire pour les plates-formes générales et donné des ordonnancements périodiques asymptotiquement optimaux. Dans le cas où la plate-forme est représentée par une arborescence enracinée au niveau du maître, nous avons donné des formes closes pour résoudre les équations linéaires précédentes. Ces formes closes conduisent à des ordonnancement orientés bande-passante dont on peut déduire des protocoles d'ordonnancement locaux et à la demande, robustes aux variations de charge.

Nous avons également montré que l'extraction de la «meilleure» arborescence couvrante d'une plate-forme générale, c'est-à-dire l'arborescence couvrante enraciné par le maître et permettant de traiter le plus de tâches possible en régime permanent, est un problème NP-complet et inapproximable.

Dans le chapitre suivant, nous nous intéressons au cas où les tâches élémentaires recèlent du parallélisme interne et sont décrites par un graphe de tâches.

Chapitre 5

Ordonnancement de graphes de tâches indépendants en régime permanent

5.1 Introduction

Dans ce chapitre, nous nous intéressons à l'exécution d'une application complexe sur une plate-forme hétérogène. Cette application consiste en une suite de problèmes indépendants à résoudre et chaque problème est composé d'un ensemble de tâches pouvant dépendre les unes des autres. Toute application où un algorithme (avec du parallélisme potentiel) doit être exécuté sur un ensemble de valeurs distinctes rentre dans ce type de modèle. Considérons par exemple le graphe de la figure 5.1(a) emprunté à Subhlok, Stichnoth, O'Hallaron et Gross [105]. Une boucle principale est exécutée un certain nombre de fois. Chaque instance du problème travaille sur des données distinctes mais chaque problème a le même graphe de tâches. Les modèles d'application et de plate-forme sont donc les mêmes que ceux présentés dans la section 4.2, si ce n'est, cette fois, que l'on considère des graphes quelconques et plus seulement constitués d'une seule tâche.

Ce chapitre est composé de trois parties. La première (section 5.2) expose les modifications à apporter aux équations du chapitre précédent pour prendre en compte les graphes de tâches généraux. La deuxième (section 5.3) explique comment construire un ordonnancement qui réalise le régime permanent calculé dans la section précédente. Enfin, la section 5.4 établit la complexité des algorithmes précédents et donne des pistes possibles en vue d'une mise en œuvre réelle.

5.2 Régime permanent optimal pour un DAG

Dans cette section, nous montrons que les équations établies au chapitre précédent sont insuffisantes pour calculer le régime permanent si le graphe est quelconque. Commençons par rappeler le programme linéaire qui nous avait permis de calculer le régime permanent

optimal pour des tâches indépendantes sur une plate-forme quelconque.

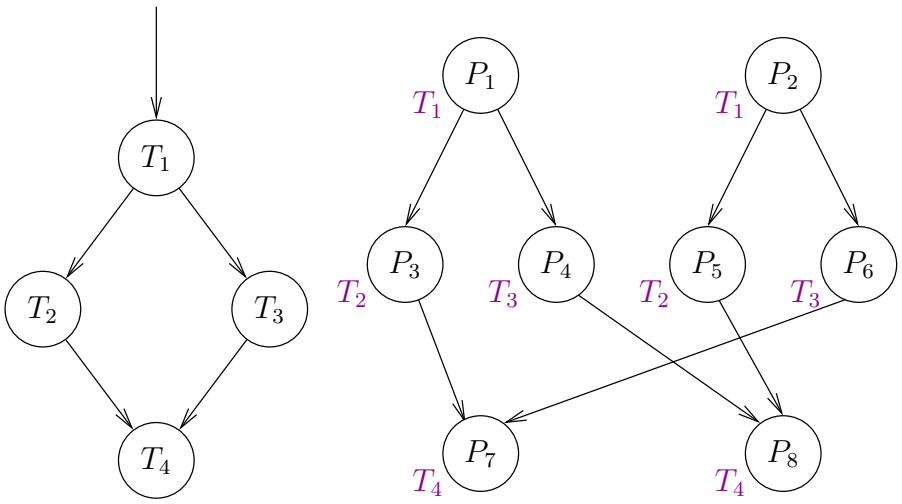
$$
\begin{array}{l}
\text{MAXIMISER } \rho = \sum_{i=1}^{p} cons(P_i, T_{end}), \\
\text{SOUS LES CONTRAINTES} \\
\left\{
\begin{array}{ll}
\text{(5.1a)} & \forall P_i, \forall T_k \in V_A, 0 \leqslant \alpha(P_i, T_k) \leqslant 1 \\
\text{(5.1b)} & \forall P_i \to P_j, \forall e_{k,l} \in E_A, 0 \leqslant s(P_i \to P_j, e_{k,l}) \leqslant 1 \\
\text{(5.1c)} & \forall P_i \to P_j, \forall e_{k,l} \in E_A, s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}) \times (data_{k,l} \times c_{i,j}) \\
\text{(5.1d)} & \forall P_i, \forall T_k, \alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k} \\
\text{(5.1e)} & \forall P_i, \sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1 \\
\text{(5.1f)} & \forall P_i, \sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) \leqslant 1 \\
\text{(5.1g)} & \forall P_i, \sum_{T_k \in V_A} \alpha(P_i, T_k) \leqslant 1 \\
\text{(5.1h)} & \forall P_i, \forall e_{k,l} \in E_A : T_k \to T_l, \\
& \qquad \sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k) = \\
& \qquad\qquad \sum_{P_j \to P_i} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l)
\end{array}
\right.
\end{array}
\tag{5.1}
$$

En section 5.2.1, nous expliquons pourquoi les équations précédentes fonctionnaient quand le graphe de l'application représentait des tâches indépendantes et pourquoi elles sont incomplètes dans le cas général. Nous montrons ensuite en section 5.2.2 qu'il est possible de les étendre en rajoutant d'autres variables exprimant des contraintes plus fortes. Les sections 5.2.3 et 5.2.4 établissent les relations entre ces variables et le programme linéaire final.

5.2.1 Pourquoi les graphes de tâches généraux sont-ils plus difficiles à gérer que les tâches simples ?

Le calcul du régime permanent pour un graphe général est bien plus difficile que dans le cas de tâches indépendantes. Considérons l'exemple représenté en figure 5.1.

Dans cet exemple, chaque processeur ne peut traiter qu'un seul type de tâche. En supposant que les temps de communication et de calcul sont tous égaux à 1, le programme linéaire (5.1) calculerait un débit de deux graphes de taches par unité de temps alors que cette plate-forme ne permet d'en traiter aucun. La difficulté provient de la partie *join* du graphe de tâches. En effet, il faut fusionner des données ($e_{2,4}$ et $e_{3,4}$) qui proviennent de la même instance du graphe de tâche. Or, les fichiers $e_{2,3}$ et $e_{3,4}$ correspondant à une



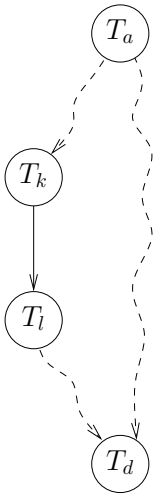
(a) Graphe de l'application

(b) Graphe de plate-forme : chaque processeur ne peut traiter qu'un seul type de tâche

FIG. 5.1 – Contre-exemple

même tâche T_1 sont systématiquement envoyés sur des processeurs différents. Il n'est donc pas possible de décomposer la solution du programme linéaire en une somme pondérée d'allocations comme cela était fait en section 4.3.5. Afin de pouvoir effectuer cette décomposition, il faut donc conserver une partie de l'ordonnancement de certains ancêtres afin d'être sûr que les fusions puissent s'effectuer correctement.

5.2.2 Ajout de contraintes



Afin de résoudre le problème exposé dans la section précédente, nous allons garder trace de l'allocation en annotant nos variables. Les variables $sent(P_i \to P_j, e_{k,l})$, $s(P_i \to P_j, e_{k,l})$, $\alpha(P_i, T_k)$ et $cons(P_i, T_k)$ vont donc être annotées par une liste de contraintes L et s'écriront donc $sent(P_i \to P_j, e^L_{k,l})$, $s(P_i \to P_j, e^L_{k,l})$, $\alpha(P_i, T^L_k)$ et $cons(P_i, T^L_k)$. Ces listes de contraintes représentent l'allocation de certains ancêtres (par exemple $\{T_{begin} \mapsto P_1, T_1 \mapsto P_2, T_3 \mapsto P_2\}$) et cette section explique comment nous construisons ces listes.

Intuitivement, lorsqu'il y a un *fork* dans le graphe de tâches, les fichiers qui en descendent doivent être annotés par l'allocation de cette tâche *fork*. En revanche, lorsqu'il y a un *join*, il est parfois possible d'oublier une partie de l'allocation.

Définition 5.1. *Pour toute dépendance $e_{k,l}$ on dit qu'une tâche T_a est contraignante pour $e_{k,l}$ si T_a est un ancêtre de T_l (ce peut donc être T_k) et s'il existe un T_d (qui peut être T_l) tel que :*

– *T_d est un descendant de T_l,*

– *il y a un chemin de T_a à T_d sommet-disjoint (excepté T_a et T_d) de celui allant de T_a à T_d en passant par T_l.*

Les *tâches contraignantes* d'une dépendance $e_{k,l}$ sont les tâches dont l'allocation doit être mémorisée si l'on veut pouvoir traiter correctement les parties *join* du graphe de tâches. Une telle tâche est une tâche *fork* dont tous les descendants n'ont pas encore été rassemblés. On peut les construire et annoter le graphe (voir figure 5.2) grâce à l'algorithme 5.1. L'idée consiste, non pas à chercher quelles sont les tâches contraignantes de chaque arête, mais à marquer pour chaque tâche, les arêtes pour lesquelles elle est contraignante. Ceci se fait en faisant un parcours en largeur et en comptant le nombre de parties *join* à partir de la tâche en question et non réunies par un *fork*. On peut remarquer que pour un même T_k, tous les $e_{k,l}$ ont le même ensemble de tâches contraignantes.

Lemme 5.1. *Pour un même T_k, tous les $e_{k,l}$ ont le même ensemble de tâches contraignantes.*

Démonstration. Dans cette démonstration, γ dénotera un chemin allant de T_{begin} à T_{end} et $\gamma[T_u, T_v]$, la suite ordonnée des sommets rencontrés en allant de T_u à T_v sur ce chemin.

Supposons qu'une tâche T_a soit contraignante pour $e_{k,l}$. Alors notons γ_1 et γ_2 deux chemins (voir figure 5.3(a)) tels que :

$$\begin{cases} \gamma_1[T_{begin}, T_a] = \gamma_2[T_{begin}, T_a] \\ (\gamma_1]T_a, T_d[) \cap (\gamma_2]T_a, T_d[) = \emptyset \\ \gamma_1[T_d, T_{end}] = \gamma_2[T_a, T_{end}] \\ e_{k,l} \in \gamma_2[T_a, T_d] \end{cases}$$

Soit $T_{l'}$ une tâche telle que $e_{k,l'}$ existe. Soit alors γ_3 un chemin tels que :

$$\begin{cases} \gamma_3[T_{begin}, T_k] = \gamma_2[T_{begin}, T_k] \\ e_{k,l'} \in \gamma_3[T_k, T_{end}] \end{cases}$$

Notons alors $T_{d'}$ le premier sommet de $(\gamma_2[T_l, T_{end}]) \cap (\gamma_3[T_{l'}, T_{end}])$.

Cas 1 : $T_{d'} \in \gamma_2[T_d, T_{end}]$. Deux possibilités se présentent :

– si $(\gamma_1[T_a, T_d]) \cap (\gamma_3[T_{l'}, T_{d'}]) = \emptyset$ (voir figure 5.3(b)) alors

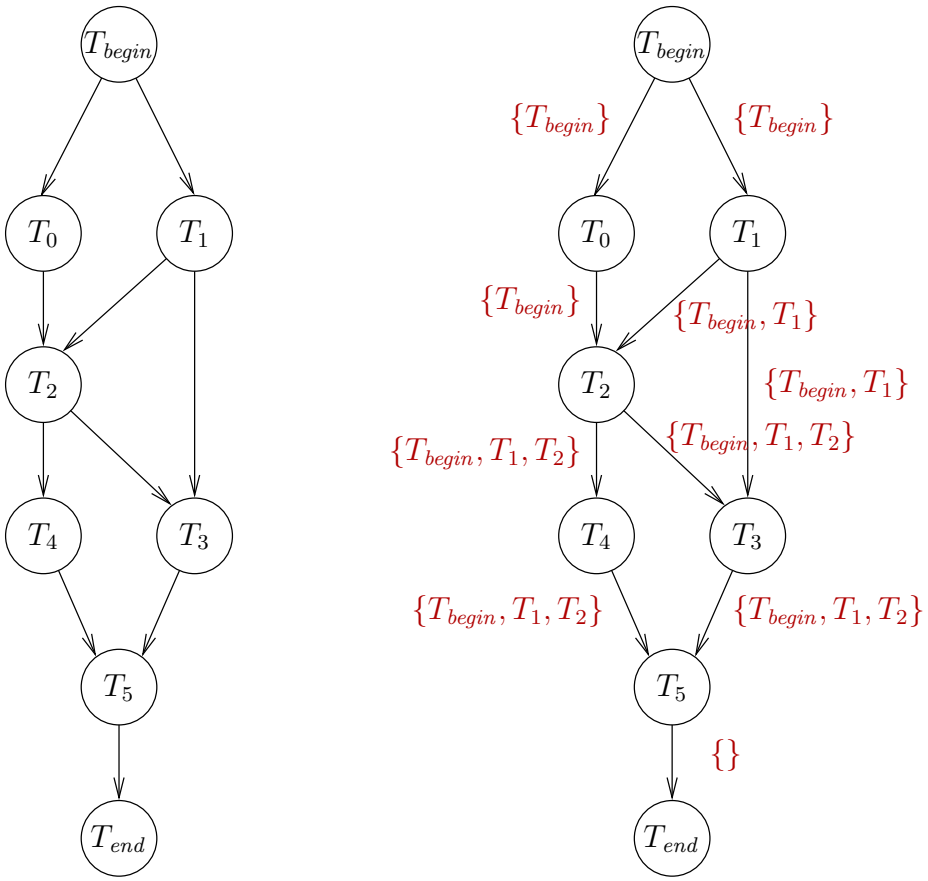
$$(\gamma_1]T_a, T_{d'}[) \cap (\gamma_3]T_a, T_{d'}[) = \emptyset \text{ avec } e_{k,l'} \in \gamma_3[T_a, T_{d'}].$$

et T_a est contraignante pour $e_{k,l'}$ à cause de $T_{d'}$.

– Sinon, notons $T_{d''}$ le premier sommet de $(\gamma_1[T_a, T_d]) \cap (\gamma_3[T_{l'}, T_{d'}])$ (voir figure 5.3(c)). On a alors

$$(\gamma_1]T_a, T_{d''}[) \cap (\gamma_3]T_a, T_{d''}[) = \emptyset \text{ avec } e_{k,l'} \in \gamma_3[T_a, T_{d''}].$$

T_a est donc bien contraignante pour $e_{k,l'}$ à cause de $T_{d''}$.



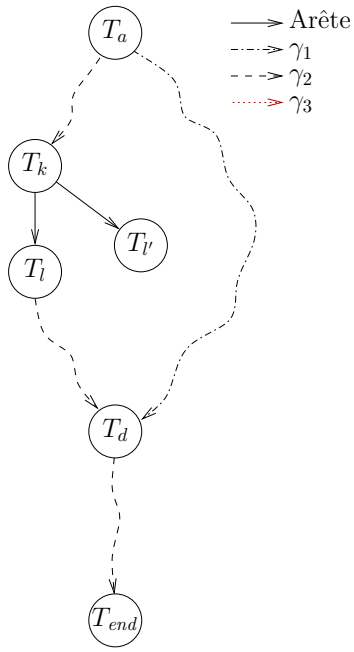
(a) Graphe de l'application

(b) Annotation du graphe. $e_{1,3}$ est annotée par T_{begin} et T_1 à cause de T_3 ; $e_{1,2}$ est annotée par T_{begin} et T_1 pour la même raison ;$e_{2,3}$ est annotée par T_{begin}, T_1 et T_2 à cause de T_3,T_3 et T_5 ; T_{end} n'est annotée par personne car le seul moyen d'y accéder est de passer par T_5 ;etc.

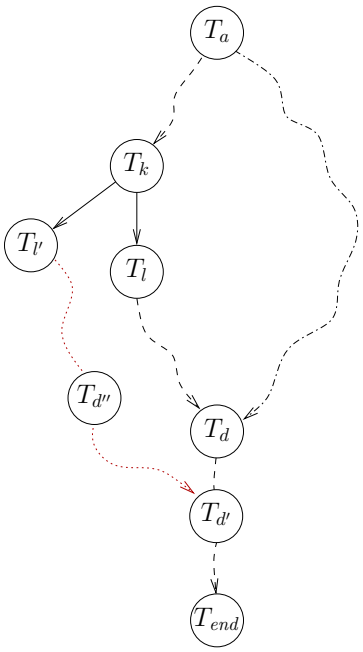
FIG. 5.2 – Annotation d'un graphe de tâche à l'aide de l'algorithme 5.1. Seuls les ancêtres nécessaires à la reconstruction sont mémorisés. Les annotations des arêtes entrantes nous indiquent les tâches qui doivent être mémorisées.

```
ACTIVATE(T_u)
 1: Active[T_u] ← 1
 2: Pour tout T_w t.q. T_u → T_w :
 3:    Si Active[T_w] ≠ 1 Alors
 4:       ToActivate[T_w]
 5:       inc(counter)
 6: dec(counter)
 7: ToActivate[T_u] ← 0

REMEMBER(T_u,T_v)
 8: Si counter>1 Alors
 9:    Pour tout T_w t.q. T_v → T_w :
10:       List[e_{v,w}] ← List[e_{v,w}] ∪ T_u

COMPUTELISTE()
11: Pour tout e_{k,l} :
12:    List[e_{k,l}] ← ∅
13: Pour tout T_u :
14:    Counter ← 1
15:    ACTIVATE(T_u)
16:    REMEMBER(T_u,T_u)
17:    Tant que |ToActivate| > 0 et counter > 1 :
         TO_ACTIVATE:
18:       Pour tout T_v t.q. ToActivate[T_v]=1 :
19:          nb ← 0
20:          Pour tout T_w t.q. T_w → T_v :
21:             Si (il existe un chemin de T_u à T_w) Alors
22:                Si Active[T_w] Alors
23:                   next TO_ACTIVATE
24:                inc nb
25:          ACTIVATE(T_v)
26:          counter ← counter - nb + 1
27:          REMEMBER(T_u,T_v)
```

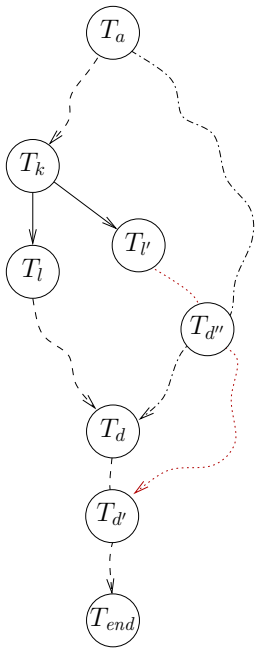
Algorithme 5.1: Calcul des tâches contraignantes.



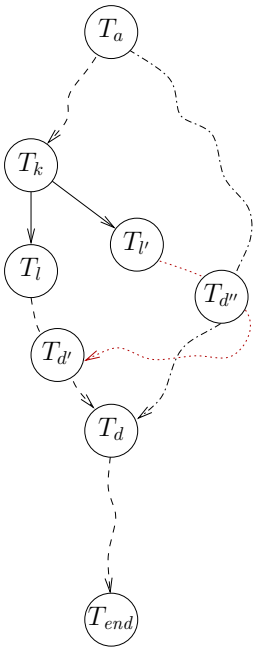
(a) Situation initiale : T_a est contraignante pour $e_{k,l}$ et on dispose donc de γ_1 et γ_2



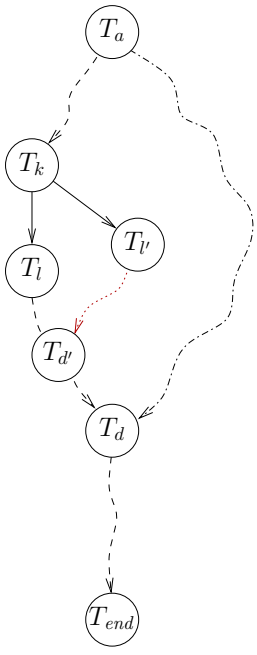
(b) Cas 1 : T_a est alors contraignante pour $e_{k,l'}$ à cause de $T_{d'}$



(c) Cas 1 : T_a est alors contraignante pour $e_{k,l'}$ à cause de $T_{d''}$



(d) Cas 2 : T_a est alors contraignante pour $e_{k,l'}$ à cause de $T_{d''}$



(e) Cas 2 : T_a est alors contraignante pour $e_{k,l'}$ à cause de T_d

FIG. 5.3 – Pour un même T_k, tous les $e_{k,l}$ ont le même ensemble de tâches contraignantes.

Cas 2 : $T_{d'} \in \gamma_2[T_l, T_d]$. Deux possibilités se présentent alors :
- Si $(\gamma_1[T_a, T_d]) \cap (\gamma_3[T_{l'}, T_{d'}])$ est non vide, notons $T_{d''}$ le premier sommet rencontré sur ces deux chemins (voir figure 5.3(d)). On a alors

$$(\gamma_1]T_a, T_{d''}[) \cap (\gamma_3]T_a, T_{d''}[) = \emptyset \text{ avec } e_{k,l'} \in \gamma_3[T_a, T_{d''}],$$

  et T_a est donc bien contraignante pour $e_{k,l'}$ à cause de $T_{d''}$.
- Si $(\gamma_1[T_a, T_d]) \cap (\gamma_3[T_{l'}, T_{d'}])$ est vide (voir figure 5.3(e)), on a alors

$$(\gamma_1]T_a, T_d[) \cap (\gamma_3]T_a, T_d[) = \emptyset \text{ avec } e_{k,l'} \in \gamma_3[T_a, T_d],$$

  et T_a est donc bien contraignante pour $e_{k,l'}$ à cause de T_d. ■

On peut maintenant définir les listes de contraintes pour les tâches T_k et pour les dépendances $e_{k,l}$.

Définition 5.2. *Une liste de contraintes pour une dépendance $e_{k,l}$ est une application de $\{T_{k_1}, \ldots, T_{k_q}\}$ sur $\{P_1, \ldots, P_p\}$, où $\{T_{k_1}, \ldots, T_{k_q}\}$ est l'ensemble de tâches contraignantes de $e_{k,l}$. On utilise une liste de la forme $\{T_{k_1} \to P_{i_1}, \ldots, T_{k_q} \to P_{i_q}\}$ pour les représenter. On note alors*

$$Cnsts(e_{k,l}) = \{ \textit{ listes de contraintes pour } e_{k,l} \}$$

Définition 5.3. *On dira que deux listes de contraintes L_1 et L_2 sont «*compatibles*» si et seulement si*

$$\forall (T_k \to P_i) \in L_1, \forall P_j \neq P_i, (T_k \to P_j) \notin L_2$$

On distingue deux types de contraintes pour une même tâche T_k, selon que ce sont des contraintes (dites d'entrée) qui doivent être vérifiées pour pouvoir traiter une tâche T_k, c'est-à-dire qui doivent être vérifiées pour tous les $e_{l,k}$, ou que ce sont des contraintes (dites de sortie) qui sont vérifiées par tous les $e_{k,l}$.

Définition 5.4. $CnstsIn(T_k) = \{$ *applications de* $\bigcup_{e_{l,k}}$ *(tâches contraignantes de* $e_{l,k}$*) vers* $\{P_1, \ldots, P_p\}$ $\}$

$$CnstsOut(T_k) = \begin{cases} Cnsts(e_{k,l}) & \textit{s'il existe un } T_l \textit{ tel que } T_k \to T_l \\ \emptyset & \textit{sinon} \end{cases}$$

(puisque grace au lemme 5.1, tous les $e_{k,l}$ ont les mêmes listes de contraintes).

Une tâche est le résultat de la fusion de fichiers dont les contraintes sont compatibles entre elles. $CnstsIn(T_k)$ est donc un sur-ensemble des contraintes possibles des fichiers d'entrées alors que $CnstsOut(T_k)$ est l'ensemble des contraintes que T_k vérifie une fois qu'elle a été calculée. Les équations que nous établissons en section 5.2.3 lient ces différentes contraintes ensembles.

Les deux définitions suivantes sont juste des notations permettant de simplifier l'énoncé des équations du programme linéaire.

Définition 5.5. *On dira qu'un fichier $e_{k,l}$ respectant les contraintes $L \in Cnsts(e_{k,l})$ «*peut être transféré de P_i à P_j*» si et seulement si $c_{i,j} \neq \infty$.*

Définition 5.6. *On dira qu'«*une tâche T_k peut être exécutée sur un processeur P_i en satisfaisant les contraintes $L \in CnstsOut(T_k)$*» si et seulement si $w_{i,k} \neq \infty$ et si le traitement de T_k sur P_i ne viole pas les contraintes L (c'est-à-dire s'il n'existe pas $P_j \neq P_i$ tel que $(T_k \rightarrow P_j) \in L$).*

5.2.3 Équations

Pour chaque arête $e_{k,l} : T_k \rightarrow T_l$ du graphe de tâches, pour chaque paire de processeur $P_i \rightarrow P_j$, et chaque liste de contrainte $L \in Cnsts(e_{k,l})$, on note $s(P_i \rightarrow P_j, e^L_{k,l})$ le temps moyen passé par P_i à envoyer à P_j des données de type $e_{k,l}$ soumises aux contraintes L. Comme d'habitude, $s(P_i \rightarrow P_j, e^L_{k,l})$ est un nombre rationnel positif. De même, on note $sent(P_i \rightarrow P_j, e^L_{k,l})$ le nombre de fichiers de ce type envoyé par unité de temps. On a le même type de relation que dans le chapitre précédent :

$$s(P_i \rightarrow P_j, e^L_{k,l}) = sent(P_i \rightarrow P_j, e^L_{k,l}) \times (data_{k,l} \times c_{i,j}), \tag{5.2}$$

ce qui signifie simplement que la fraction de temps passée à transférer de tels fichiers est égale au nombre de fichiers de ce type transférés par unité de temps multiplié par le temps nécessaire à en transférer un.

Pour chaque type de tâche $T_k \in V_A$, pour chaque processeur P_i et pour toute liste de contraintes valide $L \in CnstsOut(T_k)$, on note $\alpha(P_i, T^L_k)$ le temps moyen passé à traiter sur P_i des tâches de type T_k satisfaisant les contraintes L et $cons(P_i, T^L_k)$ nombre moyen de tâches de type T_k satisfaisant L traitées sur P_i par unité de temps. On a la relation

$$\alpha(P_i, T^L_k) = cons(P_i, T^L_k) \times w_{i,k} \tag{5.3}$$

Avant de pouvoir commencer, à traiter une tâche T_k sur un processeur P_i, cette dernière doit être prête, c'est-à-dire que tous les fichiers nécessaires à son traitement doivent être rassemblés sur P_i. Or, la liste de contraintes des fichiers d'entrées (et donc appartenant à $CnstsIn(T_k)$) d'une tâche est souvent différente de la liste de contraintes des fichiers de sortie (appartenant à $CnstsOut(T_k)$). Elle peut diminuer, comme pour T_5 sur la figure 5.2(b), ou augmenter comme T_1 sur la figure 5.2(b). C'est la raison pour laquelle, nous distinguons les tâches prêtes à être traitées sous certaines contraintes ($prod(P_i, T_k)$) de celles qui viennent d'être traitées et viennent de produire de fichiers de sortie ($cons(P_i, T_k)$). On a donc le lien suivant entre $prod(P_i, T_k)$ et $cons(P_i, T_k)$:

$$cons(P_i, T^L_k) = \sum_{\substack{L_2 \in CnstsIn(T_k) \\ T_k \text{ peut être traitée sur } P_i \text{ en satisfaisant } L_2 \\ L \text{ et } L_2 \text{ sont compatibles}}} prod(P_i, T^{L_2}_k) \tag{5.4}$$

Nous cherchons donc des valeurs rationnelles des variables $s(P_i \rightarrow P_j, e^L_{k,l})$, $sent(P_i \rightarrow P_j, e^L_{k,l})$, $\alpha(P_i, T^L_k)$, $cons(P_i, T^L_k)$ et $prod(P_i, T^L_k)$. En régime permanent, ces quantités sont soumises aux contraintes suivantes :

Activité sur une période unitaire Les fractions de temps passées par les différents processeurs à faire quelque chose, que ce soit calculer ou communiquer, doivent appartenir à l'intervalle $[0, 1]$ puisque ces quantités correspondent à une activité moyenne pendant une unité de temps :

$$\forall P_i, \forall T_k \in V_A, \forall L \in CnstsOut(T_k), 0 \leqslant \alpha(P_i, T_k^L) \leqslant 1 \tag{5.5}$$

$$\forall P_i \to P_j, \forall e_{k,l} \in E_A, \forall L \in Cnsts(e_{k,l}), 0 \leqslant s(P_i \to P_j, e_{k,l}^L) \leqslant 1 \tag{5.6}$$

Modèle 1-port en sortie Les émissions du processeur P_i devant être effectués séquentiellement vers ses voisins, on a l'équation suivante :

$$\forall P_i, \quad \sum_{\substack{P_i \to P_j \\ e_{k,l} \in E_A \\ L \in Cnsts(e_{k,l})}} s(P_i \to P_j, e_{k,l}^L) \leqslant 1 \tag{5.7}$$

Modèle 1-port en entrée Les réceptions du processeur P_i devant être effectués séquentiellement, on a l'équation suivante :

$$\forall P_i, \quad \sum_{\substack{P_j \to P_i \\ e_{k,l} \in E_A \\ L \in Cnsts(e_{k,l})}} s(P_j \to P_i, e_{k,l}^L) \leqslant 1 \tag{5.8}$$

Recouvrement des calculs et des communications En raison des hypothèses faites sur le recouvrement, il n'y a pas d'autres contraintes sur $\alpha(P_i, T_k)$ que

$$\forall P_i, \quad \sum_{\substack{T_k \in V_A \\ L \in CnstsOut(T_k)}} \alpha(P_i, T_k^L) \leqslant 1 \tag{5.9}$$

5.2.4 Loi de conservation

Les dernières contraintes sont les *lois de conservation*. Considérons un processeur P_i et une dépendance $e_{k,l}^L$ du graphe de tâches annotée par les contraintes L. Pendant chaque unité de temps, P_i reçoit de ses voisins un certain nombre de fichiers de type $e_{k,l}^L$. Il en reçoit $\sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l}^L)$ exactement. Le processeur P_i lui-même exécute un certain nombre de tâches T_k^L, $cons(P_i, T_k^L)$ pour être exact, qui vont générer autant de fichiers de type $e_{k,l}^L$.

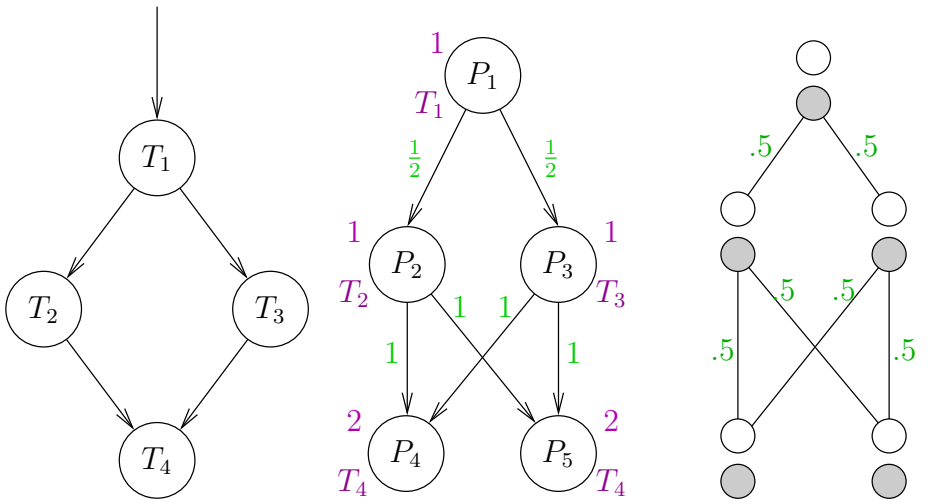
Qu'arrive-t-il à ces fichiers ? Certains sont envoyés à des voisins de P_i et d'autres sont utilisés pour produire des tâches $T_l^{L_2}$ (avec L_2 compatible avec L) prêtes à être consommées par P_i. On en déduit la loi de conservation suivante :

$$\forall P_i, \forall e_{k,l} \in E_A, \forall L \in Cnsts(e_{k,l})$$
$$\sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l}^L) + cons(P_i, T_k^L) =$$
$$\sum_{P_i \to P_j} sent(P_i \to P_j, e_{k,l}^L) + \sum_{\substack{L_2 \in CnstsIn(T_l) \\ L \text{ et } L_2 \text{ compatibles}}} prod(P_i, T_l^{L_2}) \tag{5.10}$$

Le régime permanent optimal peut donc être obtenu avec le programme linéaire suivant :

MAXIMISER $\rho = \sum_{i=1}^{p} cons(P_i, T_{end}^{\{\}})$,
SOUS LES CONTRAINTES

$$\begin{cases}
(5.11a) \quad s(P_i \to P_j, e_{k,l}^L) = sent(P_i \to P_j, e_{k,l}^L) \times (data_{k,l} \times c_{i,j}) \\
(5.11b) \quad \alpha(P_i, T_k^L) = cons(P_i, T_k^L) \times w_{i,k} \\
(5.11c) \quad cons(P_i, T_k^L) = \sum_{\substack{L_2 \in CnstsIn(T_k) \\ T_k \text{ peut être traitée sur } P_i \text{ en satisfaisant } L_2 \\ L \text{ et } L_2 \text{ sont compatibles}}} prod(P_i, T_k^{L_2}) \\
(5.11d) \quad \forall P_i, \sum_{\substack{P_i \to P_j \\ e_{k,l} \in E_A \\ L \in Cnsts(e_{k,l})}} s(P_i \to P_j, e_{k,l}^L) \leqslant 1 \\
(5.11e) \quad \forall P_i, \sum_{\substack{P_j \to P_i \\ e_{k,l} \in E_A \\ L \in Cnsts(e_{k,l})}} s(P_j \to P_i, e_{k,l}^L) \leqslant 1 \\
(5.11f) \quad \forall P_i, \sum_{\substack{T_k \in V_A \\ L \in CnstsOut(T_k)}} \alpha(P_i, T_k^L) \leqslant 1 \\
(5.11g) \quad \forall P_i, \forall e_{k,l} \in E_A : T_k \to T_l, \forall L \in Cnsts(e_{k,l}) \\
\qquad \sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l}^L) + cons(P_i, T_k^L) = \\
\qquad \sum_{P_i \to P_j} sent(P_i \to P_j, e_{k,l}^L) + \sum_{\substack{L_2 \in CnstsIn(T_l) \\ L \text{ et } L_2 \text{ compatibles}}} prod(P_i, T_l^{L_2})
\end{cases} \tag{5.11}$$



(a) Graphe de tâches. Tous les poids sont identiques

(b) Graphe de plate-forme

(c) Transformation en un graphe biparti

FIG. 5.4 – Illustration de la difficulté de la reconstruction d'un ordonnancement valide.

5.3 Ordonnancement

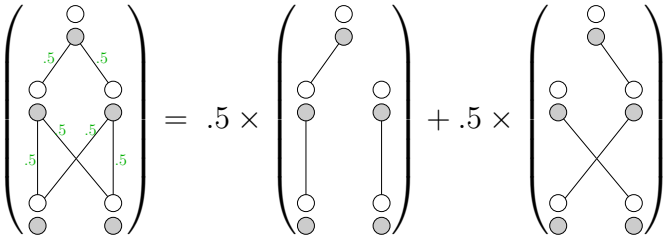
La section 5.3.1 présente un exemple simple illustrant la difficulté de la reconstruction d'un ordonnancement pour des graphes généraux par rapport aux cas des tâches indépendantes. La section 5.3.2 présente une façon de décomposer la solution du programme linéaire en une somme pondérée d'allocations et la section 5.3.3 explique comment utiliser ces allocations pour construire un ordonnancement réalisant le régime permanent optimal.

5.3.1 Pourquoi les graphes de tâches généraux sont-ils plus difficiles à gérer que les tâches simples ?

Cette section illustre la difficulté qu'il peut y avoir à reconstruire un ordonnancement valide si on ne passe pas par la décomposition en allocations.

Intéressons-nous à l'exemple de la figure 5.4. Chaque processeur de la plate-forme (figure 5.4(b)) ne peut traiter qu'un seul type de tâche. Le type de tâche et son temps de traitement sont notés à coté de chacun des processeurs. Chaque dépendance a une taille unitaire et le poids des arêtes du graphe de plate-forme représente donc le temps nécessaire au transfert d'un fichier. En utilisant le programme linéaire (5.11), on obtient un débit maximal d'un graphe de tâche par unité de temps. En utilisant les mêmes techniques qu'en section 4.3.5, on peut transformer le graphes des communications en un graphe

biparti (figure 5.4(c)) et le décomposer de la façon suivante :



(5.12)

Néanmoins, il faut encore faire attention aux parties *join* et si la décomposition en couplage permet d'obtenir un ordonnancement où les contraintes 1-ports sont respectées, elle ne peut rien pour assurer que les dépendances le sont. La figure 5.5 représente deux ordonnancements. Le premier est construit en utilisant un parcours en largeur du graphe de tâches et n'est pas valide car les fichiers correspondants à $e_{2,4}$ et à $e_{3,4}$ pour la première instance du graphe de tâches sont envoyés sur des processeurs différents (respectivement P_4 et P_5).

Ce problème vient évidemment du fait que l'on essaye de reconstruire le motif sans avoir les allocations. Mais si avant, les allocations se construisaient simplement en parcourant en profondeur le graphe de tâches, ce n'est plus possible maintenant à cause des *join*. Une allocation correcte peut néanmoins être construite en parcourant le graphe de tâches à l'envers (figure 5.5), ce qui permet de s'assurer que les parties *join* sont bien effectuées (les contraintes du programme linéaire nous assurent que cela est toujours possible). C'est la technique que nous allons utilisons dans la section suivante pour décomposer la solution du programme linéaire en une somme d'allocations.

5.3.2 Décomposition en allocations

La plate-forme que nous utilisons pour illustrer notre propos est représentée en figure 5.6(a) et le graphe d'application en figure 5.6(b).

L'idée sous-jacente à l'intérêt que nous portons au régime permanent est la possibilité offerte de pouvoir mélanger plusieurs ordonnancements pour exploiter au mieux les capacités de la plate-forme. Dans cette section, nous expliquons comment décomposer la solution du programme linéaire (5.11) en une somme pondérée d'allocations.

On obtient cette décomposition en annotant le graphe de tâches avec les valeurs non nulles de $cons(P_i, T_k^L)$, $prod(P_i, T_k^L)$ et de $sent(P_i \to P_j, e_{k,l}^L)$ (voir figure 5.7). La décomposition est également plus aisée en introduisant $sent(P_i \to P_i, e_{k,l}^L)$, le nombre de fichiers de type $e_{k,l}^L$ produit sur place et qui ne sont pas transférés vers un autre processeur mais sont utilisés directement pour un autre calcul. Cette quantité se définit donc de la façon

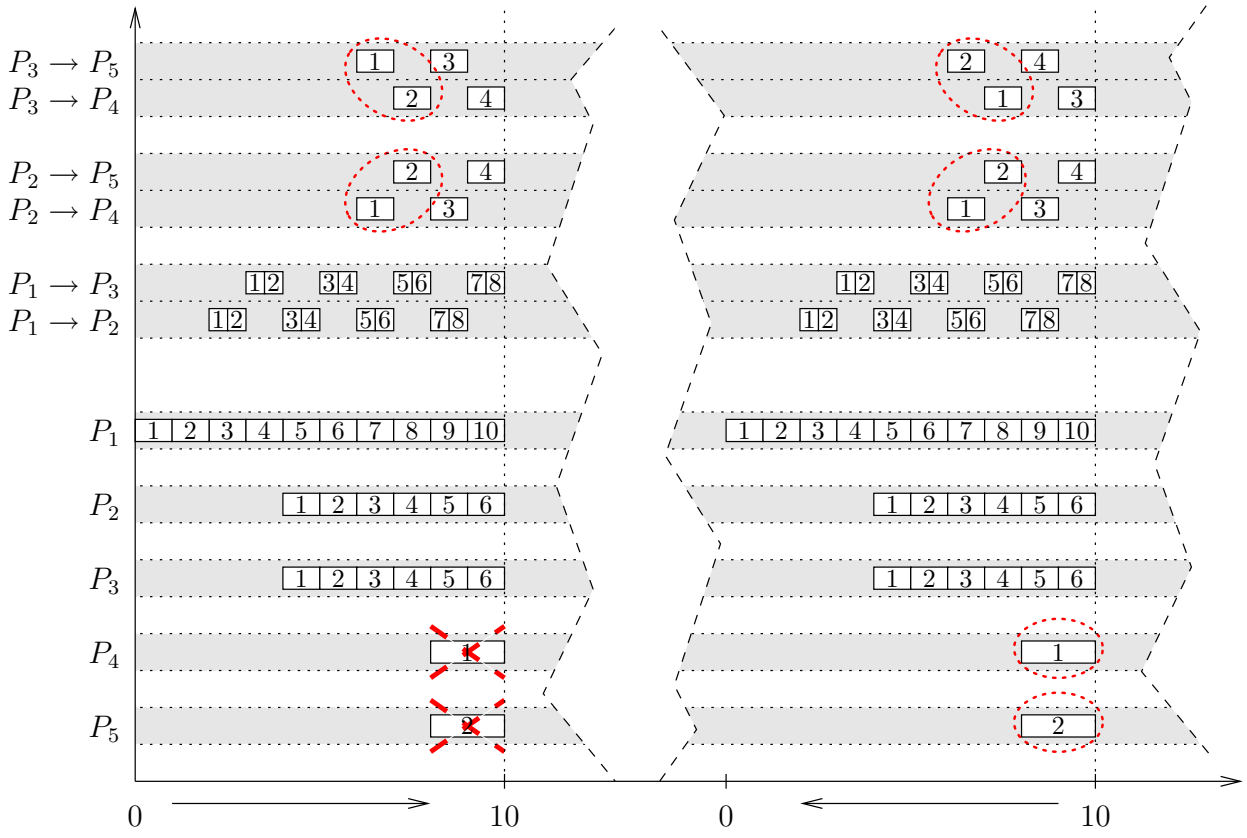
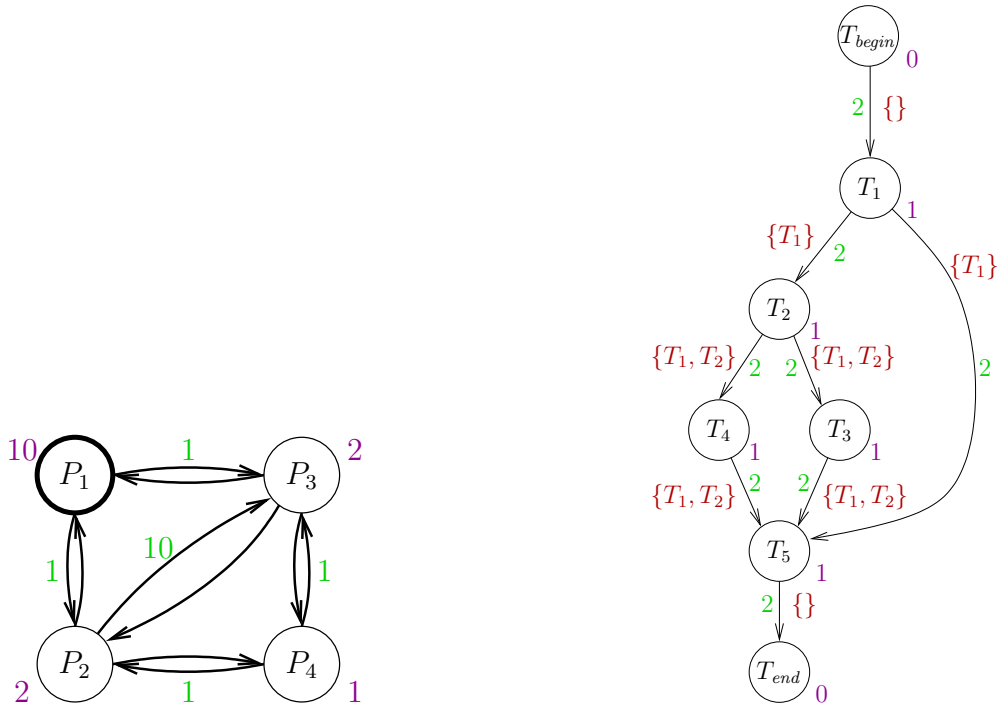


FIG. 5.5 – Ordonnancements construits à partir de la décomposition en graphes bipartis



(a) Graphe de plate-forme : Les fichiers d'entrée $e_{begin,1}$ sont placé sur P_1 à l'origine et les fichiers de sortie $e_{1,end}$ doivent être rassemblés sur P_1

(b) Graphe de tâches annoté par les listes de contraintes

FIG. 5.6 – Exemples pour illustrer la décomposition en allocations.

suivante :

$$\forall P_i, \forall e_{k,l} \in E_A : T_k \rightarrow T_l, \forall L \in Cnsts(e_{k,l}) :$$
$$sent(P_i \rightarrow P_i, e^L_{k,l}) = \sum_{\substack{L_2 \in CnstsIn(T_l) \\ T_l \text{ peut être traitée sur } P_i \text{ en satisfaisant } L_2 \\ L \text{ et } L_2 \text{ sont compatibles}}} prod(P_i, T^{L_2}_l) - \sum_{\substack{L_1 \in CnstsOut(T_k) \\ L \text{ et } L_1 \text{ sont compatibles}}} sent(P_j \rightarrow P_i, e^{L_1}_{k,l}) \quad (5.13)$$

Comme nous l'avons fait remarquer en section 5.3.1, il faut effectuer une remontée du graphe de tâches pour obtenir un ordonnancement valide. C'est ce que fait l'algorithme 5.2.

L'allocation est construite à partir des valeurs positives de $cons(P_i, T^L_k)$, $prod(P_i, T^L_k)$ et $sent(P_i \rightarrow P_j, e^L_{k,l})$. Les équations de conservation nous garantissent donc qu'une telle allocation peut toujours être extraite. Une fois une allocation extraite, on détermine son poids en prenant le minimum des quantités $cons(P_i, T^L_k)$, $prod(P_i, T^L_k)$ et $sent(P_i \rightarrow P_j, e^L_{k,l})$ impliquées dans l'allocation.

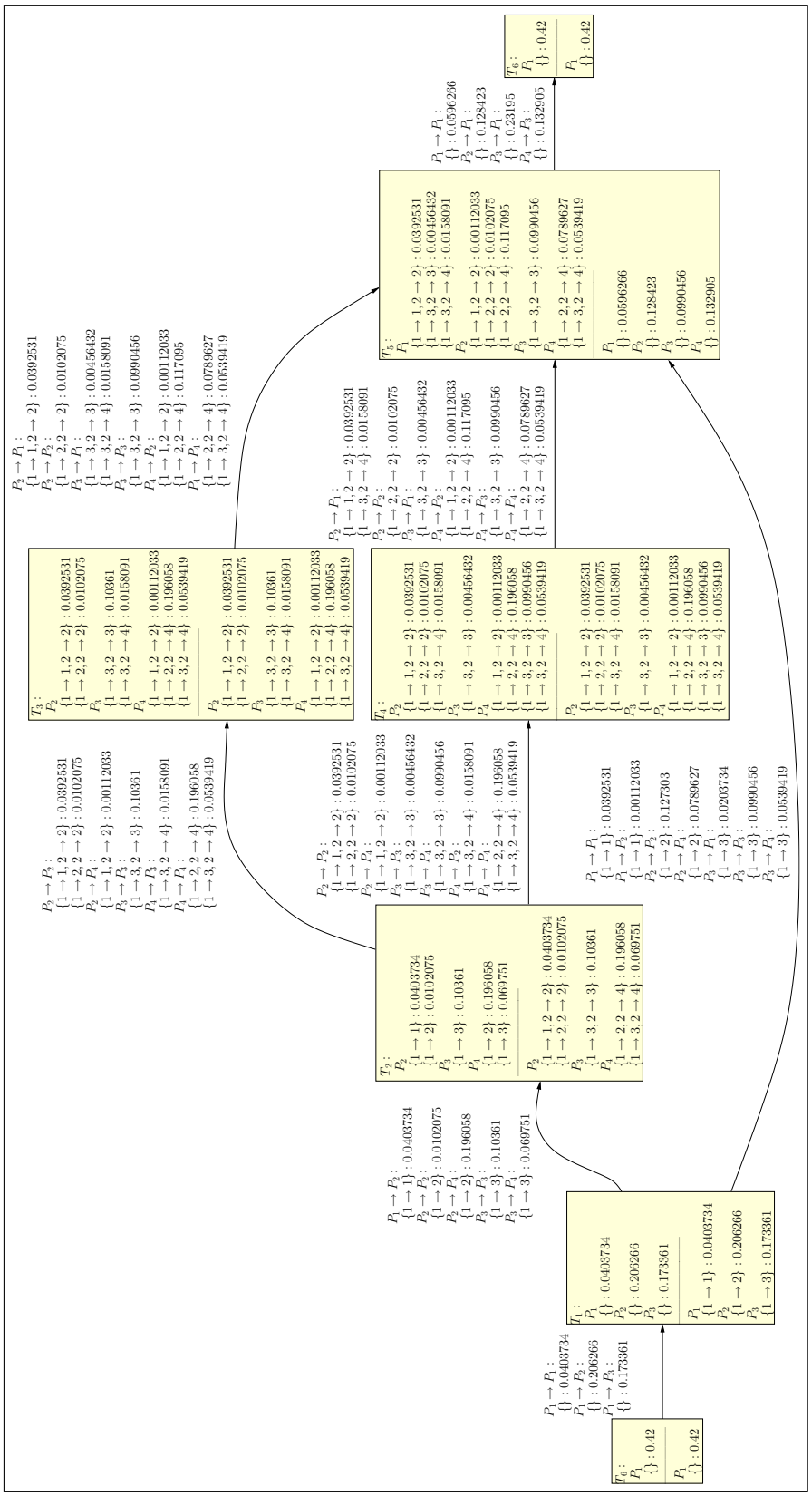


FIG. 5.7 – Solution du programme linéaire : le graphe de tâches est annoté avec les valeurs non-nulles de $cons(P_i, T_k^L)$, $prod(P_i, T_k^L)$ et $sent(P_i \rightarrow P_j, e_{k,l}^L)$

```
FIND_A_SCHEDULE()
 1: to_activate← {T_end}
 2: CnstsCons(T_end) ← ∅
 3: P(T_end) ← un P_i t.q. prod(P_i, T_end^∅) > 0
 4: Tant que to_activate ≠ ∅ :
 5:    l ← POP(to_activate)
 6:    i ← P(T_l)
 7:    L ← CnstsCons(T_l)
 8:    Soit L_1 t.q. prod(P_i, T_l^{L_1}) > 0 et L_1 compatible avec L
 9:    CnstsProd(T_l) ← L_1
10:    Si T_l ≠ T_begin Alors
11:       Pour tout T_k t.q. T_k → T_l :
12:          Soit L_2 et j t.q. sent(P_j → P_i, e_{k,l}^{L_2}) >0 et L_2 compatible avec L_1
13:          Cnsts(e_{k,l}) ← L_2
14:          CnstsCons(T_k)← L_2
15:          transfer(e_{k,l}) ← {P_j → P_i}
16:          src← j
17:          Si P_i ≠ P_j et prod(P_j, T_k^{L_2}) = 0 Alors
18:             dst← j
19:             Répéter
20:                Soit P_src ≠ P_j t.q. sent(P_src → P_dst, e_{k,l}^{L_2}) >0
21:                to_activate← to_activate ∪ {P_src → P_dst}
22:             jusqu'à ce que prod(P_src, T_k^{L_2}) >0
23:          P(T_k)← P_src
```

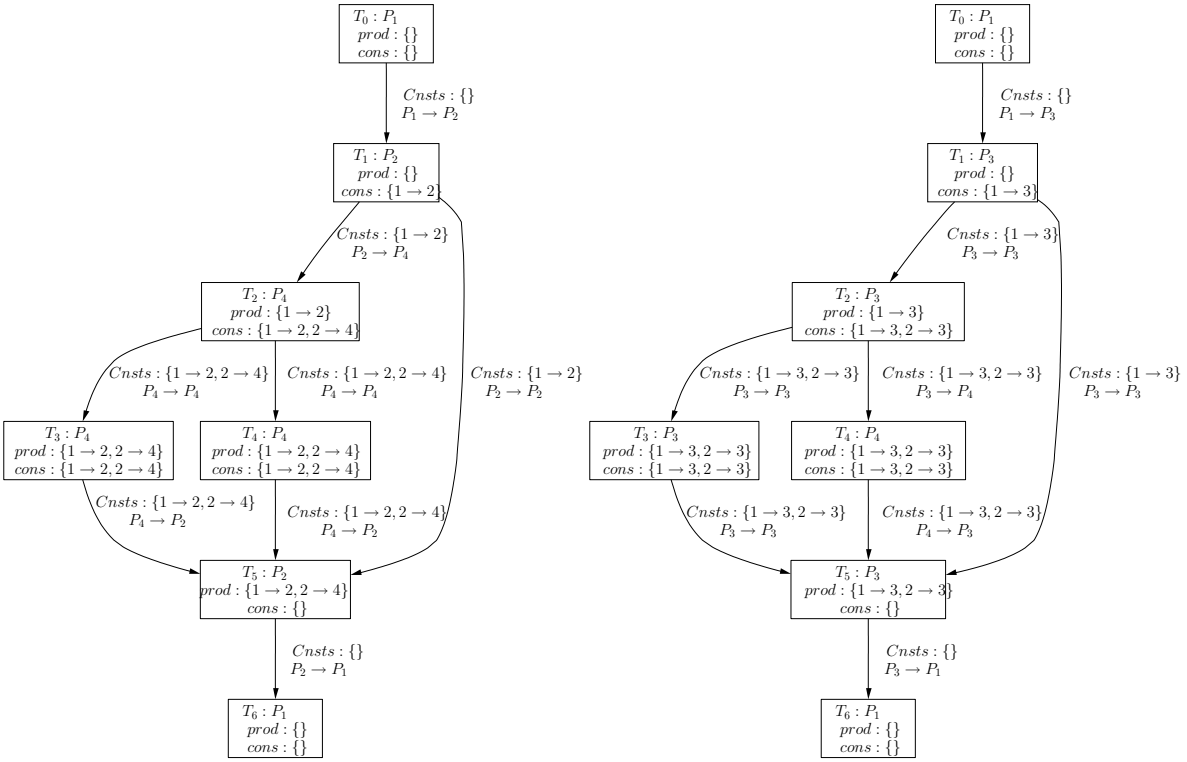
Algorithme 5.2: Algorithme d'extraction d'une allocation

Pour décomposer une solution en une somme pondérée d'allocations, il suffit donc d'extraire une allocation, d'évaluer son poids et de le soustraire à la solution du programme linéaire jusqu'à ce que $cons(P_i, T_{end}) = 0$ pour tout P_i. En effet, en soustrayant uniformément ces valeurs, les équations du programme linéaire sont toujours satisfaites et on peut donc extraire une nouvelle allocation d'ordonnancement tant qu'il reste un P_i tel que $cons(P_i, T_{end}) > 0$. Enfin, comme à chaque extraction, on supprime au moins une arête (celle qui réalisait le minimum des quantités $cons(P_i, T_k^L)$, $prod(P_i, T_k^L)$ et $sent(P_i \rightarrow P_j, e_{k,l}^L)$ impliquées dans l'allocation), la décomposition est polynomiale et se fait en temps polynomiale en ses entrées.

La solution de la figure 5.7 se décompose en une dizaine d'allocations différentes. La figure 5.8 donne les deux principaux, c'est-à-dire ceux dont le poids est le plus important.

5.3.3 Compatibilité des allocations

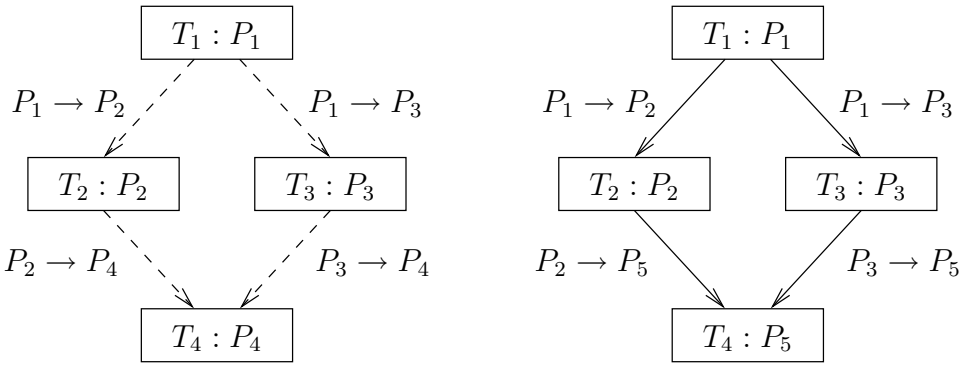
Dans cette section, nous expliquons comme utiliser les allocations obtenues dans la section précédente pour construire un ordonnancement valide et réalisant le régime per-



(a) Poids : 0.117095 (27.88%)

(b) Poids : 0.0990454 (23.58%)

FIG. 5.8 – Allocations principales de la décomposition de la solution de la figure 5.7



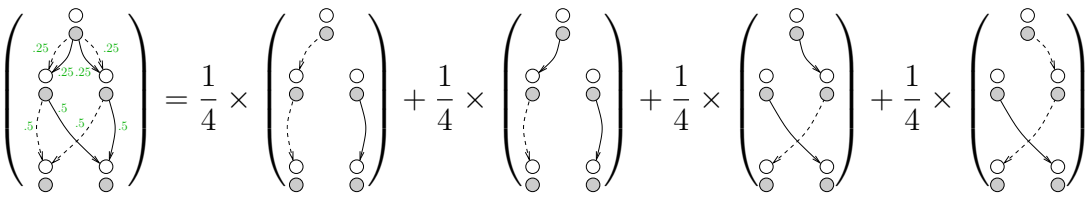
(a) Poids : 0.5 (50.0%) (b) Poids : 0.5 (50.0%)

FIG. 5.9 – Allocations de la décomposition du régime permanent optimal pour le graphe de tâches de la figure 5.4(a) sur la plate-forme de la figure 5.4(b). Les listes de contraintes sont sans importances sur cet exemple particulier et ne sont donc pas représentées pour alléger les notations.

manent optimal. La technique est rigoureusement la même que dans la section 4.3.5 si ce n'est que les allocations sont plus complexes que dans le cas de tâches indépendantes.

Nous revenons sur l'exemple de la section 5.3.1 pour illustrer cette méthode. Le régime permanent optimal est obtenu en utilisant les deux allocations représentées figure 5.9.

Un multi-graphe biparti (voir figure 5.10) est associé au graphe de plate-forme de la figure 5.4(b) et aux deux allocations de la figure 5.9. Chaque processeur est dédoublé en un nœud de réception (en blanc) et un nœud de réception (en gris). Pour chaque allocation S et chaque $sent(P_i \to P_j, e^L_{k,l})$ apparaissant dans S, on crée une arête de P_i à P_j indicée par $e^L_{k,l}$ et pondérée par le produit du poids de S avec $data_{k,l} \times c_{i,j}$. Par construction, pour tout nœud de ce graphe biparti, la somme des poids des arêtes adjacentes est inférieur à 1. En utilisant l'algorithme de coloriage donné dans [96, vol.A chapitre 20], on peut donc décomposer notre multigraphe en une somme pondérée de couplages telle que la somme des coefficients est inférieure à 1.



(5.14)

On peut alors définir un motif valide et permettant de réaliser le régime permanent optimal comme dans la section 4.3.5. L'ordonnancement ainsi obtenu est valide car chaque

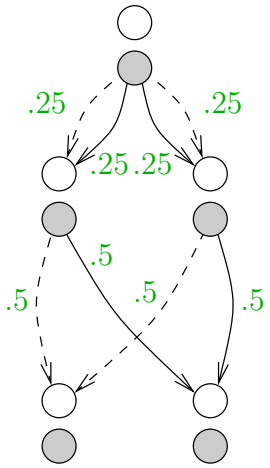


FIG. 5.10 – Graphe biparti associé au graphe de plate-forme de la figure 5.4(b) et aux allocations de la figure 5.9. Chaque arêtes va d'un processeur P_i à un processeur P_j et est associée à un $e_{k,l}$ d'une allocations donnée (les arêtes pointillées représentent l'allocation de gauche et les pleines celle de droite) . Le poids de l'arête est la fraction de temps passée à transférer ce fichier de P_i à P_j dans cette allocation.

instance du graphe de tâches est exécutée selon une allocation et car les contraintes 1-port sont bien respectées. On peut alors démontrer, comme dans le chapitre précédent, qu'un tel ordonnancement est asymptotiquement optimal.

5.4 Quelques mots sur la complexité

5.4.1 Complexité du calcul du régime permanant

Notons n le nombre de tâches du graphe de tâches et p le nombre de processeurs dans le graphe de plate-forme. Alors, le nombre de variables du programme linéaire 5.11 est potentiellement proportionnel à $p^2n^2p^n$. En effet, le nombre de listes de contraintes L, c'est-à-dire d'applications de $\{T_{k_1}, \ldots, T_{k_q}\}$ sur $\{P_1, \ldots, P_p\}$, peut être égal à p^n et on manipule des variables telles que $sent(P_i \rightarrow P_j, e^L_{k,l})$. Cette situation se produit sur les graphes composés d'un grand nombre de parties *fork* et qui ne sont réunies par un *join* que tardivement, comme c'est la cas du graphe de la figure 5.11. Sur cette figure, toutes les tâches T_1,...,T_n doivent être mémorisées pour être assuré d'une reconstruction correcte de la tâche T_f. Cependant, sur le graphe de la figure 5.12, au plus une tâche doit être mémorisée dans chaque liste de contraintes.

Définition 5.7. *La profondeur de dépendance est le nombre maximal de tâches contraignantes d'un $e_{k,l}$, sur l'ensemble des $e_{k,l}$.*

À l'aide de cette notion, on peut alors définir le problème d'optimisation suivant.

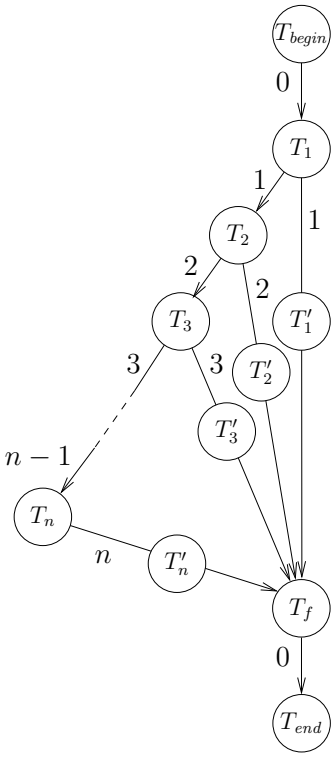


FIG. 5.11 – Toutes les tâches $T_1,\ldots,T_n$ doivent être mémorisées pour être assuré d'une reconstruction correcte de la tâche T_f. Chaque arête est annotée par son nombre de tâches contraignantes.

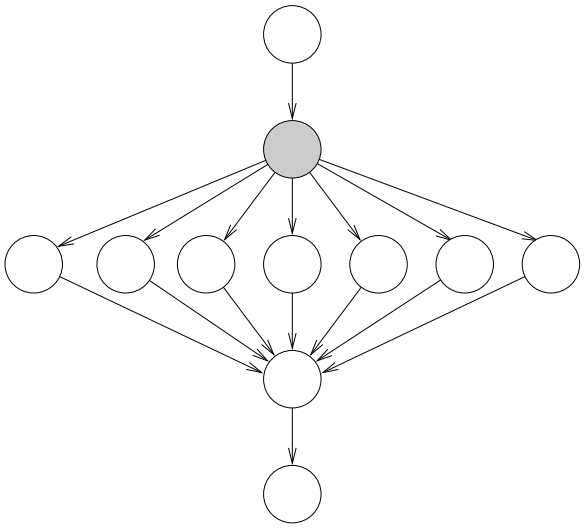


FIG. 5.12 – Graphe *fork*. Seule l'allocation de la tâche grise doit être mémorisée pour pouvoir reconstruire les autres tâches. La profondeur de dépendance est donc égale à 1.

Définition 5.8 (Reg-Perm$_d(G_a, G_p)$). *Étant donné un graphe de tâches G_a dont la profondeur de dépendance est bornée par d et un graphe de plate-forme G_p, trouver un ordonnancement périodique réalisant le régime permanent optimal.*

En se restreignant aux graphes de profondeur maximale de dépendance fixée, on peut alors montrer le résultat suivant.

Théorème 5.1. *Pour tout $d \in \mathbb{N}$, Reg-Perm$_d$ est polynomial.*

Démonstration. Le programme linéaire (5.11) peut être résolu en temps polynomial car le nombre de variables et de contraintes (avec d fixé) est un $O(p^2n^2p^d)$. Les algorithmes de la section 5.3.2 (décomposition en allocations) et 5.3.3 (décomposition en graphes bipartis) sont polynomiaux en leurs entrées et donc polynomiaux en n et p. ■

Si on ne se restreint pas aux graphes de profondeur maximale de dépendance fixée, on ne peut plus obtenir avec notre algorithme en temps polynomial le régime permanent

optimal ni un ordonnancement atteignant ce débit. Il n'y a d'ailleurs aucune raison que ce problème soit dans NP. En effet, il faut alors vérifier que la solution est effectivement valide (au niveau de la reconstruction), ce qui est, comme on l'a vu dans ce chapitre, loin d'être trivial. La NP-complétude de ce problème reste donc une question ouverte.

5.4.2 Approximation des valeurs

Notons $S_1, \ldots, S_q$ les allocations obtenues dans la section 5.3.2 et $\alpha_1 = \frac{p_1}{T_p}, \ldots, \alpha_q = \frac{p_q}{T_p}$ leur débit. On a donc la relation suivante entre le débit optimal ρ_{opt} et les α_i :

$$\rho_{\text{opt}} = \sum_{i=1}^{q} \alpha_i$$

Il se peut cependant que T_p soit très grand et non utilisable en pratique. Il peut donc être nécessaire d'arrondir les poids des allocations. Calculons le débit $\rho(T)$ qui peut être simplement obtenu en arrondissant les α_i pour une période T. Notons $r_i(T)$ le nombre de graphes de tâches pouvant être exécuté par l'allocation S_i en respectant les proportions. Posons

$$r_i(T) = \lfloor \alpha_i T \rfloor . \tag{5.15}$$

En prenant une partie entière inférieure, les inégalités du programme linéaire (5.11) restent valables. Les égalités restent valables également car on a arrondi les débit des allocations et pas directement les variables du programme linéaire. On peut donc en déduire un ordonnancement valide et qui vérifie les inégalités suivantes :

$$\rho_{\text{opt}} \geqslant \rho(T) = \sum_{i=1}^{q} \frac{r_i(T)}{T} \geqslant \sum_{i=1}^{q} \frac{\alpha_i T - 1}{T} \geqslant \rho_{\text{opt}} - \frac{q}{T}, \tag{5.16}$$

ce qui nous permet de montrer le résultat suivant :

Proposition 5.1. *Pour toute période T, il est possible de trouver un ordonnancement périodique de période T et dont le débit $\rho(T)$ tend vers l'optimal quand T devient grand.*

Cette approximation de $\alpha_i T$ est extrèmement grossière et il y a certainement plus astucieux. Quand T est petit, il est toujours possible de résoudre dans $\mathbb{N}$ le programme linéaire (5.11) dans lequel toutes les contraintes temporelles ont été multipliées par T (c'est-à-dire où les $\leqslant 1$ ont été remplacés par des $\leqslant T$). Cette solution n'est alors pas polynomiale mais parfaitement exacte.

Une autre approche consiste, une fois les différentes allocations possibles déterminées à l'aide du programme linéaire (5.11), à reformuler le choix des coefficients sous la forme d'un problème de sac à dos multidimensionnel. Ce dernier peut alors être approximé puisqu'il existe pour ce type de problème un schéma d'approximation polynomial [35].

5.4.3 Ordonnancement dynamique

La méthode de décomposition en couplage utilisée en section 5.3.3 et 4.3.5 pour s'assurer que les communications sont compatibles n'est pas vraiment utilisable en pratique. En effet, elle nécessiterait une synchronisation régulière et une horloge globale, ce qui n'est pas réaliste sur les plates-formes que nous visons.

Si on note $S_1, \ldots, S_q$ les allocations obtenues dans la section 5.3.2 et $\alpha_1, \ldots, \alpha_q$ leur débit, l'algorithme 2.1 présenté dans le chapitre 2 pour réaliser un équilibrage 1D permet de proposer une tentative de réponse à ce problème. À chaque instant le nombre d'allocations de chaque type utilisé respecte les proportions $\alpha_1, \ldots, \alpha_q$ de la meilleure façon possible. Évidemment, rien ne garantit que deux processeurs ne vont pas entrer en conflit sur une ressource (c'est-à-dire essayer de communiquer avec un même autre processeur). Cependant, en utilisant des communications asynchrones, on supprime les temps d'inactivité dûs à ces problèmes de synchronisation sans changer pour autant le modèle de plate-forme car ces hypothèses continuent d'être vérifiée au niveau matériel. Cela revient à effectuer un changement de granularité et à laisser le matériel gérer au mieux les conflits. De plus, cette approche a l'avantage de permettre de s'adapter aux micro-variations de charge et de réduire le nombre de tâches en attente sur chaque processeur (puisque cela revient à modifier la granularité). Une telle approche a déjà été utilisée pour l'ordonnancement de tâches indépendantes sur des arborescences par Carter, Casanova, Ferrante et Kreaseck [31]. Nous avons l'intention, dans un avenir proche, d'approfondir plus avant cette question ainsi que le problème de l'approximation présenté dans la section précédente, nottamment à l'aide du logiciel SimGrid présenté dans le chapitre 7.

5.5 Conclusion

Les travaux [99, 98, 65, 61, 113, 15, 13] ont porté sur le paradigme maître-esclave pour les plates-formes hétérogènes mais tous ces travaux ne concernent que les tâches indépendantes sans parallélisme interne. Les premiers travaux à notre connaissance à mélanger hétérogénéité de la plate-forme et contraintes de dépendances sont ceux de Taura et Chien [108]. Ils s'intéressent également au problème du calcul du régime permanent optimal sur une plate-forme hétérogène mais restreignent l'exécution de chacun des sommets du graphe de tâches sur un processeur donné. Leur problème se ramène donc au plongement du graphe de tâches dans le graphe de plate-forme. Le problème est donc beaucoup plus compliqué (NP-complet par réduction à Max-Cut) et ne permet pas d'exploiter l'intégralité de la plate-forme. En effet, si le graphe de tâches comporte n sommets, au plus n processeurs seront utilisés. Nous avons montré qu'en relâchant cette hypothèse, il est possible de calculer pour une large classe de graphes de tâches le régime permanent optimal et de donner un ordonnancement l'atteignant. C'est donc un premier pas vers l'extension des travaux sur le parallélisme mixte (de tâches et de données) [105, 34, 91, 7, 106] à un cadre hétérogène. Ces résultats pourraient directement

être utilisés dans des environnements comme DataCutter [17, 104].

Chapitre 6

Ordonnancement de tâches divisibles

6.1 Introduction

Après avoir considéré l'ordonnancement de tâches indépendantes (recélant éventuellement du parallélisme interne) en régime permanent, nous adaptons le cadre de travail ainsi développé aux applications constituées de tâches indépendantes infiniment fractionnables. Ce modèle plus connu sous le nom des *tâches divisibles* a été largement étudié ces dernières années et rendu populaire par le livre de Bharadwaj, Ghose, Mani et Robertazzi [18]. Une tâche divisible est une tâche qui peut être arbitrairement répartie sur un nombre quelconque de processeurs. Cela correspond à une tâche parfaitement parallèle. Ce modèle couvre un grand nombre d'applications scientifiques comme le filtrage de Kalman [102], certains traitements d'image [74], la diffusion de flux vidéo et multimédia [2, 3], la recherche d'information dans une base de données [49, 21], ou encore le traitement de gros fichiers distribués [111] (d'autres exemples peuvent être trouvés dans [18]).

Sur le plan pratique, le modèle des tâches divisibles fournit un cadre de travail à la fois simple et réaliste pour étudier la distribution de tâches indépendantes sur une plate-forme hétérogène. La granularité des tâches peut être arbitrairement choisie, ce qui permet une grande flexibilité lors d'une mise en œuvre réelle. Sur le plan théorique, le succès rencontré par ce modèle provient du fait que de nombreux problèmes peuvent être résolus de façon analytique. Des algorithmes optimaux et des formes closes existent pour les instances simples de ce modèle, ce qui n'est pas du tout le cas dans les modèles plus classique de l'ordonnancement qui conduisent très rapidement à des résultats de NP-complétude [58, 52] et d'inapproximabilité [39, 5].

Dans la majeure partie de ce chapitre, la plate-forme est constituée d'un maître (noté P_0 pour simplifier l'écriture des équations) et de p esclaves $P_1, \ldots, P_p$. Comme dans le chapitre 3, on suppose que les communications entre le maître et les esclaves sont exclusives. On peut cependant distinguer différents scénarios pour les esclaves, selon qu'ils sont capables de calculer pendant qu'ils reçoivent des tâches du maître (*full overlap*) ou non. Cette hypothèse est couramment utilisée dans la littérature car elle semble bien s'appliquer aux capacités des plates-formes actuelles. Néanmoins, les résultats de ce cha-

pitre restent valide pour les deux hypothèses, avec ou sans recouvrement. Le coût d'une communication de taille α_i entre le maître et un esclave P_i est constitué d'une latence G_i et d'un terme linéaire $\alpha_i g_i$, où g_i est l'inverse de la bande passante du lien connectant le maître à P_i. Dans le modèle d'origine de Robertazzi [18], toutes les latences sont nulles et le coût est donc complètement linéaire. Cependant, les latences sont loin d'être négligeables dans les architectures actuelles [42] et il est donc plus réaliste de les modéliser avec un coût affine de la forme $G_i + \alpha_i g_i$ pour un message de taille α_i. Enfin, notons que quand les G_i et les g_i sont constants, on se retrouve dans le cas d'un bus [102].

Dans la situation la plus simple, le maître distribue les fractions de travail aux esclaves en une seule tournée (*round* ou *installment* dans [18]). Il n'y a donc qu'une seule communication entre le maître et chaque esclave. Aussi surprenant que cela puisse paraître, la solution optimale pour une étoile hétérogène n'était pas connue, même avec un coût de communications linéaire. La section 6.4 décrit la solution optimale, étendant ainsi les résultats de [102] pour les réseaux d'interconnexion en bus.

Quand la quantité de travail à distribuer devient conséquente, distribuer le travail en une seule tournée n'est cependant pas une solution efficace en raison du temps d'inactivité forcée des derniers processeurs recevant leur part de travail. Pour minimiser le temps d'exécution, il est nécessaire de distribuer le travail en plusieurs fois (*multi-round*) : les communications sont ainsi plus courtes et pipelinées, ce qui permet aux esclaves de commencer à travailler plus tôt. Trouver une solution efficace devient alors un problème délicat en raison des nombreux paramètres à déterminer : quel est le bon nombre de tournées ? quelle est la taille des tournées ? dans quel ordre les communications doivent-elles être effectuées ? Intuitivement, la taille des tournées doit être petite au début, de façon à ce que les esclaves puissent commencer à travailler le plus tôt possible. Puis, la taille des tournées doit augmenter jusqu'à atteindre un régime permanent permettant d'optimiser l'usage de la bande passante du réseau tout en minimisant le coût des latences. Dans [18, chapitre 10], aucune méthode n'est proposée pour évaluer une valeur correcte du nombre de tournées. Plus récemment, Altilar et Paker [2, 3] et Casanova et Yang [118] ont proposé des algorithmes en plusieurs tournées et exprimé analytiquement leurs performances. Nous présentons rapidement ces algorithmes ainsi que d'autres dans la section 6.3. Aucun résultat d'optimalité concernant les algorithmes en plusieurs tournées n'a jamais été démontré pour une plate-forme hétérogène. Ce chapitre comble ce manque : en section 6.5, nous présentons un algorithme de distribution périodique en plusieurs tournées dont nous établissons l'optimalité asymptotique. Ce résultat s'applique également aux plates-formes dont le graphe d'interconnexion est un graphe quelconque.

Ce chapitre est donc organisé de la façon suivante. Nous commençons par formaliser précisément dans la section 6.2 les différents modèles que nous utilisons. Ensuite nous présentons les résultats et travaux antérieurs aux nôtres en section 6.3. La section 6.4 traite des algorithmes qui procèdent en une seule tournée et la section 6.5 des algorithmes en plusieurs tournées. Quelques simulations comparant nos algorithmes avec des algorithmes déjà existant sont présentées en section 6.7 et la section 6.8 conclue ce chapitre par quelques remarques.

6.2 Modèles

Nous supposons disposer d'une quantité de travail w_{total} parfaitement divisible. Il est courant de considérer que le maître lui même ne traite pas de tâches car le cas contraire se modélise simplement en ajoutant un esclave fictif dont le temps de communication est nul. La modélisation des coûts de calcul par une fonction linéaire est courante dans la littérature. Un temps $\alpha_i w_i$ est donc nécessaire à l'esclave P_i pour traiter une quantité α_i de tâches. Notons cependant que Casanova et Yang proposent dans [118] d'ajouter un coût de mise en marche, ou latence de calcul, si bien que le temps nécessaire au processeur P_i pour traiter une quantité α_i de travail s'écrit $W_i + \alpha_i w_i$. Ils soulignent l'importance de ce terme, nécessaire pour obtenir une modélisation correcte ainsi que des résultats pertinents pour des applications de type *data-sweep* [32]. Dans la suite, à part pour le cas des distributions en une seule tournée, nous nous contentons des coûts de calcul linéaire.

La modélisation des communications est plus délicate et différents modèles ont été proposés dans la littérature. Dans l'article original de Robertazzi [18], le coût de communication était également linéaire. Le temps de communication nécessaire au transfert de α_i tâches s'écrivait donc $\alpha_i g_i$. Si cette modélisation est acceptable pour les gros messages, elle devient irréaliste pour les petites communications. Par exemple dans [18], les auteurs reconnaissent qu'avec un tel modèle, la solution optimale pour un algorithme en plusieurs tournées consiste à utiliser des messages de taille infiniment petite. Les latences G_i ont été introduites par Drozdowski dans [49] et sont désormais couramment utilisées : le temps nécessaire au transfert de α_i tâches s'écrit donc $G_i + \alpha_i g_i$. Un modèle encore plus précis a été proposé par Rosenberg [94] et utilisé par Casanova et Yang [118]. Ils suggèrent d'utiliser l'expression $G'_i + \alpha_i g_i + G''_i$ où la première latence G'_i n'est pas recouvrable, alors que la seconde, G''_i est recouvrable par la communication suivante. Le maître peut donc commencer à envoyer le message suivant $G'_i + \alpha_i g_i$ unité de temps plus tard, alors que l'esclave ne peut commencer à travailler avant le temps $G'_i + \alpha_i g_i + G''_i$. Une fois de plus, nous nous restreignons au cas des latences non-recouvrables, même si nous expliquons comment les prendre en compte dans les algorithmes en plusieurs tournées. Enfin, notons également l'existence de travaux [67, 64] où le coût de communication est constant.

Il reste à décider de la quantité de communications et de calculs qui peuvent être recouvertes. Dans le modèle avec recouvrement, chaque esclave est capable de recevoir le prochain lot de tâches pendant qu'il s'occupe de calculer le lot précédant. Ceci correspond au modèle des processeurs *equipped with a front-end* de [18]. Dans le modèle sans recouvrement, chaque esclave effectue ses communications et ses calculs séquentiellement. Bien évidemment, cette distinction ne s'applique qu'aux algorithmes en plusieurs tournées et nous démontrerons les résultats pour les deux modèles dans la section 6.5, avec et sans recouvrement.

Le dernier point à fixer concerne la capacité du maître à traiter ou non plusieurs communications simultanées avec ses esclaves. À quelques exceptions prêt, le modèle *1-port* est utilisé : les communications sont exclusives et le maître ne peut communiquer

qu'avec un seul de ses esclaves à la fois[1]. Cependant, comme le font remarquer Casanova et Yang [118], si le modèle *1-port* est assez bien adapté à un réseau local, un modèle *multi-port* serait probablement plus adapté à un réseau de plus grande échelle.

Nous utiliserons donc le modèle suivant au long de ce chapitre :

i) Modèle *1-port* pour le maître : les communications avec les esclaves sont exclusives les unes des autres.

ii) Les esclaves sont capables de recouvrir leurs calculs par des communications.

iii) Le traitement d'une quantité α_i de tâches nécessite un temps linéaire $\alpha_i w_i$ (ou affine dans certains cas : $W_i + \alpha_i w_i$).

iv) Le temps nécessaire au transfert de α_i tâches est modélisé par une fonction affine : $G_i + \alpha_i g_i$.

Nous discuterons quelques extensions de ce modèles lorsque nous traiterons des algorithmes en plusieurs tournées dans la section 6.5.

6.3 Travaux connexes

Ce tour d'horizon est séparé en deux parties : la première concerne les algorithmes de distribution en une seule tournée et la seconde les algorithmes en plusieurs tournées. Nous nous restreignons aux plates-formes maître/esclave classiques telles que les étoiles. Des résultats concernant les interconnexions en arbre peuvent être trouvés dans [18] et des résultats pour les hypercubes dans [49].

6.3.1 Distribution en une seul tournée

Le premier problème auquel on est confronté avec les distributions en une seule tournée porte sur l'ordre dans lequel les tâches doivent être envoyées aux esclaves. Une fois cet ordre défini, les communications étant exclusives, le schéma général d'une distribution optimale dans le cas d'un bus est représenté figure 6.1. Il reste donc à déterminer les quantités de travail α_i devant être allouées à chaque processeur P_i, l'objectif étant de minimiser le temps d'exécution total.

Dans le cas d'une plate-forme où les communications sont homogènes (comme un bus où $g_i = g$ pour tout i), en utilisant un modèle de coût linéaire pour les calculs et en ne modélisant pas de latence pour les communications ($G_i = 0$ pour tout i), Bataineh, Hsiung et Robertazzi [9, 102] parviennent à fournir une solution optimale ainsi qu'une forme close pour le temps de calcul T_f. Cette solution est étonnamment simple. Notons α_i la quantité de travail assignée à chaque esclave P_i et T_i l'instant auquel P_i commence à traiter les tâches qui lui ont été assignées. On a donc $\sum_{i=1}^{p} \alpha_i = w_{\text{total}}$ et $T_f = \max_i(T_i + \alpha_i w_i)$.

[1] sauf pour la petite période correspondant à une latence recouvrable quand cette dernière est modélisée bien sûr.

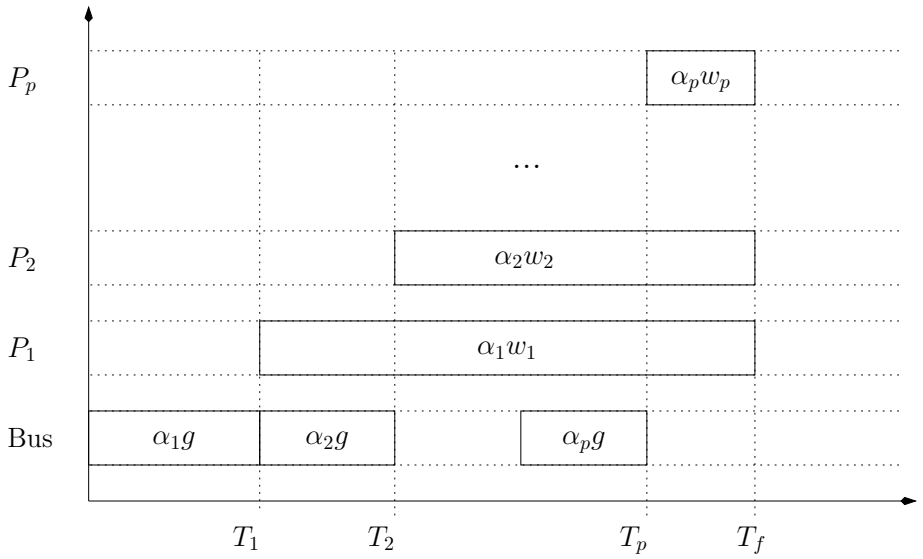


FIG. 6.1 – Forme générale de la distribution de tâches divisible sur une plate-forme de type bus sans latence ($g_i = g$ et $G_i = 0$). Tous les esclaves doivent terminer de travailler au même instant T_f.

On peut d'abord montrer que tous les processeurs doivent terminer de travailler au même instant, c'est-à-dire que pour tout i on a $T_f = T_i + \alpha_i w_i$. En effet, dans le cas contraire, il serait possible de transférer un peu de travail d'un processeur trop occupé vers un processeur inactif de façon à réduire T_f. On peut donc en déduire le système d'équations suivant si les communications ont lieu successivement vers $P_1, \ldots, P_p$:

$$\begin{cases} T_f - T_i = \alpha_i w_i, & \forall 1 \leqslant i \leqslant p \\ T_{i+1} - T_i = \alpha_{i+1} g & \forall 1 \leqslant i \leqslant p-1 \end{cases}$$

On peut en tirer assez simplement des formes closes pour les α_i et pour T_f. Ces formules sont néanmoins un peu compliquées et nous renvoyons à [102] pour leur expression algébrique exacte. Le point le plus surprenant porte sur le fait que l'expression de T_f ne dépend pas de l'ordre dans lequel les communications sont faites et que n'importe quel ordre est donc optimal.

Un peu plus tard, Charcranoon, Robertazzi et Luryi [36] ont partiellement étendu ces résultats au cas où les communications sont hétérogènes (plate-forme en étoile) : les coûts de communication sont toujours linéaires mais les g_i peuvent avoir des valeurs différentes. Les résultats ne sont cependant pas aussi impressionnants que dans le cas du bus. Le résultat principal porte sur l'établissement d'une forme close pour les α_i et pour T_f une fois l'ordre des communications fixé. Mais cette fois, l'expression de T_f dépend fortement de cet ordre et le résultat sur le fait que tous les processeurs doivent terminer de travailler au même instant T_f n'est plus valide pour tous les ordres. À notre connaissance, l'ordre optimal des communications n'était pas connu et les résultats de la section 6.4 sont donc les premiers de ce genre.

En ce qui concerne les coûts de communication affines plutôt que linéaires, un certain nombre de résultats ont été publiés, parmi lesquels [74, 49, 21, 94]. En 1997, Drozdowski [49] déclare ouverte la complexité de la minimisation du temps nécessaire au traitement de tâches sur une étoile générale (avec des G_i et des g_i distincts les uns des autres). Ce problème est toujours ouvert. Soulignons que la formulation de Drozdowski [49] à base de programmation linéaire mixte est intéressante. Dans le programme suivant, $x_{i,j}$ est un booléen valant 1 si P_i est choisi pour la $j^{\text{ème}}$ communication du maître :

$$
\begin{array}{l}
\text{MINIMISER } T_f, \\
\text{SOUS LES CONTRAINTES} \\
\left\{
\begin{array}{ll}
\text{(6.1a)} & \alpha_i \geqslant 0, \text{ pour } 1 \leqslant i \leqslant p \\
\text{(6.1b)} & \displaystyle\sum_{i=1}^{p} \alpha_i = w_{\text{total}} \\
\text{(6.1c)} & x_{i,j} \in \{0,1\} \text{ pour } 1 \leqslant i,j \leqslant p \\
\text{(6.1d)} & \displaystyle\sum_{i=1}^{p} x_{i,j} = 1 \text{ pour } 1 \leqslant j \leqslant p \\
\text{(6.1e)} & \displaystyle\sum_{j=1}^{p} x_{i,j} = 1 \text{ pour } 1 \leqslant i \leqslant p \\
\text{(6.1f)} & \displaystyle\sum_{i=1}^{p} x_{1,i}(G_i + \alpha_i g_i + \alpha_i w_i) \leqslant T_f \text{ (première communication)} \\
\text{(6.1g)} & \displaystyle\sum_{k=1}^{j-1}\sum_{i=1}^{p} x_{k,i}(G_i + \alpha_i g_i) + \sum_{i=1}^{p} x_{j,i}(G_i + \alpha_i g_i + \alpha_i w_i) \leqslant T_f \text{ pour } 2 \leqslant j \leqslant p \\
 & (j^{\text{ème}} \text{ communication})
\end{array}
\right.
\end{array}
\tag{6.1}
$$

L'équation (6.1d) signifie qu'exactement un processeur est actif durant la $j^{\text{ème}}$ communication et l'équation (6.1e) signifie que chaque processeur est actif exactement une fois. L'équation (6.1f) est un cas particulier de l'équation (6.1g) qui exprime le fait que le processeur sélectionné pour la $j^{\text{ème}}$ communication doit attendre que la communication avec son prédécesseur soit terminée avant que la sienne et ses calculs puissent commencer. Comme le fait remarquer Drozdowski [49], il est parfaitement possible que ce programme n'ait pas de solution. De plus on n'est pas assuré de trouver une solution optimale à notre problème initial puisqu'il se peut parfaitement que la solution n'utilise pas toutes les ressources. Néanmoins, il est parfaitement possible de formuler la sélection des ressources

en relâchant les contraintes sur la permutation de la façon suivante :

$$
\begin{array}{l}
\text{MINIMISER } T_f, \\
\text{SOUS LES CONTRAINTES} \\
\left\{
\begin{array}{ll}
\text{(6.2a)} & \alpha_i \geqslant 0, \text{ pour } 1 \leqslant i \leqslant p \\
\text{(6.2b)} & \displaystyle\sum_{i=1}^{p} \left(\sum_{j=1}^{p} x_{i,j} \right) \alpha_i = w_{\text{total}} \\
\text{(6.2c)} & x_{i,j} \in \{0,1\} \text{ pour } 1 \leqslant i,j \leqslant p \\
\text{(6.2d)} & \displaystyle\sum_{i=1}^{p} x_{i,j} \leqslant 1 \text{ pour } 1 \leqslant j \leqslant p \\
\text{(6.2e)} & \displaystyle\sum_{j=1}^{p} x_{i,j} \leqslant 1 \text{ pour } 1 \leqslant i \leqslant p \\
\text{(6.2f)} & \displaystyle\sum_{i=1}^{p} x_{1,i}(G_i + \alpha_i g_i + \alpha_i w_i) \leqslant T_f \text{ (première communication)} \\
\text{(6.2g)} & \displaystyle\sum_{k=1}^{j-1} \sum_{i=1}^{p} x_{k,i}(G_i + \alpha_i g_i) + \sum_{i=1}^{p} x_{j,i}(G_i + \alpha_i g_i + \alpha_i w_i) \leqslant T_f \text{ pour } 2 \leqslant j \leqslant p \\
 & \qquad (j^{\text{ème}} \text{ communication})
\end{array}
\right.
\end{array}
\tag{6.2}
$$

On est alors en mesure d'obtenir la solution optimale mais à un coût potentiellement exponentiel.

6.3.2 Distributions en plusieurs tournées

Un certain nombre d'algorithmes en plusieurs tournées ont été proposés dans la littérature [18, 2, 3, 118] mais la majorité n'ont été validés que par des simulations ou des expériences plutôt qu'avec des formules analytiques. Ce n'est pas surprenant au vu de la difficulté qu'il y a à déterminer le bon nombre de tournées. Si les petites tournées minimisent les périodes d'inactivité du début et permettent un meilleur recouvrement des calculs et des communications, les grandes tournées permettent de diminuer l'impact des latences dans le coût des communications.

Techniquement, une tournée est définie comme une séquence de communications vers les différents esclaves et il n'est pas évident de décider s'il faut utiliser tous les esclaves ou non. Quand bien même un sous-ensemble devrait être utilisé, il n'y a aucune raison pour que ce sous-ensemble reste le même au fil des tournées.

Notons $W^{(k)}$ la quantité de travail distribuée aux différents esclaves durant la $k^{\text{ème}}$ tournée : $W^{(k)} = \sum_{i=1}^{p} \alpha_i^{(k)}$, où $\alpha_i^{(k)}$ représente la quantité de travail allouée au processeur P_i durant la $k^{\text{ème}}$ tournée. Intuitivement, les $W^{(k)}$ doivent être petits durant les premières

tournées, puis atteindre une valeur adéquate et décroître durant les dernières tournées. Casanova et Yang [118] proposent que les $W^{(k)}$ suivent une progression géométrique, et qu'à l'intérieur de chaque tournée, tous les processeurs impliqués calculent pendant la même durée[2]. Ces hypothèses leur permettent d'exprimer analytiquement le temps d'exécution et le nombre total de tournées peut donc être déterminé numériquement. Les techniques employées sont donc bien plus originales que les précédentes. Cependant, cette approche comporte deux limitations : (i) la sélection des ressources (détermination du meilleur ensemble de processeurs impliqué dans les différentes tournées) est effectuée à l'aide d'une heuristique, et (ii) il n'y a pas de raison fondamentale de privilégier une progression géométrique des $W^{(k)}$ et n'importe quelle progression monotone et suffisamment régulière pourrait être adoptée.

En section 6.5, nous proposons un algorithme périodique dont nous démontrons l'optimalité asymptotique. Cet algorithme est assez simple car les différentes tournées sont toutes identiques. Le nombre optimal de tournées, la sélection des ressources, la taille des tournées et la quantité de travail assignée à chaque processeur au cours de chaque tournée sont résolus à l'aide d'un programme linéaire en nombres rationnels.

6.3.3 Prise en compte des communications retour

Dans un certain nombre de cas, il est possible d'ignorer la communication du résultat d'un calcul car sa taille est négligeable en comparaison à celle des fichiers d'entrées. Les algorithmes présentés en section 6.5 s'étendent simplement au cas où les communications retours ne peuvent plus être ignorées, tout en conservant leur propriété d'optimalité asymptotique. La résolution exacte du problème est cependant plus délicate car il convient de déterminer à la fois l'ordre des communications aller et l'ordre des communications retour. Les sections 3.3 et 3.4 montrent bien, avec un autre modèle, le saut de difficulté qui apparaît alors. Relativement peu de personnes semblent s'être intéressées à ce problème. On peut néanmoins noter les travaux de Boufflet et Carlier [23] pour des plates-formes de type bus avec des coûts de communication et de calcul affines. Après avoir analysé le problème et prouvé la dominance de certains ordres d'émission et de réception, ils proposent une méthode *branch-and-bound*, potentiellement exponentielle, pour résoudre le problème de façon exacte une fois la sélection de ressources et le nombre de tournées fixé.

6.4 Distribution en une seule tournée

Dans cette section, nous proposons une nouvelle approche de la démonstration de l'optimalité des distributions en une seule tournée. Cette approche nous permet de retrouver des résultats connus et d'en démontrer de nouveaux.

[2]La progression géométrique est interrompue à l'approche de la fin, de façon à ce que tous les processeurs terminent en même temps

6.4.1 Comparaison dans le cas général

L'idée est de comparer la quantité de travail effectuée par les deux premiers esclaves. Pour simplifier les notations, supposons que P_1 et P_2 soient les deux premiers esclaves à recevoir leur travail du maître. Il y a donc deux ordres possibles (voir figure 6.2). Pour chaque ordre, nous déterminons le nombre total $\alpha_1 + \alpha_2$ de tâches traitées en T unités de temps, ainsi que l'occupation totale t_2 du réseau pendant cet intervalle de temps. Nous notons avec un exposant (A) (resp. (B)) les différentes quantités du premier (resp. second) ordre.

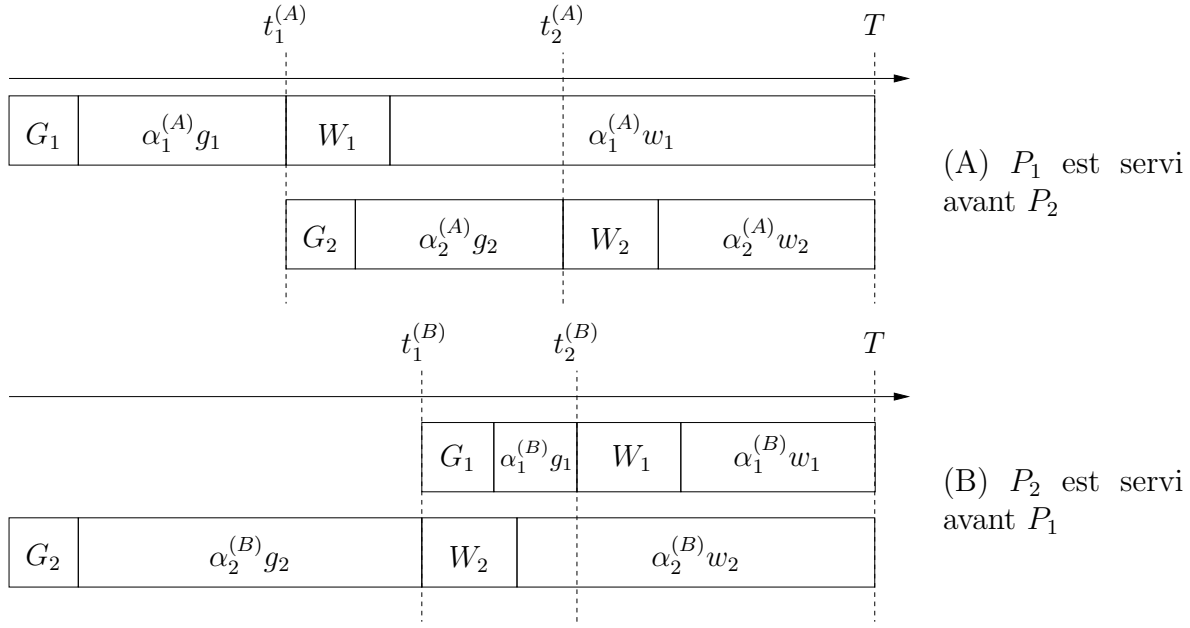


FIG. 6.2 – Comparaison des deux ordres possibles.

Commençons par déterminer les différentes quantité $\alpha_1^{(A)}, \alpha_2^{(A)}, t_1^{(A)}$ et $t_2^{(A)}$ du premier ordre de la figure 6.2 :

– De l'égalité $G_1 + W_1 + \alpha_1^{(A)}(g_1 + w_1) = T$, on déduit que :

$$\alpha_1^{(A)} = \frac{T - (G_1 + W_1)}{g_1 + w_1}. \tag{6.3}$$

– En utilisant le fait que $t_1^{(A)} = G_1 + \alpha_1^{(A)} g_1$, on en déduit (en réutilisant l'égalité (6.3)) que :

$$t_1^{(A)} = \frac{G_1 w_1 + T g_1 - W_1 g_1}{g_1 + w_1}. \tag{6.4}$$

– En utilisant l'égalité $G_1 + \alpha_1^{(A)} g_1 + G_2 + W_2 + \alpha_2^{(A)}(g_2 + w_2) = T$ et l'équation (6.3), on en déduit que :

$$\alpha_2^{(A)} = \frac{T - (G_2 + W_2)}{g_2 + w_2} - \frac{T g_1 + G_1 w_1 - W_1 g_1}{(g_1 + w_1)(g_2 + w_2)}. \tag{6.5}$$

– Enfin, en combinant l'égalité $t_2^{(A)} = G_1 + G_2 + \alpha_1^{(A)} g_1 + \alpha_2^{(A)} g_2$, et les équations (6.3) et (6.5), on trouve l'égalité suivante :

$$\begin{aligned} t_2^{(A)} =& (G_1 + G_2) + \frac{g_1(T - G_1 - W_1)}{g_1 + w_1} + \frac{g_2(T - G_2 - W_2)}{g_2 + w_2} \\ & - \frac{T g_1 g_2}{(g_1 + w_1)(g_2 + w_2)} - \frac{g_2(G_1 w_1 - W_1 g_1)}{(g_1 + w_1)(g_2 + w_2)}. \end{aligned} \tag{6.6}$$

Ainsi, le nombre total de tâches traitées en temps T est égale à (par (6.3) et (6.5))

$$\alpha_1^{(A)} + \alpha_2^{(A)} = \frac{T - (G_1 + W_1)}{g_1 + w_1} + \frac{T - (G_2 + W_2)}{g_2 + w_2} - \frac{T g_1 + G_1 w_1 - W_1 g_1}{(g_1 + w_1)(g_2 + w_2)},$$

et le temps d'occupation total du réseau est égal à (par (6.5)) :

$$\begin{aligned} t_2^{(A)} =& (G_1 + G_2) + \frac{g_1(T - G_1 - W_1)}{g_1 + w_1} + \frac{g_2(T - G_2 - W_2)}{g_2 + w_2} \\ & - \frac{T g_1 g_2}{(g_1 + w_1)(g_2 + w_2)} - \frac{g_2(G_1 w_1 - W_1 g_1)}{(g_1 + w_1)(g_2 + w_2)}. \end{aligned}$$

Ces expressions sont relativement compliquées mais elles se simplifient lorsque l'on exprime les différences entre la situation (A) et la situation (B). En effet, de la même façon que précédemment, on obtient :

$$\alpha_1^{(B)} + \alpha_2^{(B)} = \frac{T - (G_1 + W_1)}{g_1 + w_1} + \frac{T - (G_2 + W_2)}{g_2 + w_2} - \frac{T g_2 + G_2 w_2 - W_2 g_2}{(g_1 + w_1)(g_2 + w_2)}$$

et

$$\begin{aligned} t_2^{(B)} =& (G_1 + G_2) + \frac{g_1(T - G_1 - W_1)}{g_1 + w_1} + \frac{g_2(T - G_2 - W_2)}{g_2 + w_2} \\ & - \frac{T g_1 g_2}{(g_1 + w_1)(g_2 + w_2)} - \frac{g_1(G_2 w_2 - W_2 g_2)}{(g_1 + w_1)(g_2 + w_2)} \end{aligned}$$

D'où les équations suivantes :

$$\left(\alpha_1^{(A)} + \alpha_2^{(A)}\right) - \left(\alpha_1^{(B)} + \alpha_2^{(B)}\right) = \frac{(G_2 w_2 - G_1 w_1) + (W_1 g_1 - W_2 g_2) + T(g_2 - g_1)}{(g_1 + w_1)(g_2 + w_2)} \tag{6.7}$$

et

$$t_2^{(A)} - t_2^{(B)} = \frac{g_1 G_2 w_2 - g_2 G_1 w_1 + g_1 g_2 (W_1 - W_2)}{(g_1 + w_1)(g_2 + w_2)} \tag{6.8}$$

6.4.2 Étoile et coût linéaire

On s'intéresse dans cette section à la version la plus simple du problème d'allocation de tâches divisibles. La plate-forme est une étoile et les coûts sont linéaires. Le temps nécessaire au transfert de α_i tâches du maître vers le processeur P_i est donc égal à $\alpha_i g_i$ et le temps nécessaire à leur calcul est égal à $\alpha_i w_i$. Le maître n'est capable de communiquer qu'avec une seule personne à la fois. Ce problème est appelé ÉtoileLinéaire.

Nous commençons par montrer que dans une solution optimale, tous les esclaves participent au calcul et finissent de travailler en même temps. Puis, forts de ce nouveau résultat, nous donnons un algorithme optimal pour résoudre le problème de l'allocation de tâches indépendantes sur une étoile quand les coût sont linéaires

Nous commençons par montrer que dans une solution optimale tous les esclaves participent au calcul

Lemme 6.1. *Dans une solution optimale, tous les esclaves participent au calcul.*

Démonstration. Supposons qu'il existe une solution optimale où au moins un processeur ne travaille pas du tout. On peut supposer sans perte de généralité que α_1 tâches sont envoyées à P_1, puis α_2 tâches à P_2, et ainsi de suite, jusqu'à envoyer α_n tâches à P_n. Dans cette solution, au moins un des α_i est nul. Notons k le plus grand indice tel que $\alpha_k = 0$.

Cas où $k < n$. Considérons l'allocation suivante : on envoie les données dans l'ordre suivant $P_1, \ldots, P_{k-1}, P_{k+1}, \ldots, P_n, P_k$ et pour tout $i \neq k$, on pose $\alpha'_i = \alpha_i$. Cette allocation est clairement valide puisque P_k ne traitait pas non plus de tâches dans la solution initiale. Par construction, on sait que $\alpha_n \neq 0$ et donc que le réseau n'est pas utilisé pendant les $\alpha_n w_n$ unité de temps. Il est donc possible de calculer $\frac{\alpha_n w_n}{g_k + w_k}$ tâches supplémentaires, ce qui est absurde.

Cas où $k = n$. Considérons l'allocation suivante : on envoie les données dans l'ordre suivant $P_1, \ldots, P_n$ et pour tout $i \neq k$ on pose $\alpha'_i = \alpha_i$. Notons k' le plus grand indice tel que $\alpha_{k'} > 0$. Par construction, le réseau n'est pas utilisé pendant les $\alpha_{k'} w_{k'}$ unité de temps. Comme précédemment, il est donc possible de traiter $\frac{\alpha_{k'} w_{k'}}{g_k + w_k}$ tâches supplémentaires, ce qui est absurde.

Ainsi, dans une solution optimale tous les esclaves participent au calcul. ■

Lemme 6.2. *Dans une solution optimale, tous les processeurs finissent de travailler au même instant.*

Démonstration. Considérons une solution optimale. On peut supposer sans perte de généralité que α_i tâches sont envoyées à P_1, puis α_2 tâches à P_2, et ainsi de suite, jusqu'à envoyer α_n tâches à P_n. On sait également grâce au lemme précédent que tous les α_i sont strictement positifs. Les α_i sont alors une solution optimale du programme linéaire

suivant :

$$\begin{array}{l}\textsc{Maximiser } \sum \beta_i, \\ \textsc{sous les contraintes} \\ \begin{cases} (6.9a) \quad \forall i, \beta_i \geqslant 0 \\ (6.9b) \quad \forall i, \displaystyle\sum_{k=1}^{i} \beta_k g_k + \beta_i w_i \leqslant T \end{cases}\end{array} \tag{6.9}$$

Il est connu [95, chapitre 11] qu'un des sommets du polyèdre $\mathcal{P}$ induit par les inégalités d'un programme linéaire est une solution optimale. Notons $S_1 = (\beta_1, \ldots, \beta_n)$ un tel point. S_1 étant dans un espace de dimension n, au moins n des inégalités sont des égalités. Comme on sait grâce au lemme 6.1 que les inégalités (6.9a) sont toutes strictes, on en déduit qu'en S_1, les inégalités (6.9b) sont en fait des égalités, c'est-à-dire que :

$$\forall i, \sum_{k=1}^{i} \beta_k g_k + \beta i w_i = T.$$

Autrement dit, il existe un solution optimale telle que tous les esclaves terminent de travailler au même instant.

Notons $S_2 = (\alpha_1, \ldots, \alpha_n)$ la solution optimale initiale et supposons que tous les processeurs ne terminent pas au même instant. Considérons la fonction f suivante :

$$f : \begin{cases} \mathbb{R} & \to & \mathbb{R}^n \\ x & \mapsto & S_1 + x(S_2 - S_1) \end{cases}$$

S_1 et S_2 étant deux solutions optimales distinctes, on a $\sum \beta_i = \sum \alpha_i$. En notant $f(x) = (\gamma_1(x), \ldots, \gamma_n(x))$, on peut remarquer que

$$\forall x, \sum_i \gamma_i(x) = \sum_i \alpha_i = \sum_i \beta_i.$$

Tous les points $f(x)$ appartenant à $\mathcal{P}$ sont donc des solutions optimales de ÉtoileLinéaire.

$\mathcal{P}$ étant un polyèdre convexe (au vu des contraintes), pour tout $x \in [0, 1]$, $f(x)$ appartient à $\mathcal{P}$ et est donc une solution optimale de ÉtoileLinéaire. Notons x_0 la plus grande valeur des $x \geqslant 1$ tels que $f(x)$ appartienne à $\mathcal{P}$. $f(x_0)$ est alors une solution optimale de ÉtoileLinéaire, appartient à la frontière de $\mathcal{P}$ et donc au moins une des contraintes linéaires de (6.9) est une égalité. Comme S_2 est situé entre S_1 et $f(x_0)$ et que S_2 est différent de S_1, on en déduit que $f(x_0)$ est donc différent de S_1. On sait donc qu'une de ces égalités est différente de celles vérifiées en S_1 (sinon S_1 et S_2 appartiendrait au même hyperplan, la droite (S_1, S_2) aussi, et donc $f(x)$ pour $x > x_0$ ne violerait pas cette contrainte) et donc qu'il existe un i tel que $\gamma_i(x_0) = 0$, ce qui est absurde puisque $f(x_0)$ est une solution optimale et que dans une solution optimale tous les esclaves travaillent.

On a donc $S_1 = S_2$. Pour un ordre d'émission fixé, la solution optimale est donc unique et dans cette solution tous les processeurs finissent de travailler au même instant. ■

Théorème 6.1. *L'allocation optimale de ÉtoileLinéaire est obtenue en envoyant les données aux esclaves dans l'ordre des g_i croissants.*

Démonstration. Cette démonstration est basée sur la comparaison de la quantité de travail effectuée par les deux premiers esclaves. Notons qu'il est possible d'utiliser ces formules parce que l'on a démontré avec les lemmes précédents que dans une solution optimale, tous les processeurs finissent en même temps. Pour simplifier les notations, supposons que P_1 et P_2 sont deux premiers esclaves à recevoir leurs données. En utilisant l'équation (6.8) où $G_1 = G_2 = 0$ et $W_1 = W_2 = 0$, on peut voir que l'ordre dans lequel les communications de P_1 et de P_2 sont effectuées n'influe pas sur le temps d'occupation du réseau. Seule la quantité de tâches traitées est donc importante et l'équation (6.7) où $G_1 = G_2 = 0$ et $W_1 = W_2 = 0$ nous permet de déterminer qu'il vaut mieux envoyer les données au processeur ayant le plus petit g_i.

Supposons que les processeurs sont déjà triés par g_i croissant. Soit σ un ordre d'émission optimal. Les tâches sont donc successivement envoyées à $P_{\sigma(1)}$,$P_{\sigma(2)}$,..., et $P_{\sigma(p)}$. Notons i, s'il existe, le plus petit indice tel que $\sigma(i) > \sigma(i+1)$. Considérons l'ordre suivant ;

$$P_{\sigma(1)}, \ldots, P_{\sigma(i-1)}, P_{\sigma(i+1)}, P_{\sigma(i)}, P_{\sigma(i+2)}, \ldots P_{\sigma(p)}.$$

Alors $P_{\sigma(1)}, \ldots, P_{\sigma(i-1)}, P_{\sigma(i+2)}, \ldots$ et $P_{\sigma(p)}$ effectuent exactement le même nombre de tâches qu'à l'origine puisque l'échange de $P_{\sigma(i+1)}$ et de $P_{\sigma(i)}$ ne modifie pas le temps d'occupation du réseau. Cependant on est capable, grâce à cet échange, de traiter $\frac{T_{\sigma(i-1)}(g_{\sigma(i)}-g_{\sigma(i+1)})}{(g_{\sigma(i+1)}+w_{\sigma(i+1)})(g_{\sigma(i)}+w_{\sigma(i)})}$ tâches supplémentaires où $T_{\sigma(i-1)}$ est le temps restant une fois les communications avec $P_{\sigma(1)}, \ldots, P_{\sigma(i-1)}$ terminées. On peut donc se ramener à une solution optimale où les tâches sont envoyées dans l'ordre des g_i croissants. ■

6.4.3 Étoile et coût affine

On s'intéresse dans cette section à la version affine du problème précédent. Le temps nécessaire au transfert de α_i tâches du maître vers le processeur P_i est donc égal à $G_i+\alpha_i g_i$ et le temps nécessaire à leur calcul est égal à $W_i + \alpha_i w_i$. Comme précédemment, le maître n'est capable de communiquer qu'avec une seule personne à la fois. Ce problème est appelé ÉtoileAffine et la difficulté principale par rapport au problème précédent réside dans la sélection des ressources.

Lemme 6.3. *Dans une solution optimale de ÉtoileAffine, tous les esclaves participant au calcul terminent leur travail au même instant.*

Démonstration. Considérons une solution optimale de ÉtoileAffine. On peut supposer sans perte de généralité que α_1 tâches sont d'abord envoyées à P_1, puis α_2 tâches sont envoyées à P_2, ... et enfin α_j tâches à P_j. $\{P_1, \ldots, P_j\}$ est donc l'ensemble des processeurs participant au calcul. Les α_i sont donc strictement positifs. Considérons le programme linéaire

suivant :

$$\begin{array}{l}\text{MAXIMISER } \sum \beta_i, \\ \text{SOUS LES CONTRAINTES} \\ \begin{cases} \forall i \leqslant j : (\mathrm{LB}(i)) \quad \beta_i \geqslant 0 \\ \forall i \leqslant j : (\mathrm{UB}(i)) \quad \displaystyle\sum_{k=1}^{i} (G_k + \beta_k g_k) + W_i + \beta_i w_i \leqslant T \end{cases} \end{array} \tag{6.10}$$

Les α_i sont clairement une solution optimale de ce programme et donc toute solution optimale de ce programme est une solution optimale de ÉtoileAffine.

Lemme 6.4. *Toute solution optimale $(\beta_1, \ldots, \beta_j)$ du programme linéaire (6.10) vérifie $\beta_k > 0$ pour tout $k < j$ et P_j termine de travailler au temps T, même s'il ne traite aucune tâche (c'est-à-dire que (UB(j)) est une égalité).*

Démonstration. Supposons qu'il existe un i tel que $\beta_i = 0$. Notons alors k le plus grand indice tel que $\beta_k = 0$.

Cas où $k < j$. Considérons la quantité de tâches traitées par les processeurs P_k et P_{k+1} ainsi que celles qui auraient pu être traitées par P_{k+1} si P_k n'avait pas été utilisé. Quand P_k et P_{k+1} sont utilisés, le réseau est utilisé par P_k et P_{k+1} pendant exactement $G_k + G_{k+1} + \beta_{k+1} g_{k+1}$ unité de temps. En enlevant P_k, β'_{k+1} le nombre de tâches qui peuvent être traitées par P_{k+1} doit vérifier les conditions suivantes :

$$G_{k+1} + \beta'_{k+1} g_{k+1} \leqslant G_k + G_{k+1} + \beta_{k+1} g_{k+1}$$

pour s'assurer que le réseau n'est pas plus utilisé que précédemment, et

$$G_{k+1} + W_{k+1} + \beta'_{k+1}(g_{k+1} + w_{k+1}) \leqslant G_k + W_k + G_{k+1} + W_{k+1} + \beta_{k+1}(g_{k+1} + w_{k+1})$$

pour s'assurer que P_{k+1} finit de travailler avant le temps T.

Ces deux conditions sont équivalentes à la suivante :

$$\beta'_{k+1} \leqslant \beta_{k+1} + \min\left(\frac{G_k}{g_{k+1}}, \frac{G_k + W_k}{g_{k+1} + w_{k+1}}\right).$$

Donc, en posant $\beta'_{k+1} = \beta_{k+1} + \min\left(\frac{G_k}{g_{k+1}}, \frac{G_k + W_k}{g_{k+1} + w_{k+1}}\right)$, on obtient une solution dans laquelle le nombre de tâches traitées est strictement plus grand que dans la solution où P_k est utilisé, ce qui est absurde.

Cas où $k = j$. En utilisant la démonstration précédente, on sait qu'aucun des P_i pour $i < j$ n'est inactif. Notons t_j le temps à partir duquel les $j-1$ premiers processeurs ont reçu leurs tâches et donc à partir duquel le réseau est libre. On a alors $T = t_j + G_j + W_j$ sinon P_j serait capable de traiter des tâches. Ainsi, dans toute solution optimale, le dernier processeur impliqué (P_j) finit de travailler au temps T, et ce même s'il ne traite aucune tâche. ■

Pour montrer que tous les processeurs participant au calcul finissent de travailler au temps T, nous utilisons, comme précédemment, quelques résultats connus sur la programmation linéaire. Un des sommets $S_1 = (\beta_1, \ldots, \beta_j)$ du polyèdre convexe $\mathcal{P}$ défini par les contraintes du programme linéaire (6.10) est une solution optimale. En S_1, au moins j des $2j$ inégalités de (6.10) sont des égalités.

Lemme 6.5. *En S_1, les contraintes (UB(i)) pour $i \leqslant j$ sont toutes des égalités.*

Démonstration. On sait grâce au lemme 6.4 que toutes les contraintes (LB(i)) sont strictes pour $i < j$. Donc, si (LB(j)) est stricte en S_1, notre propriété est vraie.
Dans le cas contraire, considérons le programme linéaire correspondant au cas où seul les $j-1$ processeurs sont utilisés. S_1 étant une solution optimale de (6.10), toute solution optimale de notre nouveau programme induit une solution optimale de (6.10). En utilisant le lemme 6.4 dans $\mathbb{R}^{j-1}$, on en déduit qu'aucun des $j-2$ premiers processeurs n'est inactif. Si P_{j-1} était inactif, on pourrait construire une solution utilisant les processeurs $P_1, \ldots, P_{j-2}, P_j$ et traitant strictement plus de tâches. Tous les processeurs $P_1, \ldots, P_{j-1}$ traitent donc des tâches. La projection de S_1 étant un sommet du polyèdre induit par notre nouveau programme linéaire, au moins $j-1$ contraintes sont des égalités et donc, les (UB(i)) sont des égalités pour $i < j$. On sait grâce au lemme 6.4 que (UB(j)) est une égalité, ce qui achève la démonstration. ■

Notons $S_2 = (\alpha_1, \ldots, \alpha_j)$ la solution optimale initiale et supposons qu'il existe en S_2 un (UB(i)) pour $i \leqslant j$ qui n'est pas une égalité et donc que S_1 est différent de S_2. Considérons la fonction f suivante :

$$f : \left\{ \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R}^j \\ x & \mapsto & S_1 + x(S_2 - S_1) \end{array} \right.$$

S_1 et S_2 étant deux solutions optimales distinctes, on a $\sum \beta_i = \sum \alpha_i$. En notant $f(x) = (\gamma_1(x), \ldots, \gamma_n(x))$, on peut remarquer que

$$\forall x, \sum_i \gamma_i(x) = \sum_i \alpha_i = \sum_i \beta_i.$$

Tous les points $f(x)$ appartenant à $\mathcal{P}$ sont donc des solutions optimales de ÉtoileAffine.

$\mathcal{P}$ étant un polyèdre convexe (au vu des contraintes), pour tout $x \in [0, 1]$, $f(x)$ appartient à $\mathcal{P}$ et est donc une solution optimale de ÉtoileAffine. Notons x_0 la plus grande

valeur des $x \geqslant 1$ tels que $f(x)$ appartienne à $\mathcal{P}$. $f(x_0)$ est alors une solution optimale de ÉtoileAffine, appartient à la frontière de $\mathcal{P}$ et donc au moins une des contraintes linéaires de (6.10) est une égalité.

Cette contrainte est nécessairement une des (LB(i)) (sinon S_1 et S_2 appartiendrait au même hyperplan, la droite (S_1, S_2) aussi, et donc $f(x)$ pour $x > x_0$ ne violerait pas cette contrainte) et, comme (par le lemme 6.4) les (LB(i)) sont strictes pour $i < j$, c'est nécessairement (LB(j)). Nous savons donc que la $j^{\text{ème}}$ composante de $f(x_0)$ est nulle et que les $j-1$ premières composantes $S'_2 = (\alpha'_1, \ldots, \alpha'_{j-1})$ sont strictement positives et sont une solution optimale du programme linéaire ou seuls les $j-1$ premiers processeurs sont utilisés.

En notant $S'_1 = (\beta'_1, \ldots, \beta'_{j-1})$ une solution optimale appartenant à un sommet du polygone, $(\beta'_1, \ldots, \beta'_{j-1}, 0)$ est une solution optimale du programme linéaire d'origine. Toutes les inégalités (LB(i)) sont donc strictes pour $i < j$ (lemme 6.4) et, comme précédemment, on peut montrer que les contraintes (UB(i)) en S'_1 sont des égalités pour $i < j$.

- Si $S'_1 = S'_2$, alors les (UB(i)) sont également des égalités en $f(x_0)$ pour $i < j$. Dans ce cas, ce sont des égalités en S_1 et en $f(x_0)$ et c'est également le cas en S_2, ce qui est absurde.
- Si $S'_1 \neq S'_2$, alors considérons la fonction g suivante :

$$g : \left\{ \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R}^{j-1} \\ x & \mapsto & S'_1 + x(S'_2 - S'_1) \end{array} \right.$$

  Par construction, tous les points $g(x)$ appartenant à $\mathcal{P}'$ sont des solution optimales du programme linéaire. Notons x'_0 la plus grande valeur des $x \geqslant 1$ tels que $g(x)$ appartienne à $\mathcal{P}'$. $g(x'_0)$ est alors une solution optimale de notre programme linéaire et donc au moins une des contraintes linéaires est une égalité. Cette contrainte est nécessairement une des (LB(i)) et on sait par le lemme 6.4 que ce n'est pas possible.

Les contraintes (UB(i)) en S_2 pour $i \leqslant j$ sont donc des égalités, ce qui signifie que dans toute solution optimale, tous les esclaves participant au traitement des tâches terminent leur exécution au même instant. ■

Théorème 6.2. *Si la quantité de travail à effectuer est suffisamment grande, tous les esclaves participent au traitement des tâches dans toute solution optimale et les communications se font dans l'ordre des g_i croissant.*

Démonstration. Considérons uns solution valide de ÉtoileAffine avec comme borne temporelle T. On peut supposer sans perte de généralité que $\alpha_{\sigma(1)}$ tâches sont d'abord envoyées à $P_{\sigma(1)}$, puis $\alpha_{\sigma(2)}$ tâches à $P_{\sigma(2)}$, ... et enfin $\alpha_{\sigma(k)}$ tâches à $P_{\sigma(k)}$. L'ensemble $\mathcal{S} = \{P_{\sigma(1)}, \ldots, P_{\sigma(k)}\}$ est donc l'ensemble des processeurs participant au travail et σ est une injection de $[\![1, k]\!]$ dans $[\![1, n]\!]$ représentant l'ordre dans lequel les communications sont effectuées. Notons n_{opt} le nombre optimal de tâches traitées avec cet ensemble de processeurs et cet ordre.

Considérons l'instance suivante de ÉtoileLinéaire avec les k processeurs $P'_{\sigma(1)}, \ldots, P'_{\sigma(k)}$ et où pour tout i, on a $G'_i = 0, W'_i = 0, g'_i = g_i, w'_i = w_i$ et $T' = T$. Comme toutes les

latences de calcul et de communications ont été supprimées, le nombre optimal de tâches $n_{\text{opt}}^{(1)}$ traitées dans le modèle linéaire est supérieur à n_{opt}. $n_{\text{opt}}^{(1)}$ est égal à $f(\mathcal{S},\sigma)T$ où f est la forme close donnée par Robertazzi dans [36].

Considérons l'instance suivante de ÉtoileLinéaire avec les k processeurs $P'_{\sigma(1)},\dots,P'_{\sigma(k)}$ et où pour tout i, on a $G'_i = 0, W'_i = 0, g'_i = g_i, w'_i = w_i$ et $T' = T - \sum_{i\in\mathcal{S}}(G_i + W_i)$. Comme cela revient à comptabiliser toutes les latences de calcul et de communications à chaque fois, le nombre optimal de tâches $n_{\text{opt}}^{(2)}$ traitées dans le modèle linéaire est inférieure à n_{opt}. Comme précédemment, $n_{\text{opt}}^{(2)}$ est égal à $f(\mathcal{S},\sigma)\left(T - \sum_{i\in\mathcal{S}}(G_i + W_i)\right)$.

On a donc

$$f(\mathcal{S},\sigma)\left(1 - \frac{\sum_{i\in\mathcal{S}}(G_i+W_i)}{T}\right) \leqslant \frac{n_{\text{opt}}}{T} \leqslant f(\mathcal{S},\sigma).$$

Quand T devient grand, le débit de la plate-forme est donc arbitrairement proche de $f(\mathcal{S},\sigma)$, c'est-à-dire du débit de la plate-forme où les latences sont ignorées.

Or, on a montré (théorème 6.1) que $f(\mathcal{S},\sigma)$ est maximum quand $\mathcal{S}$ est l'ensemble de tous les processeurs et quand σ vérifie «$g_j > g_i \Rightarrow \sigma(i) > \sigma(j)$». Donc, quand T est suffisamment grand, tous les processeurs doivent être utilisés et on doit les servir dans l'ordre des g_i croissants. On sait également grâce au lemme 6.3 que tous les processeurs doivent terminer au même instant. On peut donc donner une forme close à partir du moment où T est suffisemment grand. ■

6.5 Distributions en plusieurs tournées sur une étoile

Dans cette section, nous proposons un algorithme périodique asymptotiquement optimal pour la distribution de tâches divisibles, que les processeurs soient capables ou non de recouvrir leurs calculs avec leurs communications.

6.5.1 Sans recouvrement

Le schéma général de l'algorithme est le suivant : le temps de calcul total T est divisé en k périodes identiques de durée T_p dont nous déterminerons les valeurs par la suite. Pour simplifier les notations, nous ignorons les latences de calcul W_i mais il suffit de remplacer partout G_i par $G_i + W_i$ pour obtenir le résultat en toute généralité.

Notons α_i la quantité de données envoyées par le maître à l'esclave P_i durant une période de durée T_p (voir figure 6.3). Il est parfaitement possible que tous les processeurs ne travaillent pas. Notons $\mathcal{I} \subset [\![1,p]\!]$ l'ensemble des indices des processeurs participant au calcul. L'inégalité suivante doit alors être respectée pour s'assurer que les communications avec le maître sont bien exclusives :

$$\sum_{i\in\mathcal{I}} (G_i + \alpha_i g_i) \leqslant T_p. \tag{6.11}$$

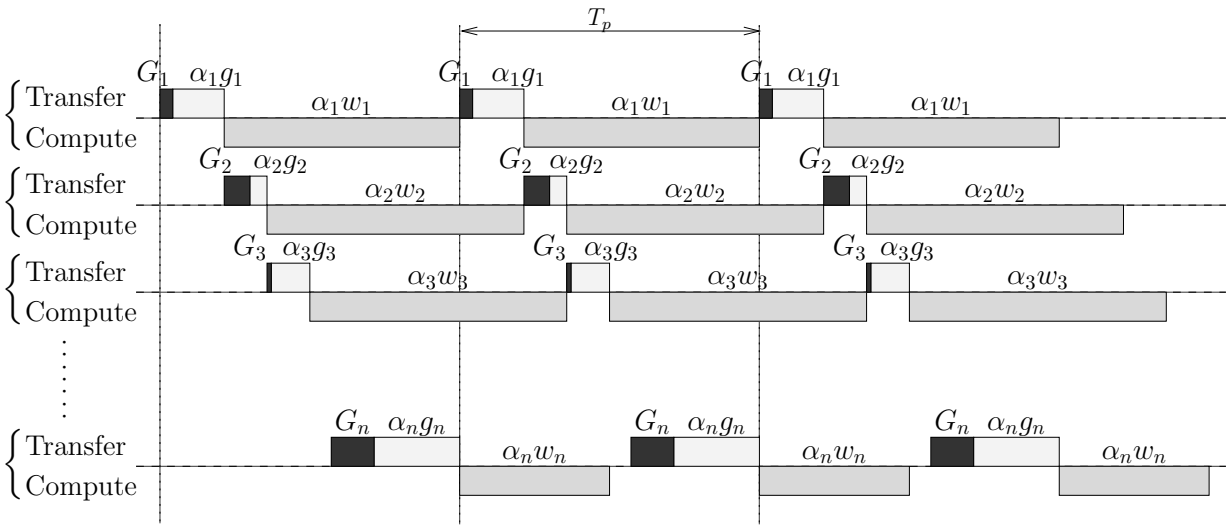


FIG. 6.3 – Schéma de la distribution périodique en plusieurs tournées. Ce schéma n'utilise que les n premiers esclaves $P_1 \dots P_n$.

Les processeurs ne pouvant pas recouvrir leurs calculs par des communications, on a également les inégalités suivantes :

$$\forall i \in \mathcal{I}, \quad G_i + \alpha_i(g_i + w_i) \leqslant T_p.$$

En exprimant les contraintes en fonction de $\frac{\alpha_i}{T_p}$, le nombre moyen de tâches traitées par P_i en une unité de temps, le système de contraintes devient

$$\begin{cases} \forall i \in \mathcal{I}, \quad \dfrac{\alpha_i}{T_p}(g_i + w_i) \leqslant 1 - \dfrac{G_i}{T_p} & \text{(pas de recouvrement)} \\ \displaystyle\sum_{i \in \mathcal{I}} \frac{\alpha_i}{T_p} g_i \leqslant 1 - \frac{\sum_{i \in \mathcal{I}} G_i}{T_p} & \text{(communications exclusives avec le maître)} \end{cases}, \tag{6.12}$$

et nous cherchons à maximiser le nombre total de tâches traitées par unité de temps, c'est-à-dire $\sum_{i \in \mathcal{I}} \frac{\alpha_i}{T_p}$.

Il est cependant impossible de résoudre ce programme linaire tant que l'on ne connaît pas $\mathcal{I}$. C'est pourquoi nous proposons d'ajouter quelques contraintes avec la formulation suivante :

$$\begin{array}{l} \text{MAXIMISER } \sum_{i=1}^{p} \frac{\alpha_i}{T_p}, \\ \text{SOUS LES CONTRAINTES} \\ \begin{cases} \text{(6.13a)} \quad \forall 1 \leqslant i \leqslant p, \quad \dfrac{\alpha_i}{T_p}(g_i + w_i) \leqslant 1 - \dfrac{\sum_{i=1}^{p} G_i}{T_p} \\ \text{(6.13b)} \quad \displaystyle\sum_{i=1}^{p} \frac{\alpha_i}{T_p} g_i \leqslant 1 - \frac{\sum_{i=1}^{p} G_i}{T_p} \end{cases} \end{array} \tag{6.13}$$

Les inéquations étant plus fortes que les précédentes, toute solution de (6.13) vérifie également les inégalités initiales. Ces solutions n'ont en revanche aucun raison d'être

optimale pour le problème initial. Cependant, plus T_p est grand, plus les contraintes de (6.13) se rapprochent de celles de (6.12) et donc plus la solution de (6.13) devient proche de celle de (6.12). Du coup, quand T_p devient grand, cette approximation est légitime et a peu d'influence. Nous montrons par la suite que malgré cette simplification, on arrive à en déduire une allocation très efficace.

La substitution de $1 - \frac{G_i}{T_p}$ par $1 - \frac{\sum_{i=1}^{p} G_i}{T_p}$ dans la première ligne n'est pas nécessaire. Le système d'inéquations aurait été suffisant sans cette substitution puisque c'est alors un simple problème de sac-à-dos fractionaire. Cependant, cette substitution nous permet d'aboutir à un programme linéaire homothétique à celui rencontré en section 4.4. Il suffit alors de trier les processeurs par g_i croissant et de définir q comme le plus grand indice tel que $\sum_{i=1}^{q} \frac{g_i}{g_i+w_i} \leqslant 1$. On définit la quantité ε par $1 - \sum_{i=1}^{q} \frac{g_i}{g_i+w_i}$ si $q < p$ et par $\varepsilon = 0$ sinon. Ce dernier cas correspond au cas où l'utilisation de tous les processeurs ne sature pas les capacités de communications du maître. La solution de ce programme linéaire est alors définie par :

$$\forall 1 \leqslant i \leqslant q, \quad \frac{\alpha_i}{T_p} = \frac{1 - \frac{\sum_{i=1}^{p} G_i}{T_p}}{g_i + w_i}$$

et (si $q < p$) :

$$\frac{\alpha_{q+1}}{T_p} = \left(1 - \frac{\sum_{i=1}^{p} G_i}{T_p}\right)\left(\frac{\varepsilon}{g_{q+1}}\right),$$

et $\alpha_{q+2} = \alpha_{q+3} = \ldots = \alpha_p = 0$.

Théorème 6.3. *Soit $((G_i)_{1\leqslant i\leqslant p}, (g_i)_{1\leqslant i\leqslant p}, (w_i)_{1\leqslant i\leqslant p})$ une plate-forme. Si on note*

$$\rho_{opt} = \left(\sum_{i=1}^{q} \frac{1}{g_i + w_i} + \frac{\varepsilon}{g_{p+1}}\right),$$

la distribution périodique précédente de B tâches avec $T_p = \sqrt{\frac{B}{\rho_{opt}}}$ est asymptotiquement optimale.

Démonstration. Le rendement par unité de temps de cette distribution est donc égal à

$$\rho = \sum_{i=1}^{p} \frac{\alpha_i}{T_p} = \left(1 - \frac{\sum_{i=1}^{p} G_i}{T_p}\right) \rho_{\text{opt}}.$$

Considérons une distribution optimale des B tâches et notons $\rho^\star$ le nombre de tâches traitées par unité de temps. Notons $\beta_i^\star$ le nombre de tâches traitées par le processeur P_i en une unité de temps dans cette solution. Les $\beta_i^\star$ satisfont donc les inégalités suivantes (dont les G_i ont été retirées).

$$\begin{cases} \forall 1 \leqslant i \leqslant p, \quad \beta_i^\star (g_i + w_i) \leqslant 1 \\ \sum_{i=1}^{p} \beta_i^\star g_i \leqslant 1 \end{cases}$$

Dans les équations précédentes, comme il n'y a pas de latence, nous pouvons sans perte de généralité supposer que tous les esclaves travaillent et poser $\beta_i^\star = 0$ pour ceux qui ne traitent aucune tâches. On en déduit que :

$$\rho^\star \leqslant \rho_{\text{opt}}.$$

Si on note $T^\star$ le temps optimal nécessaire au traitement des B tâches, on a donc :

$$T^\star \geqslant \frac{B}{\rho^\star} \geqslant \frac{B}{\rho_{\text{opt}}}.$$

Notons T le temps nécessaire au traitement des B tâches avec notre algorithme. Le nombre de périodes nécessaires pour traiter les B tâches vérifie donc $\rho(k-1)T_p \geqslant B$ et donc on a :

$$k = \left\lceil \frac{B}{\rho T_p} \right\rceil + 1.$$

Ainsi on a

$$T \leqslant \frac{B}{\rho} + 2T_p \leqslant \frac{B}{\rho_{\text{opt}}} \left(\frac{1}{1 - \sum_{i=1}^{p} \frac{G_i}{T_p}} \right) + 2T_p,$$

et donc, quand $T_p \geqslant 2\sum_{i=1}^{p} G_i$,

$$T \leqslant \frac{B}{\rho_{\text{opt}}} + 2\frac{B}{\rho_{\text{opt}}} \sum_{i=1}^{p} \frac{G_i}{T_p} + 2T_p.$$

Enfin, si on pose $T_p = \sqrt{\frac{B}{\rho_{\text{opt}}}}$, on peut vérifier que

$$T \leqslant \frac{B}{\rho_{\text{opt}}} + 2\sum_{i=1}^{p} G_i T_p + 2T_p \leqslant T^\star + 2\left(\sum_{i=1}^{p} G_i + 1 \right) \sqrt{T^\star},$$

et donc que

$$\frac{T}{T^\star} \leqslant 1 + 2\left(\sum_{i=1}^{p} G_i + 1 \right) \frac{1}{\sqrt{T^\star}} = 1 + O\left(\frac{1}{\sqrt{T^\star}} \right) = 1 + O\left(\frac{1}{\sqrt{B}} \right),$$

ce qui achève la démonstration de l'optimalité asymptotique de notre distribution. ■

On remarquera que la sélection des ressources s'effectue par notre résolution explicite du programme linéaire. Intuitivement, il suffit de sélectionner les processeurs à la bande passante la plus rapide tant que la somme des rapports entre leur temps de communication avec leur temps de calcul *plus* leur temps de communication est inférieur à 1. On notera également qu'il est très simple de prendre en compte la latence de calcul W_i introduite par Casanova et Yang [118] puisqu'il suffit de remplacer G_i par $G_i + W_i$.

6.5.2 Avec recouvrement

Dans le cas où les esclaves sont capables de recouvrir leurs calculs et leurs communications, il suffit de modifier quelque peu les équations. Durant la période $i+1$, les esclaves traitent les tâches qu'ils ont reçues à la période i, si bien qu'aucun calcul n'est effectué durant la première tournée et qu'aucune communication n'a lieu durant la dernière tournée. Ces modifications ne remettent pas en question la démonstration de l'optimalité asymptotique de la méthode. Le système d'inéquations sur le nombre de tâches traitées par unité de temps devient simplement :

$$\begin{cases} \forall i \in \mathcal{I}, \dfrac{\alpha_i}{T_p} w_i \leqslant 1 & \text{(recouvrement)} \\ \displaystyle\sum_{i \in \mathcal{I}} \frac{\alpha_i}{T_p} g_i \leqslant 1 - \frac{\sum_{i=1}^{p} G_i}{T_p} & \text{(communications exclusives avec le maître)} \end{cases}$$

On peut alors montrer que, comme précédemment, ρ vérifie :

$$\rho \geqslant \left(1 - \frac{\sum_{i=1}^{p} G_i}{T_p}\right) \left(\sum_{i=1}^{q} \frac{1}{w_i} + \frac{\varepsilon}{g_{q+1}}\right),$$

où q est le plus grand indice tel que $\sum_{i=1}^{q} \frac{g_i}{w_i} \leqslant 1$ et où ε vaut $1 - \sum_{i=1}^{q} \frac{g_i}{w_i}$ si $q < p$ et 0 sinon. De la même façon, on montre que pour la distribution optimale de B tâches on a :

$$\rho^\star \leqslant \left(\sum_{i=1}^{q} \frac{1}{w_i} + \frac{\varepsilon}{g_{q+1}}\right).$$

On montre ensuite que

$$T \leqslant T^\star + 2(\sum_{i=1}^{p} G_i + 1)\sqrt{T^\star},$$

et donc que

$$\frac{T}{T^\star} \leqslant 1 + 2(\sum_{i=1}^{p} G_i + 1)\frac{1}{\sqrt{T^\star}} = 1 + O\left(\frac{1}{\sqrt{B}}\right),$$

ce qui montre l'optimalité asymptotique de cette distribution. Comme précédemment, il est aisé de prendre en compte les W_i en remplaçant partout G_i par $G_i + W_i$.

6.6 Distributions en plusieurs tournées sur une plate-forme quelconque

Dans cette section, nous proposons une distribution en plusieurs tournées pour les plates-formes quelconques. Dans la section 6.6.1, nous présentons le modèle de plate-forme et les notations que nous utilisons. En section 6.6.2, nous donnons une borne supérieure de

la puissance de la plate-forme. En section 6.6.3, nous expliquons comment construire un motif efficace et nous l'utilisons en section 6.6.4 pour proposer une distribution dont nous démontrons l'optimalité asymptotique. Enfin, en section 6.6.5, nous expliquons comment prendre en compte d'autres modèles de communications.

6.6.1 Modélisation de la plate-forme

La plate-forme est représentée par un graphe $G = (V, E, w_i, G_{i,j}, g_{i,j})$. Les nœuds du graphe représentent les ressources de calcul et sont pondérés par les w_i, c'est-à-dire le temps nécessaire au traitement d'une tâche élémentaire. Les arêtes représentent le réseau d'interconnexion et sont pondérées par une latence $G_{i,j}$ et l'inverse d'une bande-passante $g_{i,j}$, si bien que le temps nécessaire au transfert de α tâches élémentaires le long de l'arête $e_{i,j} = (P_i, P_j)$ est égal à $G_{i,j} + \alpha g_{i,j}$. Les hypothèses concernant les différents modèles de communication que l'on peut utiliser seront discutées en section 6.6.5. En attendant, nous utilisons un modèle 1-port en entrée et 1-port en sortie, ainsi qu'un recouvrement complet des calculs et des communications.

6.6.2 Borne supérieure de la puissance de la plate-forme

Notre objectif est de trouver une borne supérieure du nombre de tâches pouvant être traitées en temps T par la plate-forme. Pour ce faire, nous considérons une plate-forme *idéale* dont on a supprimé les latences. Une telle plate-forme a clairement une puissance supérieure à la plate-forme originale. Considérons un nœud P_i en régime permanent et notons α_i^{ideal} la quantité de tâches qu'il traite par unité de temps. Notons $c_{j,i}^{\text{ideal}}$ la quantité de tâches qu'il reçoit de ses voisins et $c_{i,j}^{\text{ideal}}$ la quantité de tâches qu'il envoie à ses voisins. L'ensemble des voisins auquel P_i peut envoyer des tâches est noté $P_i \to P_j$ et on note $P_j \to P_i$ l'ensemble des processeurs envoyant des données à P_i. Les hypothèses que nous avons fait sur le modèle de communications induisent l'ensemble de contraintes suivant :

$$\begin{cases} \text{(6.14a)} & \alpha_i^{\text{ideal}} w_i \leqslant 1 \\ \text{(6.14b)} & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} g_{j,i} \leqslant 1 \qquad \text{(plus de latence)} \\ \text{(6.14c)} & \displaystyle\sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} g_{i,j} \leqslant 1 \qquad \text{(plus de latence)} \\ \text{(6.14d)} & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} = \alpha_i^{\text{ideal}} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} \end{cases} \tag{6.14}$$

La contrainte (6.14a) porte sur la capacité de calcul, la contrainte (6.14b) sur le modèle 1-port en entrée, la contrainte (6.14c) sur le modèle 1-port en sortie et l'équation (6.14d) est une loi de conservation. Ainsi, en régime permanent, le nombre de tâches optimal que

peut traiter la plate-forme est donné par la solution du programme linéaire suivant :

$$\begin{array}{l}\text{MAXIMISER } \sum_{i=1}^{p} \alpha_i^{\text{ideal}}, \\ \text{SOUS LES CONTRAINTES} \\ \begin{cases} \forall P_i \in V, \alpha_i^{\text{ideal}} w_i \leqslant 1 \\ \forall P_i \in V, \sum\limits_{P_j \to P_i} c_{j,i}^{\text{ideal}} g_{j,i} \leqslant 1 \\ \forall P_i \in V, \sum\limits_{P_i \to P_j} c_{i,j}^{\text{ideal}} g_{i,j} \leqslant 1 \\ \forall P_i \in V, \sum\limits_{P_j \to P_i} c_{j,i}^{\text{ideal}} = \alpha_i^{\text{ideal}} + \sum\limits_{P_i \to P_j} c_{i,j}^{\text{ideal}} \\ \forall P_i \in V, \alpha_i^{\text{ideal}}, c_{i,j}^{\text{ideal}}, c_{j,i}^{\text{ideal}} \geqslant 0 \end{cases} \end{array} \tag{6.15}$$

Notons $\rho_{\text{opt}}^{\text{ideal}} = \sum_{P_i \in V} \alpha_i^{\text{ideal}}$ le nombre optimal de tâches traitées par la plate-forme *idéale* en une unité de temps. Le temps nécessaire au traitement de N tâches sur cette plate-forme idéale est donc clairement supérieur à $T_{\text{sup}} = \frac{N}{\rho_{\text{opt}}^{\text{ideal}}}$. C'est donc également le cas pour le temps nécessaire au traitement de N tâches sur la plate-forme réelle.

Dans les sections suivantes, nous proposons une distribution en plusieurs tournées périodique qui traite N tâches en temps $\frac{N}{\rho_{\text{opt}}^{\text{ideal}}} + O\left(\sqrt{\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}}\right)$ et qui est donc asymptotiquement optimale.

6.6.3 Un motif efficace

Dans cette section, nous définissons un motif de durée T_p pour la plate-forme initiale à partir de la solution du programme linéaire (6.15). De la même façon qu'en section 4.3.5, la solution du programme linéaire (6.15) induit un motif définit par une décomposition en couplage du graphe de communications $\sum_c \alpha_c \chi_c$ telle que $\sum_c \alpha_c \leqslant 1$ (voir figure 6.4(a)).

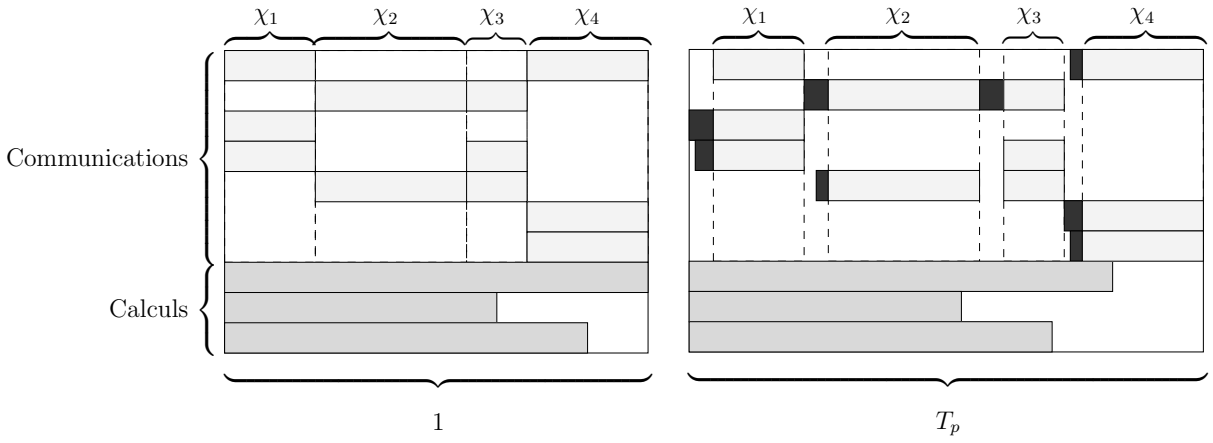
Alors, en multipliant chacune des quantités définissant le motif associé par T_p, on ne pourra pas obtenir directement un motif valide de durée T_p à cause des latences. En revanche, en les multipliant par

$$T_p - \sum_c \max_{i,j \in \chi_c} G_{i,j} = T_p - A_1,$$

on obtient un motif valide (c'est-à-dire respectant les contraintes de ressources), de durée inférieure à T_p, et permettant de traiter $\rho_{\text{opt}}^{\text{ideal}}(T_p - A_1)$ tâches (voir figure 6.4(b)).

6.6.4 Distribution asymptotiquement optimale

L'algorithme que nous proposons comporte 3 phases : une phase d'initialisation permettant d'atteindre le régime permanent, un certain nombre de tournées de durée T_p en utilisant le motif utilisé dans la section précédente, et une phase de nettoyage.



(a) Motif linéaire de durée 1 (b) Motif affine de durée T_p

FIG. 6.4 – Passage d'un motif linéaire de durée 1 à un motif affine de durée T_p

6.6.4.1 Phase d'initialisation

Dans cette phase, aucune tâche n'est traitée par les processeurs. Ils se contentent de recevoir le nombre de tâches qu'ils reçoivent normalement pendant une période, de façon à pouvoir se mettre en régime permanent à l'étape suivante. Le processeur P_i doit donc recevoir :

$$\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}}(T_p - A_1) \leqslant \frac{T_p - A_1}{\min_{P_j \to P_i} g_{j,i}} \leqslant \frac{T_p - A_1}{\min_{P_j \to P_i} g_{j,i}} \leqslant \frac{1}{\min_{P_j \to P_i} g_{j,i}} T_p \leqslant \frac{1}{\min_{(P_k,P_l)\in E} g_{k,l}} T_p.$$

Rappelons que le nombre de processeurs étant égal à n, n est une borne supérieure de la longueur du chemin du maître à P_i et le temps nécessaire à l'envoi de leurs tâches au processeur P_i est donc borné par

$$n \left(\max_{(P_k,P_l)\in E} G_{k,l} + \frac{\max_{(P_k,P_l)\in E} g_{k,l}}{\min_{(P_k,P_l)\in E} g_{k,l}} T_p \right).$$

Le temps nécessaire à l'envoi de leurs tâches à l'ensemble des esclaves est borné par :

$$n^2 \left(\max_{(P_k,P_l)\in E} G_{k,l} + \frac{\max_{(P_k,P_l)\in E} g_{k,l}}{\min_{(P_k,P_l)\in E} g_{k,l}} T_p \right).$$

Le temps d'initialisation est donc borné par

$$A_2 + T_p A_3,$$

où $A_2 = n^2 \max G_{k,l}$ et $A_3 = n^2 \frac{\max g_{k,l}}{\min g_{k,l}}$ ne dépendent que des caractéristiques de la plate-forme mais pas du nombre de tâches à traiter ni de la période T_p.

6.6.4.2 Régime permanent

Les tâches étant présentes à chaque nœud de la plate-forme à la fin de la phase d'initialisation, on peut se mettre en régime permanent. À l'étape i, chaque processeur traite donc les tâches qu'il a reçues à l'étape $i-1$. Après k étapes, le nombre de tâches traitées est donc égal à $k\rho_{\text{opt}}^{\text{ideal}}(T_p - A_1)$. Le nombre d'étapes de régime permanent est donc égal à

$$k = \left\lfloor \frac{N}{(T_p - A_1)\rho_{\text{opt}}^{\text{ideal}}} \right\rfloor$$

La durée de la phase de régime permanent est donc bornée par

$$\frac{NT_p}{(T_p - A_1)\rho_{\text{opt}}^{\text{ideal}}}$$

où A_1 est défini en section 6.6.3 et ne dépend que des caractéristiques de la plate-forme et pas du nombre de tâches à traiter ni de la période T_p.

6.6.4.3 Phase de nettoyage

À la fin de la $k^{\text{ème}}$ étape du régime permanent, au plus $T_p\rho_{\text{opt}}^{\text{ideal}}$ tâches n'ont pas encore été traitées. Le temps nécessaire à leur traitement en séquentiel sur la machine la plus lente est borné par $T_p\rho_{\text{opt}}^{\text{ideal}} \max_i w_i$ et la durée de la phase de nettoyage est donc bornée par

$$A_4 T_p,$$

où $A_4 = \rho_{\text{opt}}^{\text{ideal}} \max_i w_i$ dépend uniquement de la plate-forme.

6.6.4.4 Optimalité asymptotique

Une distribution de N tâches telle que celle que nous venons de décrire est donc de durée $T^{\text{périodique}}$ bornée par

$$T^{\text{périodique}} \leqslant A_2 + (A_3 + A_4)T_p + \frac{NT_p}{\rho_{\text{opt}}^{\text{ideal}}(T_p - A_1)}.$$

L'expression de droite est minimisée lorsque

$$T_p = A_1 + \sqrt{\frac{N.A_1}{\rho_{\text{opt}}^{\text{ideal}}(A_3 + A_4)}} = A_1 + \alpha\sqrt{\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}}.$$

On a alors

$$\begin{aligned} T^{\text{périodique}} &\leqslant A_2 + (A_3 + A_4)\left(A_1 + \alpha\sqrt{\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}}\right) + \frac{N}{\rho_{\text{opt}}^{\text{ideal}}}\left(\frac{A_1 + \alpha\sqrt{\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}}}{\alpha\sqrt{\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}}}\right) \\ &\leqslant (A_2 + A_1A_3 + A_1A_4) + \left(\alpha(A_3 + A_4) + \frac{A_1}{\alpha}\right)\sqrt{\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}} + \left(\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}\right) \\ T^{\text{périodique}} &= \frac{N}{\rho_{\text{opt}}^{\text{ideal}}} + O\left(\sqrt{\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}}\right). \end{aligned}$$

Comme la durée optimale d'une distribution est supérieure à $\frac{N}{\rho_{\text{opt}}^{\text{ideal}}}$, cette distribution périodique de période $T_p = A_1 + \sqrt{\frac{N.A_1}{\rho_{\text{opt}}^{\text{ideal}}(A_3+A_4)}}$ est asymptotiquement optimale.

6.6.5 Extension du modèle

Dans cette section, nous expliquons comment étendre le résultat précédent quand les hypothèses sur la plate-forme sont différentes. En effet, le principe précédent reste valide pour obtenir une distribution asymptotiquement optimale quand les hypothèses sur les capacités des processeurs à recouvrir leurs communications par du calcul sont différentes. Nous ne refaisons cependant pas complètement la démonstration car elle est quasiment identique à celle de la section 6.6.4 mais il suffit juste de modifier un peu les équations de la section 6.6.2 pour la plate-forme idéale. Rappelons donc les équations utilisées en section 6.6.2.

$$\begin{cases} (6.14a) & \alpha_i^{\text{ideal}} w_i \leqslant 1 \\ (6.14b) & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} g_{j,i} \leqslant 1 \\ (6.14c) & \displaystyle\sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} g_{i,j} \leqslant 1 \\ (6.14d) & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} = \alpha_i^{\text{ideal}} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} \end{cases}$$

6.6.5.1 Nombre de ports de communication

Dans cette section, on suppose toujours que les processeurs sont capables de faire du recouvrement, mais on effectue différentes hypothèses sur le nombre de ports en entrée et en sortie.

– Si le nœud P_i n'a qu'un seul port pour les communications entrantes et les commu-

nications sortantes, les contraintes deviennent

$$\begin{cases} (6.16a) & \alpha_i^{\text{ideal}} w_i \leqslant 1 \\ (6.16b) & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} g_{j,i} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} g_{i,j} \leqslant 1 \\ (6.16d) & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} = \alpha_i^{\text{ideal}} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} \end{cases} \tag{6.16}$$

– Si le nœud P_i est capable de traiter autant de communications en parallèle que nécessaire, alors les contraintes deviennent

$$\begin{cases} (6.17a) & \alpha_i^{\text{ideal}} w_i \leqslant 1 \\ (6.17b) & \forall P_j \text{ t.q. } P_j \to P_i, c_{j,i}^{\text{ideal}} g_{j,i} \leqslant 1 \\ (6.17c) & \forall P_j \text{ t.q. } P_i \to P_j, c_{i,j}^{\text{ideal}} g_{i,j} \leqslant 1 \\ (6.17d) & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} = \alpha_i^{\text{ideal}} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} \end{cases} \tag{6.17}$$

Ces équations nous permettent de trouver une borne supérieure de l'efficacité d'une allocation mais on retrouve cependant les mêmes problèmes qu'en section 4.5.2 pour expliciter une allocation atteignant cette borne. Tant que le nombre de ports est supérieur à 2 (un en entrée et un en sortie), il est possible d'utiliser la décomposition en couplages de la section 4.3.5 car le graphe de communications est biparti. Mais dans le cas contraire, ce n'est plus toujours possible et l'existence d'allocations atteignant le régime permanent ainsi calculé reste une question ouverte.

6.6.5.2 Recouvrement des communications par du calcul

Nous nous intéressons maintenant aux cas où les processeurs ne sont pas capables de recouvrir leurs communications par du calcul.

– Si le nœud P_i n'a qu'un seul port pour les communications entrantes et les communications sortantes, alors il peut soit envoyer, soit recevoir, soit traiter des tâches et les contraintes deviennent

$$\begin{cases} (6.18a) & \alpha_i^{\text{ideal}} w_i + \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} g_{j,i} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} g_{i,j} \leqslant 1 \\ (6.18d) & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} = \alpha_i^{\text{ideal}} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} \end{cases} \tag{6.18}$$

– Dans le cas où le nœud P_i est capable de traiter autant de communications en parallèle que nécessaire, alors toutes les communications interfèrent avec le calcul et les contraintes deviennent

$$\begin{cases} (6.19a) & \forall P_j \text{ t.q. } P_j \to P_i, \alpha_i^{\text{ideal}} w_i + c_{j,i}^{\text{ideal}} g_{j,i} \leqslant 1 \\ (6.19b) & \forall P_j \text{ t.q. } P_i \to P_j, \alpha_i^{\text{ideal}} w_i + c_{i,j}^{\text{ideal}} g_{i,j} \leqslant 1 \\ (6.19d) & \displaystyle\sum_{P_j \to P_i} c_{j,i}^{\text{ideal}} = \alpha_i^{\text{ideal}} + \sum_{P_i \to P_j} c_{i,j}^{\text{ideal}} \end{cases} \tag{6.19}$$

En fait, l'uniformité des capacités de nœuds de calcul n'est absolument pas nécessaire. Les solutions étant obtenues à partir des équations du régime permanent pour chaque nœud, on peut utiliser des inégalités différentes.

6.7 Simulations

Afin d'évaluer les performances de notre algorithme, nous avons mis en place une simulation à l'aide du simulateur SimGrid décrit dans le chapitre 7. Un des principaux avantages de ce simulateur est que les caractéristiques des processeurs et du réseau correspondent à des valeurs réalistes et que la mise en place de la simulation est relativement proche d'une mise en oeuvre réelle. Nous détaillons d'abord en section 6.7.1 les différents paramètres de la simulation tels que la modélisation de la plate-forme puis nous présentons les différentes heuristiques nous avons mises en œuvre et comparées. Enfin nous présentons dans les autres sections les résultats de ces études sur différents types de plates-formes.

6.7.1 Modélisation

Dans nos expériences, la quantité totale de travail à affecter w_{total} varie entre 100 et 2000 par incrément de 100. Nous utilisons le modèle des tâches divisibles et nous permettons donc d'assigner des quantités rationnelles de tâches à des processeurs. La quantité de travail nécessaire à une tâche (c'est à dire le nombre d'opérations par unité de tâches) varie d'une expérience à l'autre de façon à pouvoir faire varier le rapport calculs/communications.

Dans les expériences, aucun recouvrement calcul-communication n'est effectué. Nous avons comparé notre algorithme en plusieurs tournées sans recouvrement avec l'algorithme *multi-installment* proposé dans [18]. Nous avons donc comparé un total de onze heuristiques. Les trois premières sont des variations basées sur la formulation en programmation linéaire. L'algorithme *multi-installment* a été testé pour un nombre de tournées variant de 1 à 8.

Programmation linéaire avec une période fixe. Nous utilisons le programme linéaire de la section 6.5.1 en fixant arbitrairement la durée de la période T_p à 2000. Tant qu'il reste des tâches à traiter, nous les allouons en utilisant une variante de l'équation 6.11 : nous maximisons $\sum_{i=1}^{p} \alpha_i$ sous les contraintes que $\sum_{i=1}^{p}(G_i + \alpha_i g_i) \leqslant T_p$ et que $G_i + \alpha_i(g_i + w_i) \leqslant T_p$ pour tout $i \in [\![1, p]\!]$. Le problème est donc légèrement trop contraint puisque nous incluons les latences de tous les processeurs même si certains ne participent pas. Évidemment, si certains α_i sont nuls, aucune communication n'est effectuée. Cette approximation est bonne si T_p est grand et fourni une méthode simple pour réaliser l'allocation quand la période T_p n'est pas connue.

Programmation linéaire avec une approximation fixe de la période. C'est la distribution présentée en section 6.5.1 et dont on a démontré l'optimalité asymptotique dans le théorème 6.3.

Programmation linéaire avec une période adaptative. C'est une variante de l'heuristique précédente. À chaque étape, la période T_p est recalculée en tant que $T_p = \sqrt{\frac{B'}{\rho_{\text{opt}}}}$, où B' est le nombre de tâches restantes. Dans les dernières étapes, nous arrêtons la décroissance de T_p, de façon à ce qu'au moins une tâche soit traitée.

Multi-Installment.x. C'est la procédure proposée dans [18] avec x tournées. Un ensemble d'équations linéaires dont le nombre de variables dépend de x permet de calculer les quantités de données à distribuer à chaque esclave et à chaque tournée. Notons que ces équations ne prennent pas en compte les latences. Dans nos simulations le nombre de tournées x varie de 1 à 8.

6.7.2 Plate-Forme homogène sans latence

La première série d'expériences concerne des plate-forme homogènes constituées de processeurs PIII 1GHz (capables d'effectuer 114,444 Mflop/s), interconnectés par un réseau Ethernet 100 Mbit/sec, mais dont la bande passante effective est de 32,10 Mbit/sec. Cette bande passante (ainsi que les autres données dans cette section) peuvent paraître faibles, mais elles correspondent à des bandes passantes mesurées en journée, sur un réseau un peu chargé en utilisant ENV [100]. De plus ces tests sont effectués en utilisant ssh (afin de permettre des rebonds pour rentrer dans des réseaux privés et passer à travers les *firewalls*) et pas directement des sockets, ce qui explique les faibles performances. Néanmoins, ces valeurs sont représentatives de ce que l'on peut effectivement obtenir dans le cadre d'une mise en œuvre réelle.

Le nombre de processeurs varie de 1 à 20 et une unité de travail équivaut à 1 Gflop de calcul et 2 Mbit de données à échanger. En d'autre termes, le temps de communication d'une tâche vers un esclave est de 0,06 secondes et le temps de traitement est de 9 secondes. Le rapport entre les calculs et les communications est donc relativement élevé, ce qui est un cas assez favorable.

La figure 6.5 représente les performances des 11 heuristiques pour une plate-forme à 5 processeurs. Le comportement général est identique pour les plates-formes de tailles différentes. On peut remarquer que l'heuristique à base de programmation linéaire avec une période fixe a un comportement étrange par rapport aux autres. Ce phénomène de paliers est dû au fait que dans cette heuristique, on cherche juste à maximiser le nombre de tâches effectuées en un temps T fixe. Quand T est trop grand, on calcule une allocation qui utilise quand même tout ce temps. Toutes les autres stratégies ayant un comportement similaire, nous représentons le rapport des performances des heuristiques restantes avec celle de l'heuristique à base de programmation linéaire avec une période adaptative sur la figure 6.6. Pour plus de clarté, nous ne représentons pas M.I.1, M.I.6, M.I.7 et M.I.8 car ils ne sont pas plus performants que les précédents. On peut faire les remarques suivantes :

– la stratégie adaptative est toujours très proche de la meilleure heuristique,

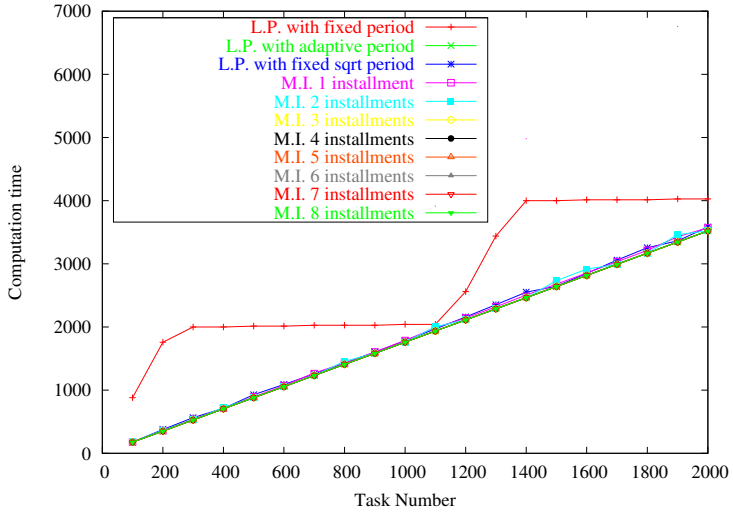


FIG. 6.5 – Temps simulé pour une plate-forme homogène à 5 processeurs.

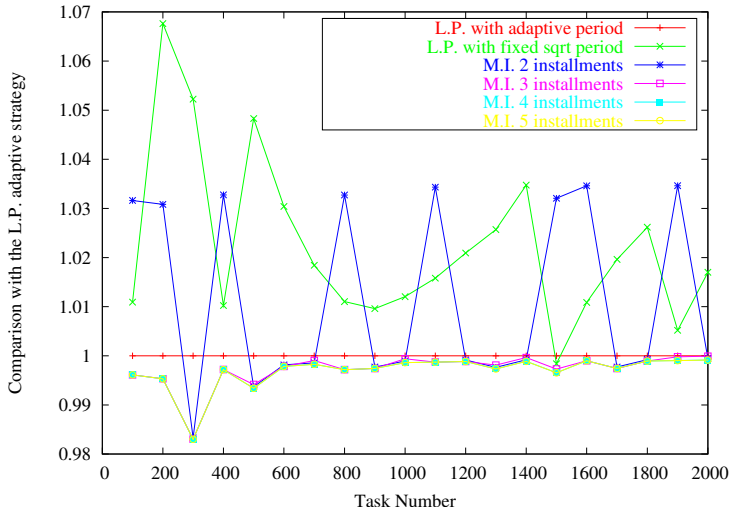


FIG. 6.6 – Comparaison avec les performances de l'heuristique adaptative sur une plate-forme homogène à 5 processeurs.

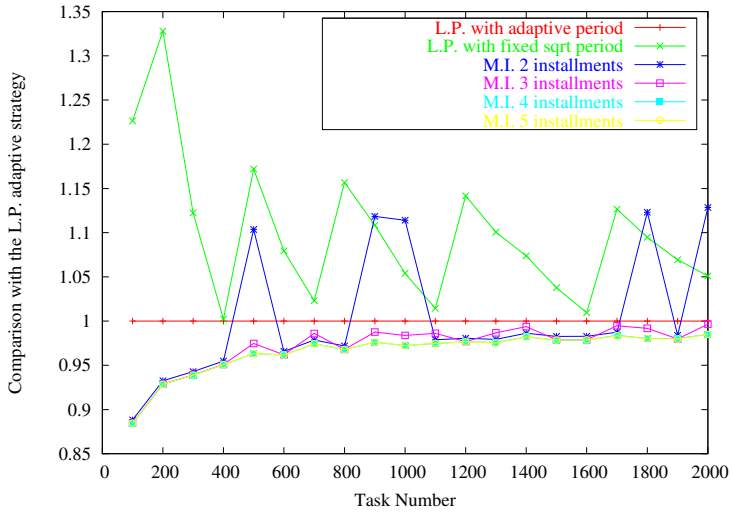


FIG. 6.7 – Comparaison avec l'heuristique adaptative pour une plate-forme homogène à 20 processeurs.

- le nombre de tournées optimal pour M.I.x est supérieur ou égal à 3 et l'augmentation du nombre de tournées au delà de cette valeur n'apporte pas vraiment d'améliorations significatives,
- les performances de l'heuristique que nous avons garantie (à base de programmation linéaire avec une approximation fixe) ne sont pas très régulières mais restent aux alentours des 5% de la meilleure heuristique à partir du moment où le nombre de tâches devient suffisamment important.

Ce type de comportement est également observé lorsque la taille de la plate-forme augmente. La stratégie adaptative requière cependant une quantité de tâches un peu plus importante pour obtenir le même type d'efficacité. Le nombre de tournées nécessaire pour la stratégie *Multi-Installment* de Robertazzi augmente également. À titre d'exemple, la figure 6.7 compare les performances des différentes heuristiques pour une plate-forme homogène à 20 processeurs.

6.7.3 Plate-Forme hétérogène sans latence

Les mesures présentées dans cette section ont été établies à l'aide de 2000 plates-formes simulées constituées de processeurs aléatoirement choisis dans l'ensemble de processeurs suivants : PPro 200MHz (22,151 Mflop/s), PII 450MHz (48,492 Mflop/s), PII 350MHz (34,333 Mflop/s) et PIII 1GHz (114,444 Mflop/s). Les réseaux utilisés pour interconnecter les esclaves au maître étaient soit de l'Ethernet 10, soit de l'Ethernet 100 (les bandes passantes effectives étant de 4,70 Mbit/sec pour l'Ethernet 10 et de 32,10 ou 30,25 Mbit/sec pour l'Ethernet 100). La quantité de travail varie entre 100 et 2000 par pas de 100. Comme précédemment, une unité de travail représente 1 Gflop de calcul et 2 Mbit de données. Les esclaves communiquent donc entre 0,06 et 0,4 secondes par tâche et calculent entre 9 et 90 secondes par tâches. Une fois de plus, le rapport entre les calculs et les communication

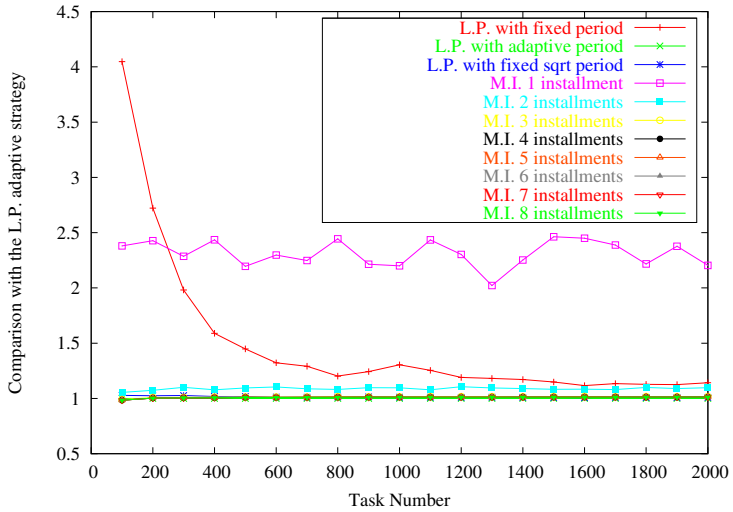


FIG. 6.8 – Comparaison avec l'heuristique adaptative pour des plates-formes hétérogènes à 5 processeurs.

est relativement élevé, ce qui permet d'obtenir de bonnes performances.

Nous avons donc lancé une simulation par plate-forme et par quantité de tâches à effectuer, ce qui représente 40000 expériences par heuristiques. Les résultats suivants correspondent à des moyennes des différentes valeurs. La figure 6.8 et sa version épurée figure 6.9 représentent la comparaison des différentes heuristiques avec l'heuristique adaptative sur des plates-formes à 5 processeurs. On peut faire les différentes observations :

- une fois de plus, l'heuristique à base de programmation linéaire avec une période fixe et l'heuristique M.I.1 ont de très mauvaises performances,
- le nombre de tournées optimal pour M.I.x est situé aux alentours de 4 et augmenter le nombre de tournées au delà de 5 n'apporte pas vraiment d'améliorations significatives,
- l'heuristique garantie est légèrement meilleure que les stratégies de Robertazzi quand le nombre de tâches est suffisamment grand,
- l'approche adaptative donne les meilleurs résultats, même si l'amélioration est extrêmement faible (autour de 1% en moyenne).

Quand la taille de la plate-forme augmente, le nombre optimal de tournées pour M.I.x augmente également et les programmations à base de programmation linéaires ont des performances correctes uniquement si la quantité de travail à distribuer est suffisamment grande. L'heuristique adaptative reste supérieure aux autres approches à base de programmation linéaire. À titre d'exemple, la figure 6.10 représente les performances que l'on peut obtenir pour une plate-forme à 20 processeurs.

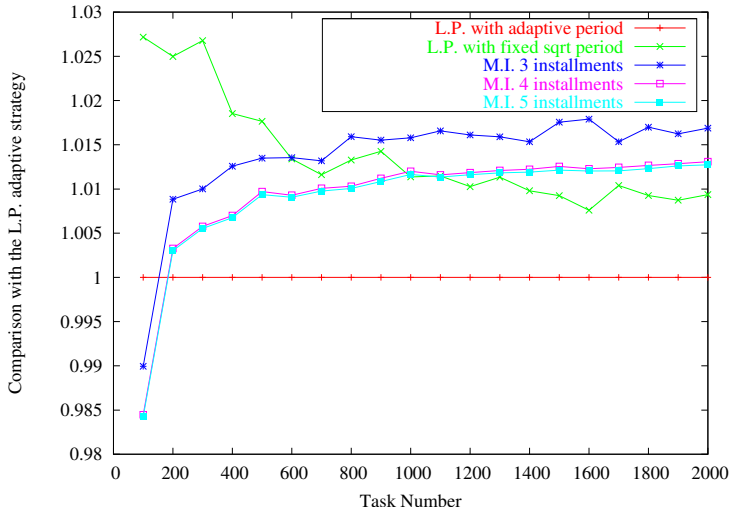


FIG. 6.9 – Comparaison avec l'heuristique adaptative pour des plates-formes hétérogènes à 5 processeurs (version épurée).

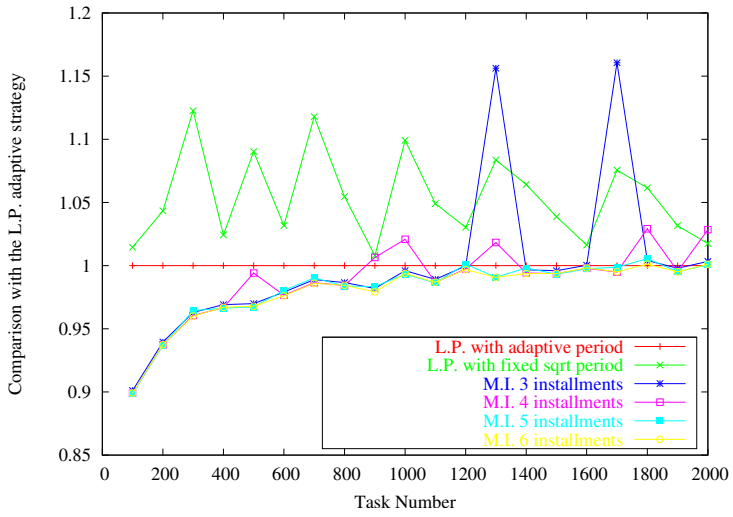


FIG. 6.10 – Comparaison avec l'heuristique adaptative pour des plates-formes hétérogènes à 20 processeurs.

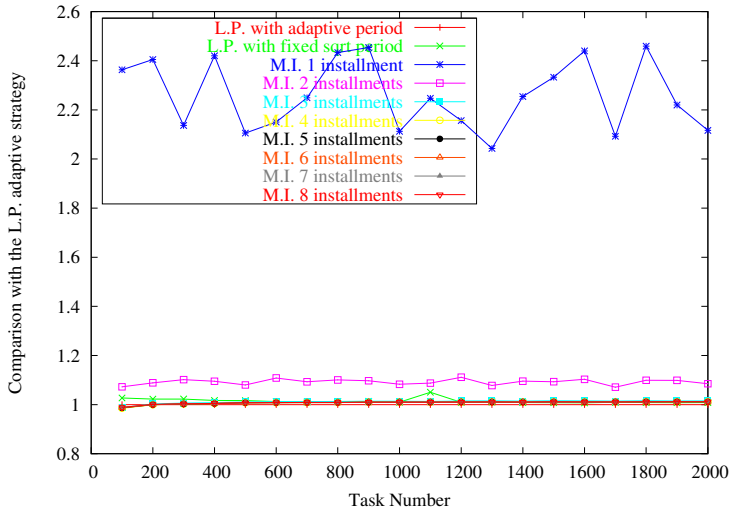


FIG. 6.11 – Comparaison avec l'approche adaptative pour les plates-formes hétérogènes à 5 processeurs, avec latence.

6.7.4 Plate-Forme hétérogène avec latences.

6.7.4.1 Avec un rapport calcul/communication élevé

Les expériences de cette section ont été effectuées avec les mêmes plates-formes et le mêmes paramètres pour les unités de travail qu'en section 6.7.3. La seule différences réside dans l'ajout d'un terme de latence modélisé par le transferts de 2 Mbit supplémentaires pour toute communication.

La figure 6.11 et sa version épurée figure 6.12 représentent la comparaison des différentes heuristiques avec l'heuristique adaptative sur des plates-formes à 5 processeurs. Les performances des distributions de Robertazzi quand le nombre de tournées est faible sont toujours mauvaises. Le nombre optimal de tournées pour ces stratégies semble être situé au alentours de 4. L'heuristique offrant les meilleurs performances est l'heuristique adaptative avec un gain assez faible de l'ordre de 1%. Le nombre optimal de tournées pour M.I.x augmente une fois de plus avec la taille de la plates-formes (voir figure 6.13) et se situe aux alentours de 5. La stratégie adaptative est toujours à au plus 3% de la meilleure distribution (quand le nombre de tâches à distribuer est supérieur à 200).

6.7.4.2 Avec un rapport calcul/communication faible

Les expériences de cette section ont été effectuées avec les mêmes plates-formes qu'en section 6.7.3. La seule différences réside dans la modification du rapport entre les calculs et les communications. Une unité de travail équivaut désormais à 50 Mflop de calcul et 2 Mbit de données à échanger, et le terme de latence est inchangé. Le rapport entre le calcul et les communications a donc environ été divisé par 20.

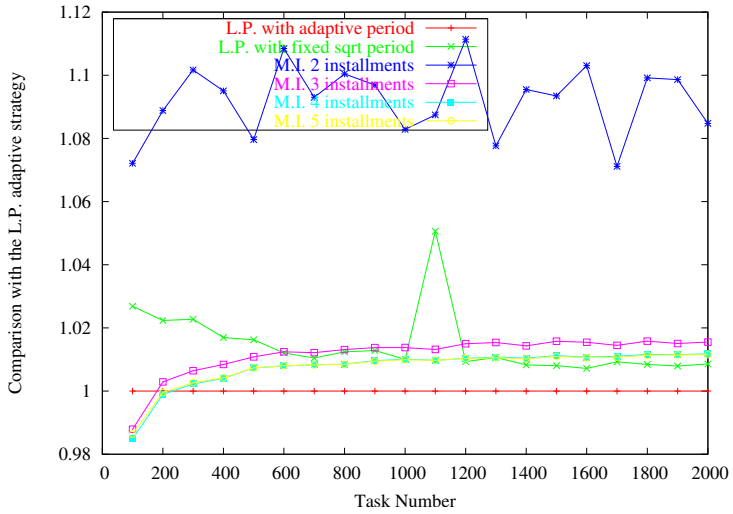


FIG. 6.12 – Comparaison avec l'approche adaptative pour les plates-formes hétérogènes à 5 processeurs, avec latence (version épurée).

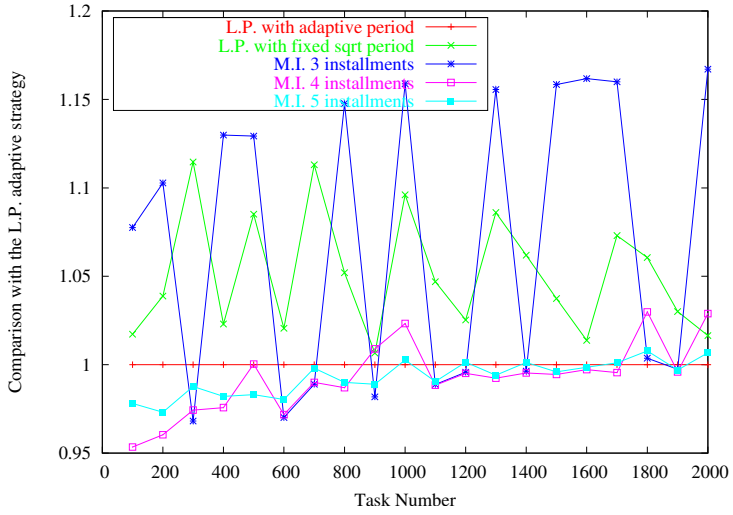


FIG. 6.13 – Comparaison avec les performances de l'heuristique adaptative sur une plate-forme homogène à 20 processeurs, avec latences.

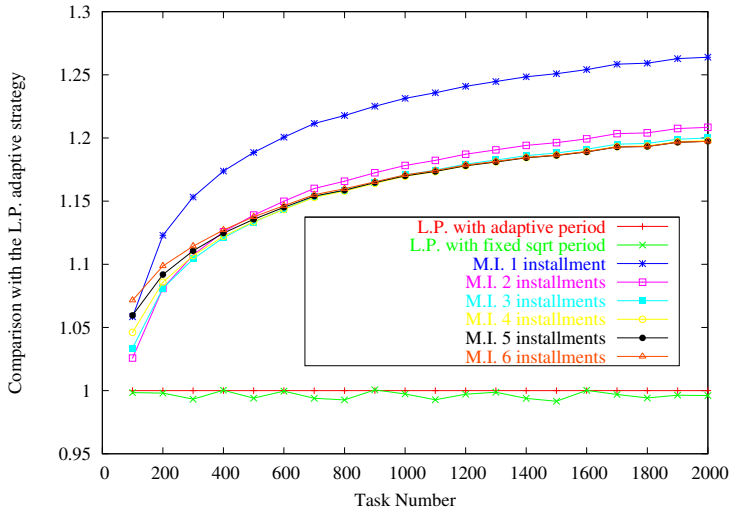


FIG. 6.14 – Comparaison avec les performances de l'heuristique adaptative sur une plate-forme homogène à 5 processeurs (rapport calcul/communication faible).

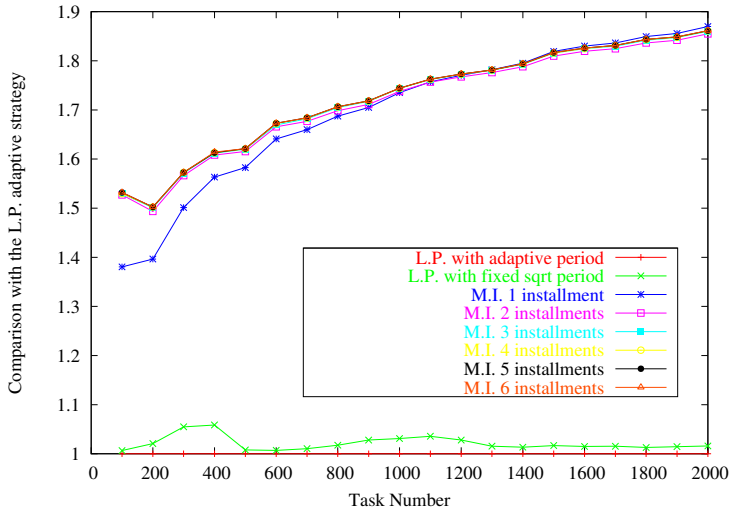


FIG. 6.15 – Comparaison avec les performances de l'heuristique adaptative sur une plate-forme homogène à 20 processeurs (rapport calcul/communication faible).

Les figures 6.14 et 6.15 représentent les comparaisons avec l'heuristique adaptative pour 5 et 20 processeurs. Les approches à base de programmation linéaire sont largement meilleures que les autres. Ceci peut s'expliquer par deux raisons :

- Les équations linéaires données dans [18] ne prennent pas en compte la latence. Le rapport calcul/communication étant faible, il devient crucial de les prendre en compte ;
- Aucune sélection des ressources n'est effectuée dans [18]. Quand la taille de la plate-forme devient importante, il devient crucial d'utiliser le réseau au mieux et de choisir les ressources de calcul à utiliser. Ce choix est effectué automatiquement lors de la résolution des programmes linéaires.

6.7.5 Synthèse

On peut conclure des expériences précédentes que les stratégies à base de programmation linéaires, qu'elles soient à période adaptative ou fixe basée sur une estimation du temps optimal de traitement des tâches, donnent d'excellents résultats en comparaison des techniques classiques. Dans la plupart des cas, elles sont proches des distributions de Robertazzi et déterminer le meilleur nombre de tournées pour cette classe d'heuristiques est loin d'être trivial. Quand le rapport entre les calcul et les communications devient faible, les stratégies à base de programmation linéaire sont bien plus performantes que les autres heuristiques. Enfin, la stratégie adaptative a l'avantage d'être *a priori* plus robuste aux variations de charge du réseau ou des machines.

6.8 Conclusion

D'un point de vue théorique, les résultats principaux de ce chapitre portent sur (i) la détermination de la solution optimale pour une étoile hétérogène et (ii) l'optimalité asymptotique de notre distribution en plusieurs tournées. Le premier résultat résout un problème ouvert depuis longtemps et le second est le premier résultat de ce type concernant les distributions en plusieurs tournées. De plus, notre distribution a un certain nombre de bonnes propriétés qui en font un candidat de choix pour une mise en œuvre réelle :

- La sélection des ressources est effectuée lors de la résolution du programme linéaire. Mieux, il existe un moyen simple d'effectuer cette sélection. Il suffit de trier les ressources par g_i et de les sélectionner jusqu'à ce que la somme des $\frac{g_i}{g_i+w_i}$ (sans recouvrement) ou la somme des $\frac{g_i}{w_i}$ soit supérieure à 1 alors que les méthodes proposées dans la littérature recouraient à des heuristiques pour effectuer leur sélection de ressources.
- Le nombre de tournées est simplement calculé en fonction du nombre de tâches, si bien qu'il n'y a pas besoin de tester différentes valeurs pour le nombre de tournées et de sélectionner la meilleure.
- Notre distribution étant périodique, elle est très simple à mettre en œuvre.

– Pour la même raison, cette distribution est capable de prendre en compte les variations de charge. Les décisions de distribution peuvent être réévaluées à chaque étape en fonction des caractéristiques de la plate-forme qui viennent d'être mesurées et ainsi s'adapter aux variations de performances de certaines machines ou de certaines parties du réseau.

Un grand nombre de simulations ont montré que notre distribution se comporte très bien en pratique et améliore notablement les heuristiques classiques quand le rapport entre les calculs et les communications devient faible.

Chapitre 7

Modélisation et Simulation

7.1 Introduction

Alors que la plupart des problèmes d'ordonnancement sont déjà difficiles sur une plate-forme de calcul homogène, ils deviennent encore plus difficiles dans un cadre hétérogène tel qu'une grille de metacomputing. Comme dans le cas homogène, dans le meilleur des cas, on peut arriver à trouver une heuristique garantie, mais la plupart du temps, ce n'est même pas possible. On a donc souvent recours à des expériences grandeur nature ou à des simulations pour pouvoir comparer deux heuristiques. Cependant, dans le cas d'une plate-forme de calcul hétérogène et distribuée, de telles expériences sont très délicates à mener en raison de l'instabilité latente de telles plates-formes. Il est en effet impossible de garantir que l'état d'une plate-forme de calcul qui n'est pas entièrement dédiée à l'expérimentation, va rester le même entre deux expériences, ce qui empêche donc toute comparaison rigoureuse. On utilise donc des simulations afin d'assurer la reproductibilité des expériences, toute la difficulté étant alors d'arriver à simuler de façon réaliste un tel environnement.

En effet, la plupart des résultats d'ordonnancement que l'on peut trouver dans la littérature sont obtenus grâce à des hypothèses assez restrictives et à des modèles assez simples. Les réseaux d'interconnexion sont souvent très simplistes (étoile, graphe complet, arbre, etc.) en regard de la réalité. Les ressources de calcul et de communication sont souvent supposées ne pas avoir de variation de performances et même si certaines heuristiques d'ordonnancement prennent en compte leur hétérogénéité, l'hypothèse que ces heuristiques sont capables d'obtenir des prédictions parfaites des différences de vitesses de leurs ressources est quasiment toujours faite. Il est nécessaire de faire ce type d'hypothèse simplificatrice pour comprendre certains phénomènes et réussir à concevoir des heuristiques efficaces. Cependant, ces hypothèses ne sont jamais vérifiées en pratique et le retour à la réalité est rarement effectué. L'impact de ces hypothèses simplificatrices (la prise en compte des variations de charge des ressources, l'impact de l'imprécision des performances, la contention qui peut survenir sur certains liens du réseau, etc.) sur une application réelle est très rarement étudié.

Pour y parvenir, une approche efficace consiste à effectuer des simulations utilisant des traces, c'est-à-dire utilisant des enregistrements de différents paramètres de la plate-forme pour obtenir un comportement réaliste. À cet effet, nous avons écrit SimGrid, un logiciel développé en collaboration avec Henri Casanova de l'Université de Californie, San Diego. C'est un simulateur modulaire écrit en C et permettant de simuler une application distribuée où les décisions d'ordonnancement peuvent être prises par différentes entités. La force de ce simulateur réside dans sa capacité à importer et simuler aisément des plates-formes réalistes (de la grille de metacomputing au réseau de stations de travail). L'objectif de ce simulateur n'est donc pas de prédire le temps que prendra telle ou telle application mais de pouvoir simuler un environnement d'exécution relativement réaliste pour comparer deux algorithmes.

Ce chapitre présente donc les enjeux et les problèmes liés à des simulations réalistes. En section 7.2, nous présentons les raisons pour lesquelles nous pensons que la simulation est une voie incontournable dans l'étude d'algorithmes d'ordonnancement sur plate-forme hétérogène. Nous y exposons également des travaux antérieurs en terme de simulation et expliquons en quoi ils ne sont pas adaptés à nos besoins d'ordonnanceurs et d'algorithmiciens. En section 7.3, nous présentons différents modèles utilisant des traces pour les ressources d'une plate-forme de calcul et qui sont mis en œuvre dans SimGrid. Nous expliquons ensuite en section 7.4 les différents concepts permettant de mettre en place une simulation d'ordonnancement distribué. Enfin, nous exposons en section 7.5 une approche possible pour la récolte de traces permettant de mettre en œuvre une simulation réaliste et nous concluons ce chapitre avec quelques pistes possibles en section 7.6.

7.2 La problème de la simulation

Comme nous l'avons déjà fait remarquer tout au long de cette thèse, les plates-formes de calcul actuelles sont *distribuées* et *hétérogènes*. Si un gros effort est actuellement effectué au niveau logiciel et matériel afin de permettre une bonne utilisation de ces plates-formes, l'écriture et l'optimisation d'une application sur de telles architectures restent encore bien délicates, principalement en raison de l'hétérogénéité du réseau d'interconnexion et des capacités de calcul des différents composants[1]. Cette difficulté est d'autant plus grande que les liens *Wide-Area* sont souvent partagés avec le trafic d'Internet et que leur performances ne sont donc pas aussi stables et fiables que celle d'une grappe homogène. À une autre échelle, les réseaux de stations hétérogènes sont rarement dédiés et les performances des processeurs, comme celles du réseau, peuvent grandement varier au cours du temps. Pour toutes ces raisons, la comparaison de deux algorithmes différents ou de deux mises en œuvre doit être faite avec les plus grandes précautions. L'étalonnage et l'optimisation d'une application nécessite de pouvoir l'exécuter de façon reproductible afin de pouvoir effectuer des comparaisons rigoureuses. Néanmoins, quand bien même il serait techniquement possible d'isoler une plate-forme du reste du monde afin de pouvoir la remettre exactement dans l'état où elle était et pouvoir ainsi réaliser des comparaisons

[1] sans même parler des problèmes techniques posés par le déploiement d'une application réelle

honnêtes, il n'en reste pas moins que le comportement des applications doit être testé dans des conditions réalistes où les performances du réseau et des processeurs peuvent être variables.

La majorité des résultats classiques d'ordonnancement est obtenue avec des hypothèses restrictives. Les ressources de calcul sont souvent homogènes et capables de délivrer une puissance constante. Parmi les autres hypothèses peu réalistes, mais utiles d'un point de vue théorique puisqu'elles permettent en général de montrer que même dans des cas très favorables, les problèmes sont extrêmement difficiles, on trouve la mise à disposition d'une infinité de processeurs, ou l'interconnexion complète des processeurs. Même si certains travaux, comme ceux présentés dans cette thèse, prennent en compte l'hétérogénéité de la plate-forme à différents degrés, il est toujours supposé que les prédictions sur les performances des ressources et les temps d'exécution des tâches sont parfaites.

Toutes ces hypothèses sont bien évidemment irréalistes sur des plates-formes telles que celles que nous visons. La topologie du réseau d'interconnexion est généralement bien plus complexe que celle simulée dans la majorité des travaux d'ordonnancement. Les performances des ressources variant au cours du temps, il est extrêmement délicat de déterminer une prédiction exacte du temps d'exécution d'une tâche. Une mise en œuvre d'un algorithme classique utiliserait donc des prédictions très approximatives et l'impact de ces imprécisions sur les performances de l'ordonnancement n'est que très rarement étudié.

Il y a donc un réel besoin d'outils de simulation permettant des modélisations de plates-formes et d'applications plus réalistes. Si un certain nombre de simulateurs existent déjà (d'un simple réseau à un système distribué complet), ils n'ont pas pour objectif la comparaison d'algorithmes d'ordonnancement. Ils sont généralement très complets et sophistiqués mais inadaptés à notre cadre. Dans cette section, nous en présentons quelques-uns et expliquons en quoi ils ne sont pas adaptés à nos besoins.

7.2.1 Simulateurs de réseaux

De façon à mieux comprendre le comportement des réseaux dans différentes situations, un certain nombre de simulateurs ont été développés [89]. Ces simulateurs, comme NS, NSE [6] ou DaSSF [77], s'intéressent à une simulation précise du mouvement des paquets circulant sur le réseau, plus qu'au comportement du réseau tel qu'il peut être perçu par une application. Ils ont pour objectif l'aide au développement de protocoles réseaux plus rapides et plus réactifs. Les applications utilisant ces protocoles n'apparaissent par contre pas dans la simulation. Tout au plus peuvent-elles être vues comme des émettrices de paquets. Il est possible de simuler des applications réelles, comme dans DaSSF, mais cela nécessite une réécriture complète de l'application adaptée à la description d'un tel simulateur, et il est alors quand même très difficile de modéliser les temps de calcul.

De plus la charge externe doit être simulée en créant des sources artificielles de trafic. Si des résultats statistiques existent sur la topologie des réseaux *Wide-Area* (voir section 7.2.3), ils restent difficiles à utiliser dans un tel cadre. La modélisation de la charge

externe est encore plus difficile (il suffit d'imaginer le nombre de connexions par heure au dessus d'Internet). Enfin, ces simulateurs sont généralement très lents puisqu'ils simulent de façon extrêmement précise le mouvement des paquets dans le réseau. L'augmentation du nombre de connexions pour simuler la charge externe ralentit donc d'autant plus le temps de simulation.

7.2.2 Simulations d'applications

Rappelons que notre objectif est de pouvoir tester rapidement l'efficacité d'un algorithme d'ordonnancement et non pas d'exécuter effectivement une application. Les différents projets présentés dans cette section replient, comme nous allons le voir, une plate-forme sur une grappe pour virtualiser la plate-forme. Ce type d'approche augmente d'autant plus le temps de la simulation et n'est vraiment utile que si l'on a déjà une vraie application, et pas seulement un algorithme, à tester.

LAPSE. LAPSE (Large Application Parallel Simulation Environment) [45] est un outil destiné à la simulation d'applications parallèles. L'objectif est principalement de permettre aux chercheurs d'exécuter leur application sur un petit nombre de nœuds car les grosses machines parallèles ne sont pas forcément disponibles pour simplement tester les performances et l'extensibilité de leur algorithme.

Le simulateur et l'application sont parallélisés sur l'ensemble des nœuds disponibles. Le code de l'application est instrumenté de façon à compter le nombre d'instructions entre les différentes communications, ce qui permet de savoir combien de temps simulé se passe entre chaque communication. Une fois l'application effectivement exécutée sur un plus petit ensemble de machines (mais avec le même nombre de processus que si elle avait été exécutée sur un grand nombre de processeurs), il est donc possible de reconstituer le temps qu'elle aurait pris sur un plus grand nombre de processeurs.

Ce simulateur a été conçu pour les applications écrites pour les machine Paragon d'Intel, même s'il est possible de l'adapter pour d'autres types de machines comme une grappe de processeurs. La modélisation du réseau et la reconstruction du temps simulé ont cependant été conçus avec cette machine à l'esprit, et il n'est pas forcément aisé de l'adapter à une plate-forme actuelle.

The MicroGrid. MicroGrid [103] fait partie du projet GrADS (Grid Application Development Software) dont l'objectif est de simplifier le calcul hétérogène distribué. MicroGrid offre aux programmeurs d'application pour la Grille la possibilité d'exécuter leur application sur une grille virtuelle dont le comportement est bien mieux maîtrisé qu'une vraie grille. Plus qu'une simulation, cette approche tient plus de l'*émulation*.

Ce logiciel permet donc de virtualiser chaque ressource d'une grille classique, telle que la mémoire, le calcul, le réseau, etc. Comme dans LAPSE, cette virtualisation est effectuée en interceptant chaque appel pertinent (ouvertures de *sockets*, `gethostname`, etc.). Cependant, la simulation n'est pas aussi stricte qu'avec LAPSE.

Au lieu de compter précisément le nombre d'instructions, les ressources de calcul sont simulées à l'aide d'un ordonnanceur local qui alloue des périodes d'exécution à chaque processus en fonction d'une vitesse de simulation prédéterminée. Il est ainsi possible de replier une grille sur une grappe puissante, en allouant des périodes plus importantes aux processus qui sont censés s'exécuter sur des processeurs rapides. Le réseau est simulé à l'aide d'une version temps-réel du simulateur VINT/NSE [6]. Il gère toutes les communications mais la configuration du réseau fait partie de la grille virtuelle et n'est pas censée varier au cours du temps.

Avec cet outil, comme la plate-forme est simulée en la projetant sur une grappe, l'application est effectivement exécutée au ralenti et le rapport entre le temps simulé et la durée de la simulation ne peut donc être bien élevé. Il l'est d'autant moins que la simulation de la partie réseau est limitante (c'est la raison pour laquelle ils envisagent d'utiliser DaSSF qui lui est parallèle). De plus, cette approche hérite également des inconvénients posés par les simulateurs de réseaux, comme la difficulté de simuler une charge externe réaliste. Enfin, il faut d'abord disposer d'une vraie application puis la compiler avec une bibliothèque modifiée de Globus pour pouvoir effectuer la simulation, ce qui ne simplifie pas forcément le développement d'algorithmes *exotiques*.

Panda. Le projet Albatros porte sur l'exécution d'applications hautes performances sur des collections de grappes ou de machines parallèles connectées par un réseau *Wide-Area*. L'objectif initial de la bibliothèque Panda [37] est de fournir une couche de portabilité efficace pour la programmation d'applications parallèles. Des travaux récents ont porté sur la simulation de variation de charge dans cette bibliothèque [72]. L'approche adoptée est originale et n'utilise pas de simulateur pour le réseau. On utilise des liens réels qui sont ralentis artificiellement pour simuler le comportement du réseau *Wide-Area*. Cette simulation a donc lieu dans des emplacements bien particuliers de la bibliothèque de communication. En forçant les messages qui auraient dû emprunter un lien *Wide-Area* à passer par des machines particulières et en les mettant dans des files d'attentes on ralentit ainsi certains liens à volonté.

À la différence des deux projets précédents, il est aisé de modifier le comportement du réseau au cours de la simulation. Cependant, ce genre d'outil n'est, une fois de plus, pas vraiment adapté à notre besoin d'évaluation d'algorithmes d'ordonnancement. Il faut disposer d'une application réelle, cette dernière est exécutée en temps réel et l'hétérogénéité et l'instabilité de la plate-forme n'apparaît qu'au niveau du réseau ; au niveau des liens *Wide-Area*, pour être exact. Ce type d'outil est parfaitement adapté aux besoins de ses concepteurs puisque leur plate-forme de test est composée de quatre grosses grappes homogènes distribuées dans les différentes universités du pays et reliées par un réseau ATM.

7.2.3 Générateurs de topologies

Revenons un instant sur le problème de la topologie du réseau. La plupart des simulations de la littérature utilisent des modèles simples (une étoile, un anneau, un hypercube,

etc.), une représentation de topologies réelles (par exemple Arpanet ou la carte d'un réseau qu'un fournisseur d'accès a consenti à rendre public) ou encore une topologie générée aléatoirement. Cette dernière approche a connu un grand intérêt ces dernières années.

Les premiers modèles sont dit *plats*. Le plus stupide consiste à placer des points aléatoirement dans un rectangle et à les connecter avec une probabilité fixe. Ce type de modèle n'a bien évidemment aucune raison de rendre compte de la réalité d'un vrai réseau. Waxman introduit en 1988 [112] un modèle simple qui a séduit grand nombre de personnes. Dans ce modèle, deux nœuds u et v sont reliés avec la probabilité suivante :

$$P(u,v) = a.e^{-d(u,v)/(bL)},$$

où $d(u,v)$ est la distance entre u et v, a et b sont des réels appartenant à $]0,1]$ et L la distance maximale entre deux points (c'est-à-dire la longueur de la diagonale du rectangle qui les contient). Le paramètre a est lié à la densité du graphe, c'est-à-dire au nombre d'arêtes. Le paramètre b, lui est lié à l'hétérogénéité de la longueur des arêtes. Un certain nombre d'études ont été réalisées avec ce modèle.

Plus récemment, des modèles de générations de réseaux, plus hiérarchiques, ont été conçus en vue de mieux appréhender la structure hiérarchique d'Internet. Des études sur la topologie d'Internet ont vu le jour [47, 29, 53, 83]. Celle de Michalis, Petros et Christos Faloutsos [53] a été particulièrement marquante. En utilisant des instantanés de la topologie d'Internet à différentes périodes, ils ont établi des lois exponentielles simples décrivant les graphes en question et leur évolution. Ces lois relient par exemple le degré des nœuds à leur fréquence, ou portent sur la fréquence des degrés, la taille des voisinages, les valeurs propres de la matrice d'adjacence. Le point positif de cet article a été la prise de conscience par cette communauté que les lois uniformes utilisées jusqu'ici n'étaient pas adaptées et que la plupart des générateurs devaient donc être fortement soumis à caution.

Un des points négatifs est l'engouement pour ces lois qui s'en est suivi. Partant du principe qu'un générateur ne suivant pas ces lois n'était pas bon, les *modèles structuraux* (Transit-Stub [29] ou Tiers [47]) qui privilégiaient l'organisation hiérarchique ont été dénigrés et abandonnés. Dans le même ordre d'idées, un bon générateur se devant de suivre ces lois, des *modèles basés sur le degré* comme celui de Barabási et d'Albert [8] ont été créés. Une croissance incrémentale et une affinité proportionnelle au degré permet de générer des graphes dont on peut démontrer qu'ils suivent ces lois exponentielles. On s'est cependant aperçu depuis que certaines de ces lois étaient reliées entre elles (c'est-à-dire que si l'une était vérifiée les autres l'étaient également), ce qui diminue singulièrement leur intérêt [83, 84]. Enfin, les graphes de Barabási et d'Albert n'exhibent de caractère hiérarchique que quand ils sont suffisamment grands [107]. Ces modèles n'ont donc de sens que lorsqu'ils comportent au moins quelques milliers de nœuds et les utiliser pour des graphes plus petits n'a pas vraiment plus de sens que d'utiliser des lois uniformes ou des modèles simples. Il semblerait donc que les *modèles structuraux* soient plus adaptés à la modélisation de plates-formes composés d'une centaine de nœuds. Ces modèles ont d'ailleurs été remis au goût du jour avec GridG [43] qui se base sur Tiers.

Si ces générateurs sont un excellent point de départ pour la construction d'une plate-forme réaliste, il manque un certain nombre d'informations cruciales comme la charge

externe (c'est-à-dire la bande passante ou la puissance de calcul disponible au cours du temps) ou les caractéristiques macroscopiques du réseau (a-t-on affaire à du simple Ethernet non *switché* ou à une grosse fibre optique ?). La technique classique pour simuler la charge externe consiste à utiliser des sources de trafic fictives en utilisant des lois aléatoires dont la validité est souvent douteuse. En effet, en raison de la variété de sources de trafic différentes que l'on peut trouver sur Internet, du développement continuel de l'utilisation du réseau et de l'arrivée croissante de nouveaux utilisateurs, il est difficile de garantir que de telles sources de trafic sont représentatives de la réalité. Dans un environnement de test utilisant des sources *réelles* plutôt que *simulées*, ce problème de la confrontation avec la réalité est grandement simplifié. Enfin, tous ces générateurs sont souvent très évolués et il est du ressort de l'utilisateur de fixer les différents paramètres, ce qui nécessite une grande expérience et une bonne intuition de l'utilisation de ces outils.

7.3 Modélisation à base de trace

Dans cette section, nous présentons dans un premier temps différents modèles disponibles dans SimGrid pour simuler des ressources simples. Nous présentons ensuite des travaux portant sur une modélisation plus fine du réseau.

7.3.1 Modélisation d'une ressource simple

SimGrid modélise les liens réseau ou les processeurs d'une plate-forme comme une collection de *ressources*. Chaque ressource est modélisée par une *latence* L (en secondes) et un débit D (la *bande passante* en octets/seconde ou la *puissance* en flop/seconde). Les valeurs de latence et de débit peuvent être constantes ou variables à l'aide de trace. Une trace est une série de valeurs datées pouvant être générée artificiellement ou à partir de mesures d'un réseau réel.

Ainsi, la durée T d'un transfert de S octets (ou d'un calcul équivalent à S flop) commençant à un temps t_0 est définie par l'équation suivante (voir figure 7.1) :

$$\int_{t=t_0+L(t_0)}^{t_0+T} B(t)dt = S$$

Quand la latence et la bande passante sont constantes, on retrouve le modèle classique :

$$T = L + \frac{S}{B}$$

Dans sa version la moins évoluée, une simulation avec SimGrid se fait en ordonnançant des tâches (qui peuvent avoir des dépendances entre elles) sur des ressources. SimGrid propose quatre modes d'accès différents à une ressource :

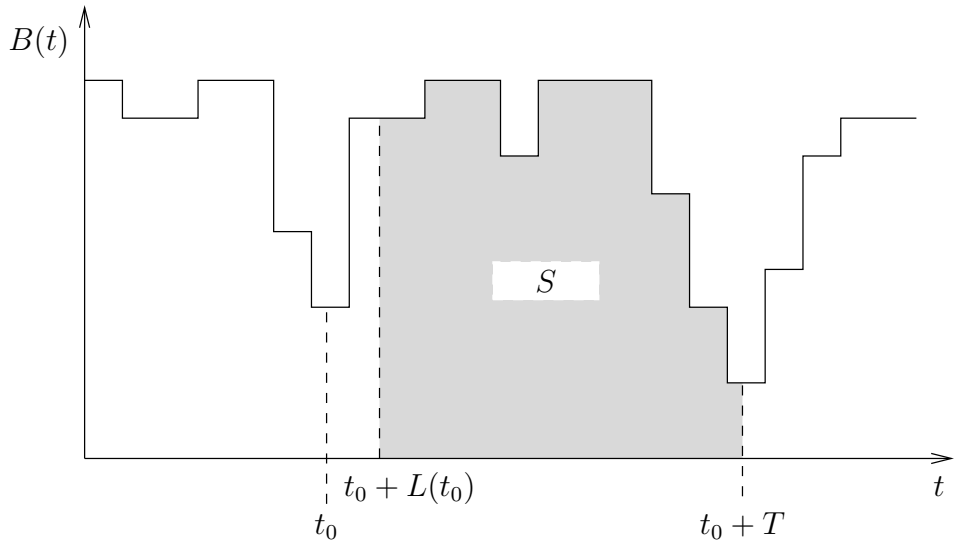


FIG. 7.1 – Illustration de l'intégration d'une trace.

ordonné. Ce mode-ci, comme le suivant est séquentiel. Lorsqu'une tâche est ordonnancée sur une telle ressource, celle-ci est mise dans une file d'attente et est exécutée dès qu'elle est prête et dès que la ressource n'est plus occupée.

non-ordonné. Dans ce mode-ci, lorsqu'une tâche est ordonnancée sur une telle ressource, elle est mise dans une file d'attente et quand la ressource n'est plus occupée, la première tâche prête dans la file est exécutée.
Ces deux modèles peuvent être adaptés à la modélisation d'un processeur utilisé avec un système de *batch* et garantissant l'exclusivité de son utilisation.

partagé. Dans ce mode-ci, comme dans le suivant, une exécution de tâche commence dès qu'elle est prête et indépendamment des tâches en cours de traitement sur la ressource. Dans le mode *partagé*, le débit de la ressource est partagé entre les différentes tâches qui s'y exécutent. Autrement dit, s'il y a N exécutions simultanées sur une ressource de débit D, chaque tâche se voit attribuer un débit D/N. Un tel partage est dit *équitable*. Ce type de modèle est particulièrement adapté à un processeur multi-programmé ou à un lien réseau simple partagé entre différentes personnes.

non-partagé. À la différence du modèle précédent, il n'y a pas de partage des ressources et chaque tâche se voit attribuer un débit D. Ce type de modèle est adapté, par exemple, à un lien *Wide-Area* qui dispose d'une bande passante donnée mais qui est capable de supporter une montée en charge importante.

7.3.2 Modélisation d'un ensemble de ressources de communication

Si ce type de modèle s'est avéré pertinent lorsque la plate-forme est simple (quelques liens réseaux, et quelques processeurs), quand l'interconnexion de la plate-forme se complexifie, on peut remettre en cause sa validité à rendre compte de la réalité. En effet,

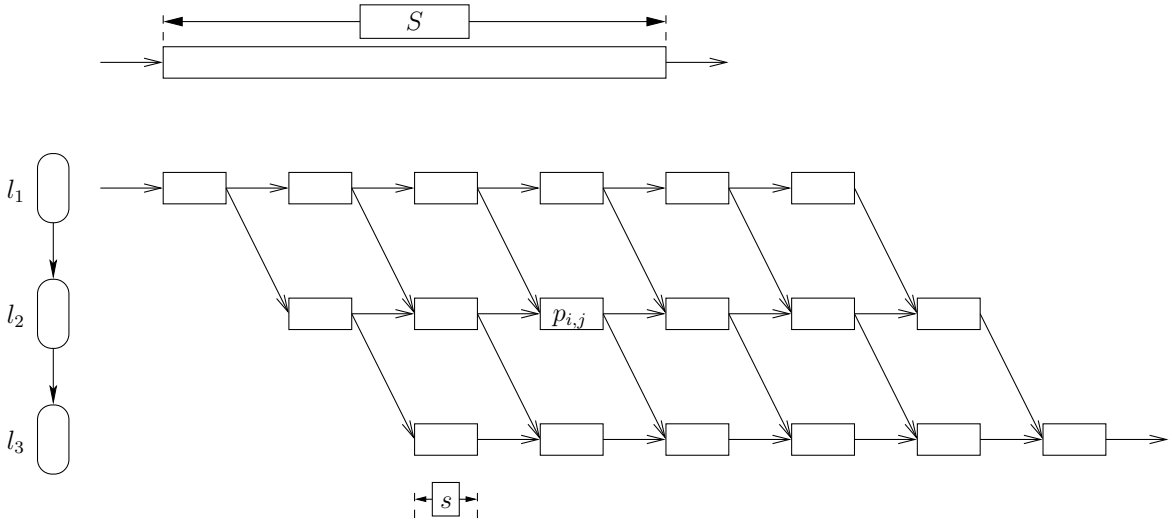


FIG. 7.2 – Discrétisation du transfert d'un fichier de taille S en paquets de taille s sur les liens l_1, l_2 et l_3.

comment gérer un transfert impliquant un certain nombre de ressources ? Cela est nécessaire pour pouvoir modéliser les phénomènes de contention et les goulots d'étranglement. SIMGRID propose deux modèles différents pour prendre ce phénomène en compte.

Le premier consiste à découper un fichier de taille S en petits paquets de taille s et à organiser le transfert des petits paquets avec une politique *store-and-forward*. Le transfert du $i^{\text{ème}}$ paquet ne peut commencer sur le lien l_j que dès qu'il a été transféré sur le lien l_{j-1} (voir figure 7.2). En diminuant la taille des paquets, on se rapproche de l'idée que l'on peut se faire du transfert d'un gros fichier. Néanmoins, ce faisant, on augmente considérablement le temps de simulation et on se rapproche d'une simulation dont la philosophie est contraire à celle d'une simulation macroscopique à base de trace. De plus, ce type de simplification était peut-être valable sur le réseau d'une Paragon mais elle ne l'est plus maintenant car elle néglige complètement les temps d'attente dans les routeurs, les retours d'accusés de réception qui peuvent limiter la bande passante ou encore le phénomène de *slow-start* de TCP.

Un certain nombre de travaux récents ont proposé des modèles théoriques du partage de bande passante dans un réseau [93, 81, 22]. La plupart procèdent par analogie avec des fluides. Ils supposent que l'écoulement des *flots* est parfaitement continu et ignorent les problèmes de granularité posés par la taille des paquets. Ils se positionnent également en régime permanent de façon à gommer les phénomènes de *slow-start* dûs à la granularité d'une communication réelle. Ce type de modélisation ne s'applique donc qu'aux transferts de fichiers de taille conséquente.

Ainsi, si on considère un réseau constitué d'un ensemble $\mathcal{L}$ de *liens* où chaque lien $l \in \mathcal{L}$ a une capacité $c_l > 0$, une *route* r est une suite de liens. Chaque route (ou flot) a donc un certain débit ρ_r et pour un ensemble de routes $\mathcal{R}$ donné, on a donc la contrainte

suivante :

$$\forall l \in \mathcal{L}, \quad \sum_{r \in \mathcal{R} \text{ t.q. } l \in r} \rho_r \leqslant c_l,$$

signifiant que la capacité d'un lien ne peut être excédée.

Reste donc à trouver une allocation de bande passante représentative du comportement d'un réseau comme celui que l'on peut utiliser dans le cadre de calcul hétérogène sur une grille de calcul.

Équité Max-Min C'est le modèle traditionnel présenté par exemple dans [16]. L'objectif est maximiser le minimum des ρ_r, c'est-à-dire de trouver une allocation telle que l'augmentation d'un ρ_r tout en conservant une allocation valide diminue nécessairement un $\rho_{r'}$ tel que $\rho_{r'} < \rho_r$. Ce type de contrainte conduit à la contrainte suivante :

$$\forall r \in \mathcal{R}, \quad \exists\, l \in r \text{ t.q. } \sum_{r' \ni l} \rho_{r'} = C_l \text{ et } \rho_r = \max\{\rho_{r'}, r' \ni l\}$$

Autrement dit, pour tout flot, il y a un goulot d'étranglement, et au niveau de ce goulot d'étranglement ce flot fait déjà partie des flots les mieux servis.

Équité proportionnelle Cette notion a été introduite par Kelly [71] qui mettait en cause la pertinence de l'équité Max-Min qui a tendance à favoriser les routes longues. Ce principe est décrit dans [81, 93] et consiste à trouver une allocation maximisant la somme $\sum_{\mathcal{R}} \rho_r log(\rho_r)$. Une telle allocation est unique et vérifie

$$\text{Pour toute autre allocation} (\rho'_r)_{r \in \mathcal{R}}, \quad \sum_{r \in \mathcal{R}} \frac{\rho'_r - \rho_r}{\rho_r} \leqslant 0.$$

Minimisation du temps de transfert Une autre idée pour l'allocation de bande passante consiste à minimiser le temps nécessaire à la terminaison de tous les transferts.

La question que l'on peut alors se poser est, «laquelle de ces allocations est représentative de l'utilisation d'un réseau avec le protocole TCP ?». Tout le monde semble s'accorde sur le fait que le protocole TCP produit des allocations *proches* de l'équité Max-Min. Chiu par exemple, montre dans [38] que TCP n'implémente pas exactement l'équité Max-Min mais que cette approximation n'est pas trop mauvaise. En fait, l'allocation de bande passante pour un flot donné semble être bornée par une quantité proportionnelle à l'inverse du *round-trip-time* (RTT).

Un certain nombre de travaux récents sur la modélisation de TCP à un niveau plus microscopique prennent en compte la perte des paquets. Dans [88], par exemple, on peut trouver une étude stochastique prenant en compte les mécanismes d'acquittement pour la gestion de la congestion, la perte de paquets, etc. Cette étude synthétise le comportement macroscopique en une formule faisant intervenir tous ces paramètres (RTT, taux de perte, le taux d'acquittement). Si on suppose les différents paramètres non simplement mesurables constants, on obtient une nouvelle fois une relation où le taux de transfert est inversement proportionnel au RTT. Cette relation est donc couramment acceptée [54, 82].

S'inspirant de ces travaux, un nouveau modèle macroscopique a été proposé par Loris Marchal et Henri Casanova dans [33] et implémenté dans SimGrid. Il a été validé par rapport à NS [6] qui fait référence en terme de simulateurs de TCP au niveau *microscopique*.

Le RTT est une valeur fondamentale qu'il convient de modéliser convenablement. Il est défini comme le temps moyen entre l'émission d'un paquet et la réception de son acquittement. Intuitivement, le RTT est donc égal à deux fois la latence d'un lien quand la route n'utilise qu'un seul lien et à deux fois la somme des latences des différents liens quand la route est plus complexe. Cependant, ce type de calcul ne prend pas en compte les pertes, le temps passé dans les différents routeurs, le *piggybacking*, etc. Quoiqu'il en soit, c'est une des seules valeurs qu'il est possible de mesurer simplement (contrairement à la latence) et donc d'utiliser.

Un flot seul passant par un lien simple est obtenu avec la relation suivante :

$$\mathrm{BW}_{exp} = \min\left(\mathrm{BW}_{max}, \frac{\gamma}{\mathrm{RTT}_{exp}}\right),$$

où γ est une constante à déterminer et dépendant du réseau. Dans le cas où l'on dispose de plusieurs flots partageant des liens, on commence par déterminer le lien tenant lieu de goulot d'étranglement (c'est-à-dire le lien tel que la bande passante restante divisée par le nombre de flots l'utilisant est minimum). Une fois ce lien $\tilde{l}$ déterminé, chaque flot r_i l'utilisant reçoit une fraction de la bande passante restante de $\tilde{l}$ proportionnelle à

$$1/\mathrm{RTT}(r_i)$$

On ajoute de plus que la portion de bande passante ρ_i allouée à r_i est bornée par

$$\rho_i \leqslant \gamma/\mathrm{RTT}(r_i)$$

pour que chaque flot soit effectivement bornée par une quantité inversement proportionnelle à son RTT et on répète l'opération jusqu'à ce que la valeur de chaque flot ait été déterminée.

7.4 Simulation d'un ordonnancement distribué

Dans cette section, nous présentons quels types d'études peuvent être réalisées simplement avec les modèles précédents, et nous introduisons les concepts qui nous paraissent nécessaires à la mise en œuvre d'une simulation qui soit la plus réaliste possible.

7.4.1 Nécessité de l'étude des ordonnancements distribués

Les modèles et concepts précédents (allocation de tâches sur des ressources) permettent de mettre en place très rapidement une simulation où la plate-forme n'est pas

trop compliquée et où les décisions d'ordonnancement sont centralisées. La première version de notre simulateur proposait l'utilisation directe de ces différents modèles. Il s'est avéré extrêmement pénible et délicat de mettre en œuvre un ordonnancement non centralisé comme celui qui découle naturellement des techniques orientées bande passante et qui sont présentées dans la section 4.4. Lorsque les décisions d'ordonnancement sont prises indépendamment les unes des autres par des entités communicantes, il est délicat d'exprimer ces décisions autrement.

Plus généralement, dans un système distribué à grande échelle, il est difficile, voire impossible, d'ordonnancer correctement des applications en utilisant uniquement un mécanisme global. Les ordonnancements centralisés ont déjà démontré un grand nombre de faiblesses : ils sont moins tolérants aux pannes, ne passent pas bien à l'échelle [30], les politiques d'ordonnancement dépendent généralement très fortement de contraintes locales [76, 78],etc. Il est donc crucial de pouvoir modéliser et étudier simplement ce type de systèmes où un certain nombre de décisions indépendantes peuvent être prise en même temps.

Pour pouvoir être extensible, il est commode que des ordonnanceurs puissent décider de se dupliquer et de se répartir en des lieux stratégiques pour mieux gérer la charge de travail à la quelle ils doivent faire face. L'organisation d'une plate-forme de calcul étant souvent très hiérarchique, la position des différents ordonnanceurs et leur politique de délocalisation est cruciale. Un des aspects du projet DIET [30] concerne justement le placement et le mouvement des différents agents d'ordonnancement sur la plate-forme.

Il était donc crucial que SimGrid offre des abstractions pour modéliser une application composée d'agents communicants et effectuant différentes tâches de calcul. Gardant néanmoins à l'esprit que l'objectif de cet outil est de simplifier la tâche de l'algorithmicien ou de l'ordonnanceur, il fallait donc trouver une interface suffisamment simple pour pouvoir s'abstraire des problèmes classiques liés à l'utilisation d'une plate-forme distribuée à grande échelle, et suffisamment proche de la réalité pour éviter la mise en place de situations totalement irréalistes.

7.4.2 Concepts de base

Fort de notre expérience d'utilisation de différentes bibliothèques de communications comme PVM [59], MPI [101] ou PM^2 [109], et conscients des besoins très spécifiques qu'un algorithmicien pouvait avoir, nous avons utilisé les abstractions suivantes.

Agent. On a besoin de pouvoir simuler des décisions d'ordonnancement indépendantes les unes des autres. La notion d'*agent* (ou *processus*) est donc au cœur de SimGrid. Un agent est défini simplement par un code et un lieu d'exécution (ou hôte). Actuellement, l'interface proposée pour définir le code en question suit le prototype classique d'une fonction principale en C (c'est-à-dire un `int main(int argc,char **argv)`), ce qui permet d'être extrêmement générique. Cependant, de façon à englober dans le même temps le concept de processus léger qui est indispensable dès que l'on souhaite introduire de l'asynchronisme et de la réactivité, il est également

possible d'attacher des données à un agent au moment de sa création et ainsi partager des données avec l'agent créateur.

Hôte. Un *hôte* (ou *lieu d'exécution*) est un endroit où un agent peut s'exécuter. Il est donc représenté par une ressource de calcul partagée (de façon à ce que plusieurs agents puissent s'exécuter sur la même ressource) définie par la trace de la puissance qu'elle est capable de délivrer et par des boites aux lettres permettant aux agents de communiquer entre eux. La notion de boite aux lettres est liée au concept de *canal* que nous détaillerons plus tard. Enfin, il est possible d'attacher à un hôte des données partagées par les agents s'exécutant sur l'hôte en question.

Tâche. De nombreux modèles d'ordonnancement s'appuient sur la notion de *tâche* pouvant être soit traitée localement, soit transférée à un autre endroit. Cela nous a semblé être le niveau d'abstraction le plus adapté à nos besoins. Une tâche est donc définie par une quantité de travail et la taille des fichiers qui l'accompagnent. Notons qu'avec ce type d'expression d'un ordonnancement, une tâche n'est pas exécutée ou transférée dès qu'une ressource est libre (par exemple) mais dès qu'un agent décide l'exécuter ou de la transférer. La notion de dépendance entre tâches n'a donc pas lieu d'être quand de telles tâches sont manipulées par des agents, car les dépendances sont gérées par l'ordonnancement. De plus la notion de dépendance entre tâches peut grandement dépendre d'un modèle à l'autre et en obligeant l'utilisateur à exprimer son ordonnancement sous forme d'agents communicants, on reste générique et proche de la réalité. Afin de permettre des modélisations complexes, il est donc évidemment possible d'adjoindre à une tâches des données privées qui peuvent servir à identifier le type de tâches et les autres tâches dont elle dépend.

Route. Une route est une suite de ressources de communications. Chacune de ces ressources a sa propre trace de bande passante et de latence et peut appartenir à plusieurs routes différentes. Chaque paire d'hôte est reliée par une route et les communications d'un agent à l'autre se font donc en n'ayant pas d'accès direct aux ressources de communication.

Canal. Pour permettre à certains agents de communiquer entre eux sans interférer avec les autres, nous utilisons la notion de canal. Un canal est un simple numéro de boite aux lettres et seul les agents intéressés par les informations circulant sur un canal y accèdent. Cette notion est proche de la notion de *canal* de PM^2 ou de *tag* dans MPI.

Pour résumer, un ordonnancement est donc décrit en terme d'agents communicants, s'exécutant sur des hôtes, et pouvant envoyer, recevoir ou exécuter des tâches. Un agent ne peut accéder directement aux ressources de calcul ou de communications mais peut les utiliser par le biais des différentes fonctions de traitement des tâches. Il peut par exemple utiliser le processeur de l'hôte sur lequel il est situé en exécutant une tâche. Il utilise les différents liens du réseau en envoyant une tâche à un autre hôte à travers un canal. La réception d'une tâche se fait simplement en consultant la boite aux lettres (de l'hôte sur lequel l'agent s'exécute) correspondant à canal donné. C'est ensuite aux différents agents de gérer les dépendances possibles entre les différentes tâches ainsi que leur traitement.

La mise en place d'une simulation doit suivre les étapes suivantes :

1. définition du code des différents agents,
2. création de la plate-forme d'exécution (les hôtes, les liens, les routes, etc.),
3. déploiement des agents sur la plate-forme,
4. lancement de la simulation.

Mise à part la première étape qui est très dépendante du type de problème auquel on s'intéresse, les autres étapes ont été automatisées au maximum, de façon à pouvoir étudier simplement un grand nombre de situations différentes.

7.4.3 Gestion de la simulation

Une fois le code de chaque agent défini, nous devons simuler leur exécution en parallèle. Nous avons donc utilisé dans un premier temps des pthreads. Néanmoins, nous devions *simuler* une exécution parallèle et non pas les exécuter en parallèle. Nous avions donc besoin de court-circuiter l'ordonnanceur du système pour le substituer par le notre et effectuer manuellement les changements de contexte. Ceci s'est fait simplement à l'aide de variables de synchronisations. À chaque instant un seul thread s'exécute et le changement de contexte s'effectue grâce aux fonctions de traitement des tâches.

À chaque instant, on a donc une liste d'agents prêts, c'est-à-dire à qui on doit passer successivement la main. Supposons par exemple que l'un de ces agent souhaite exécuter une tâche. Il va donc appeler la fonction correspondante (`MSG_execute`). Cette fonction va ordonnancer l'exécution de la tâche en question sur la ressource correspondant à l'hôte de l'agent. Elle va ensuite faire passer l'agent dans la liste des agents bloqués et passer la main au prochain agent prêt. Une fois que tous les agents sont bloqués, on peut commencer l'intégration des traces et chercher la première tâche qui se termine (ou «les» s'il y en a plusieurs qui se terminent en même temps). On fait alors passer chacun des agents concernés dans la liste des agents prêts et on recommence.

La notion de threads dont nous avons besoin pour simuler l'exécution parallèle de nos agents est donc beaucoup plus «faible» que celle offerte par les pthreads. Les changements de contexte s'effectuant uniquement par le biais des fonctions de la bibliothèque (qui sont toutes bloquantes), l'utilisation de pthreads qui sont des processus légers de niveau noyau entraîne un gaspillage de ressources systèmes inutile. Même des processus légers de niveau utilisateur comme ceux fournis par PM^2 offrent des fonctionnalités inutiles à notre situation. En fait, l'abstraction adaptée à ce type de situation est celle des contextes d'exécution proposée originellement dans l'API Système V et actuellement disponible dans la bibliothèque C de GNU.

SimGrid propose donc deux modes de simulation. L'une repose sur les pthreads qui sont très répandus et l'autre repose sur les contextes de la bibliothèque C de GNU. Dans un cas comme dans l'autre, il est possible (si le système le permet) de créer autant d'agents que l'on souhaite. Les abstractions fournies dans SimGrid permettent de programmer par passage de message (c'est le style de base), par appel de procédure à distance et même d'effectuer très simplement de la migration de processus.

7.4.4 Un petit exemple

Afin de mieux cerner l'intérêt d'un simulateur comme SimGrid, nous allons modéliser un ordonnancement simple sur une étoile. Comme nous l'avons expliqué en section 7.4.2, la première étape d'une simulation consiste à définir le code des différents agents. Dans cet ordonnancement, le maître va simplement envoyer un nombre fixe de tâches (`NB_TASK`) à chaque esclave. Pour s'exécuter convenablement, le maître a donc besoin de la liste des machines qu'il peut utiliser comme esclave. Cette liste sera donc passée en argument sur sa ligne de commande.

```
1  int master(int argc, char *argv[])
2  {
3    int slaves_count = 0;    /* le nombre d'esclaves  */
4    m_host_t *slaves = NULL; /* la liste des esclaves */
5    int todo_count = 0;      /* le nombre de tâches à effectuer */
6    m_task_t *todo = NULL;   /* la liste des tâches   */
7
8    int i;
9
10   {                  /* Organisation logique de la plate-forme */
11     slaves_count = argc - 1;
12     slaves = calloc(slaves_count, sizeof(m_host_t));
13
14     for (i = 1; i < argc; i++) {
15       /* on récupère l'hôte correspondant en utilisant son nom */
16       slaves[i-1] = MSG_get_host_by_name(argv[i]);
17       if(slaves[i-1]==NULL) {
18         PRINT_MESSAGE("Unknown host %s. Stopping Now! \n", argv[i]);
19         abort();
20       }
21     }
22     /* À partir d'ici, le maître connaît tous ses esclaves  */
23     /* et pourrait leur envoyer du travail s'il en avait... */
24   }
25
26   {                  /*  Création des tâches */
27     char sprintf_buffer[64];
28     int slave  = slaves_count;
29
30     todo = calloc(NB_TASK * slave, sizeof(m_task_t));
31     todo_count = NB_TASK * slave;
32
33     for (i = 0; i < NB_TASK * slave; i++) {
34       sprintf(sprintf_buffer, "Task_%d", i);
35       /* On crée une tâche Task_i équivalente à 5000 MFlop de calcul */
36       /* et constituée de 10 Mo de fichiers. */
37       todo[i] = MSG_task_create(sprintf_buffer, 5000, 10, NULL);
```

```
38        }
39  o     /* À partir d'ici, le maître a du travail à distribuer  */                o
40        /* à ses esclaves.                                      */
41  o   }                                                                           o
42
43  o   /* Il nous tient au courant de ce qu'il sait...          */                o
44      PRINT_MESSAGE("Got %d slave(s) :\n", slaves_count);
45  o   for (i = 0; i < slaves_count; i++) PRINT_MESSAGE("\t %s\n", slaves[i]->name); o
46      PRINT_MESSAGE("Got %d task to process :\n", todo_count);
47  o   for (i = 0; i < todo_count; i++) PRINT_MESSAGE("\t\"%s\"\n", todo[i]->name);  o
48
49  o   /* ... puis distribue ses tâches cycliquement à chacun de */                o
50      /* ses esclaves.                                        */
51  o   for (i = 0; i < todo_count; i++) {                                          o
52
53  o     PRINT_MESSAGE("Sending \"%s\" to \"%s\"\n", todo[i]->name,               o
54                      slaves[i % slaves_count]->name);
55  o     /* On envoie la i-ème tâche sur le canal PORT_22 du     */                o
56        /* (i % slaves_count) processeur.                       */
57  o     MSG_task_put(todo[i], slaves[i % slaves_count], PORT_22);                 o
58        PRINT_MESSAGE("Send completed\n");
59  o   }                                                                           o
60
61  o   free(slaves);                                                             o
62      free(todo);
63  o   return 0;                                                                 o
64    }
```

Les esclaves se contentent d'attendre des tâches et de les exécuter immédiatement. Leur code est encore plus simple que le précédent :

```
1   o  int slave(int argc, char *argv[])                                          o
2      {
3   o    int todo_count = 0;                                                      o
4        m_task_t *todo = calloc(NB_TASK, sizeof(m_task_t));
5   o    int i;                                                                   o
6
7   o    for (i = 0; i < NB_TASK;) {                                              o
8          PRINT_MESSAGE("Awaiting Task %d \n", i);
9   o      /* On attend la i-ème tâche sur le canal PORT_22 */                     o
10         MSG_task_get(&(todo[i]), PORT_22);
11  o      todo_count++;                                                          o
12         PRINT_MESSAGE("Received \"%s\" \n", todo[i]->name);
13  o      PRINT_MESSAGE("Processing \"%s\" \n", todo[i]->name);                  o
14         /* Une fois reçue, on l'exécute... */
15  o      MSG_task_execute(todo[i]);                                             o
```

```
16        PRINT_MESSAGE("\"%s\" done \n", todo[i]->name);
17        sched_data_task_destroy(MSG_task_get_data(todo[i]));
18        /* ... puis on fait le ménage. */
19        MSG_task_destroy(todo[i]);
20        i++;
21      }
22      free(todo);
23      return 0;
24    }
```

Une fois le code des deux différents types d'agents définis, il ne nous reste plus qu'à créer une plate-forme, déployer les agents sur les différents endroits et lancer la simulation.

```
1   void test_all(char *platform_file,char *application_file)
2   {
3     {                              /*  Configuration de la simulation */
4       MSG_global_init();                      /* Initialisation du simulateur   */
5       MSG_set_verbosity(MSG_SILENT);          /* Paramètre de debug             */
6       MSG_set_channel_number(MAX_CHANNEL);    /* Définition du nombre de canaux */
7       MSG_create_environment(platform_file);  /* Création de la plate-forme     */
8     }
9     {                             /*   Déploiement de l'application   */
10      MSG_function_register("master", master);  /* Désignation des différentes */
11      MSG_function_register("slave", slave);    /* fonctions.                  */
12      MSG_launch_application(application_file); /* Déploiement                 */
13    }
14
15    MSG_main();   /* Lancement de la simulation */
16    MSG_clean();  /* Suppression des données internes du simulateur */
17  }
```

La description de la plate-forme est donc complètement externe au programme et indépendante du reste de la simulation. Nous expliquons en section 7.5 comment créer automatiquement des fichiers de description réalistes. Pour ce qui est du déploiement, la description est également externe. Il est donc aisé de tester un ordonnancement sur une grande variété d'environnements puisque ce sont les deux seuls paramètres de la simulation.

Nous présentons brièvement le résultat de l'exécution de cet ordonnancement simpliste sur la plate-forme de la figure 7.3 avec le fichier de déploiement suivant :

```
the-doors.ens-lyon.fr  master moby.ens-lyon.fr canaria.ens-lyon.fr popc.ens-lyon.fr
moby.ens-lyon.fr slave
canaria.ens-lyon.fr slave
popc.ens-lyon.fr slave
```

Ainsi, un agent va être créé sur la machine *the-doors.ens-lyon.fr*, va exécuter la fonction enregistrée avec le nom `master` et aura comme arguments de ligne de commande `moby.ens-lyon.fr canaria.ens-lyon.fr popc.ens-lyon.fr`. Sur chacune des autres machines (*moby.ens-lyon.fr*, *canaria.ens-lyon.fr* et *popc.ens-lyon.fr*), un agent exécutant la fonction enregistrée sous le nom `slave` sera créé. On obtient alors la sortie suivante :

```
moby:~/simgrid2/examples/msg $ ./masterslave3 platform.xml deployment.txt
Tester the-doors.ens-lyon.fr for domain ens-lyon.fr found
Tester popc0.popc.private for domain popc.private found
Sites merged...
Building route table...
Route table built...launching simulation
[0.00] P1 | [master] Got 3 slave(s) :
[0.00] P1 | [master]       moby.ens-lyon.fr
[0.00] P1 | [master]       canaria.ens-lyon.fr
[0.00] P1 | [master]       popc.ens-lyon.fr
[0.00] P1 | [master] Got 9 task to process :
[0.00] P1 | [master]      "Task_0"
[0.00] P1 | [master]      "Task_1"
[0.00] P1 | [master]      "Task_2"
[0.00] P1 | [master]      "Task_3"
[0.00] P1 | [master]      "Task_4"
[0.00] P1 | [master]      "Task_5"
[0.00] P1 | [master]      "Task_6"
[0.00] P1 | [master]      "Task_7"
[0.00] P1 | [master]      "Task_8"
[0.00] P1 | [master] Sending "Task_0" to "moby.ens-lyon.fr"
[0.00] P2 | [slave] Awaiting Task 0
[0.00] P3 | [slave] Awaiting Task 0
[0.00] P4 | [slave] Awaiting Task 0
[1.98] P1 | [master] Send completed
[1.98] P1 | [master] Sending "Task_1" to "canaria.ens-lyon.fr"
[1.98] P2 | [slave] Received "Task_0"
[1.98] P2 | [slave] Processing "Task_0"
[3.97] P3 | [slave] Received "Task_1"
[3.97] P3 | [slave] Processing "Task_1"
[3.97] P1 | [master] Send completed
[3.97] P1 | [master] Sending "Task_2" to "popc.ens-lyon.fr"
[20.99] P4 | [slave] Received "Task_2"
[20.99] P4 | [slave] Processing "Task_2"
[20.99] P1 | [master] Send completed
[20.99] P1 | [master] Sending "Task_3" to "moby.ens-lyon.fr"
[45.67] P2 | [slave] "Task_0" done
[45.67] P2 | [slave] Awaiting Task 1
[47.65] P1 | [master] Send completed
[47.65] P1 | [master] Sending "Task_4" to "canaria.ens-lyon.fr"
[47.65] P2 | [slave] Received "Task_3"
```

```
[47.65] P2 | [slave] Processing "Task_3"
[91.34] P2 | [slave] "Task_3" done
[91.34] P2 | [slave] Awaiting Task 2
[149.60] P3 | [slave] "Task_1" done
[149.60] P3 | [slave] Awaiting Task 1
[151.58] P1 | [master] Send completed
[151.58] P1 | [master] Sending "Task_5" to "popc.ens-lyon.fr"
[151.58] P3 | [slave] Received "Task_4"
[151.58] P3 | [slave] Processing "Task_4"
[246.71] P4 | [slave] "Task_2" done
[246.71] P4 | [slave] Awaiting Task 1
[263.73] P1 | [master] Send completed
[263.73] P1 | [master] Sending "Task_6" to "moby.ens-lyon.fr"
[263.73] P4 | [slave] Received "Task_5"
[263.73] P4 | [slave] Processing "Task_5"
[265.71] P2 | [slave] Received "Task_6"
[265.71] P2 | [slave] Processing "Task_6"
[265.71] P1 | [master] Send completed
[265.71] P1 | [master] Sending "Task_7" to "canaria.ens-lyon.fr"
[297.21] P3 | [slave] "Task_4" done
[297.21] P3 | [slave] Awaiting Task 2
[299.20] P1 | [master] Send completed
[299.20] P1 | [master] Sending "Task_8" to "popc.ens-lyon.fr"
[299.20] P3 | [slave] Received "Task_7"
[299.20] P3 | [slave] Processing "Task_7"
[309.40] P2 | [slave] "Task_6" done
[444.83] P3 | [slave] "Task_7" done
[489.45] P4 | [slave] "Task_5" done
[489.45] P4 | [slave] Awaiting Task 2
[506.48] P1 | [master] Send completed
[506.48] P4 | [slave] Received "Task_8"
[506.48] P4 | [slave] Processing "Task_8"
[732.20] P4 | [slave] "Task_8" done
MSG: Congratulations ! Simulation terminated : all process are over
```

Pour plus d'informations sur les différentes possibilités offertes par ce simulateur, nous renvoyons à la documentation qui est disponible à l'adresse suivante :

http://www.ens-lyon.fr/~alegrand/simgrid2/doc/html/

La section suivante sera consacrée au problème de l'acquisition de paramètres permettant d'obtenir des plates-formes réalistes.

7.5 Observation de la plate-forme

Les travaux présentés dans cette section ont été menés en collaboraion avec Julien Lerouge et Martin Quinson.

7.5.1 Besoins pour une simulation

La création à la main des différentes ressources de calcul et de communication d'une plate-forme hétérogène est rapidement très fastidieuse. Il s'est rapidement révélé nécessaire de fournir un moyen automatique pour importer une plate-forme réaliste dans SimGrid. Pour ce faire, il fallait un outil capable de déterminer une topologie à peu près réaliste d'une plate-forme réelle, en incluant les *switchs*, les *hubs* et les routeurs, et les capacités des différentes ressources.

Nous avons utilisé deux outils que nous avons légèrement adaptés à nos besoins : *Effective Network View* et *Network Weather Service*. ENV nous permet de trouver déterminer la topologie d'un réseau telle qu'un hôte donné peut la percevoir et NWS nous permet de rassembler les traces dont nous avons besoin (latence, bande passante, charge du système) pour simuler la charge externe.

Dans la section 7.5.2, nous présentons un certain nombre d'outils permettant de découvrir une topologie et expliquons en quoi ils ne répondent pas à nos besoins.

7.5.2 Topologie de niveau 2 ou 3

Il y a plusieurs notions de topologie possibles selon le niveau auquel on se situe. Les deux principales sont celles du niveau 2 (*Liaison*) et du niveau 3 (*Réseau*) du modèle OSI/ISO. Les protocoles comme IP se situent au niveau 3 alors qu'Ethernet se situe au niveau 2. La topologie peut donc être très différente selon le niveau où l'on se place. Celle du niveau 2 est plus proche des liens physiques et est susceptible de fournir des informations sur les routeurs et les *switches* que l'on ne peut accéder directement par le niveau 3. Néanmoins, nous ne nous sommes pas basés sur la topologie de niveau 2 pour différentes raisons que nous allons exposer.

- La topologie de niveau 2 est souvent très difficile à découvrir car, étant plus proche des ressources physiques, la façon de récupérer les informations est souvent très dépendante du matériel, souvent mal documentée et parfois possible uniquement avec des outils propriétaires.
- Les outils permettant de découvrir la topologie de niveau 2 sont généralement basés sur le protocole SNMP (Simple Network Management Protocol [19]). Parmi les grand projets se basant sur cette approche, on retiendra principalement Remos [79, 46] qui utilise SNMP pour obtenir la topologie des réseaux locaux et de simples tests de bande passante pour obtenir des informations sur les WAN [85]. Cependant l'utilisation d'un tel protocole est la plupart du temps réservée à quelques utilisateurs chargés de l'administration, et ce principalement pour deux raisons : la

sécurité car il est alors possible d'effectuer des attaques de type *Deny Of Service* en inondant le réseau de requêtes, et la confidentialité car la plupart des fournisseurs d'accès ne souhaitent généralement pas exposer publiquement les points de contention possibles de leur installation.
En utilisant une topologie de niveau 2, on se place à un niveau «microscopique», ce qui est contradictoire avec l'approche de la simulation que nous avons choisie. Enfin, quand bien même nous réussirions à obtenir une carte précise de la topologie de niveau 2, nous ne souhaitons pas simuler l'ensemble des liens d'Internet, et il n'est pas possible de simuler uniquement la charge externe des quelques liens utilisés par une application sans tenir compte de la globalité du réseau.

7.5.2.1 `traceroute`, `patchchar` et leurs petits frères

Un certain nombre d'outils comme TopoMon [28] ou Lumeta [27] s'appuient essentiellement sur `traceroute` pour inférer la topologie de niveau 2. `traceroute` et `patchchar` ont été développés par Van Jacobson à 10 ans d'intervalle [68]. Tous deux utilisent le champ *Time-To-Live* des paquets IP qui détermine le nombre de relais qu'un paquet peut franchir avant d'être invalidé. Si `traceroute` permet de connaître la route empruntée par des paquets IP pour aller d'un point à un autre à un instant donné, `patchchar` permet de déterminer la capacité de chacun des liens traversés. En envoyant des rafales de paquets de différentes tailles, il est possible d'inférer statistiquement la latence et la bande passante des différents liens [48].

Une des grandes limitations de `patchchar` est que l'estimation des différents paramètres est faite en supposant que pour chaque lien, les paquets tests vont passer un temps négligeable dans les files d'attente des différents routeurs. Pour qu'une telle hypothèse soit réalisée, il est nécessaire d'effectuer un très grand nombre de tests, car sa probabilité est très faible. Dans les mises en œuvre actuelles de cette méthodologie, plus de 1500 tests sont généralement effectués et de tels tests sur des routes composées de plusieurs liens peuvent durer quelques heures. De plus, si `traceroute` ne nécessitait aucun droit particulier, `patchchar` a besoin d'ouvrir des sockets en mode *RAW* et demande donc des droits particuliers.

Enfin, de tels outils nous donnent la capacité de chaque lien, ce qui peut être une information intéressante mais non suffisante pour le type de simulation que nous visons puisqu'elle ne nous permet pas de déterminer la bande passante effective de bout en bout d'une connexion (sur un réseau tel que VTHD, la capacité est énorme mais elle est inatteignable avec une seule connexion).

7.5.2.2 Autre outils de niveau 3

On trouve un certain nombre de projets visant à donner des modélisations de la topologie d'Internet ou de son comportement. Nous en présentons quelques-uns et expliquons en quoi ils sont inapplicables à notre contexte ;

SPAND : *Shared Passive Network Performance Discovery* [97] est un outil permettant de déterminer la bande passante disponible et le taux de perte sur les routes conduisant à une machine donnée. Cet outil est passif, ce qui signifie que la machine en question a une pile réseau modifiée et qui collecte des statistiques. Un tel outil n'est pas utilisable dans notre situation car la plupart du temps, nous n'avons qu'un accès d'utilisateur non privilégié aux différentes machines de notre plate-forme. De plus, cet outil ne fournit aucune information sur la nature des liens utilisés.

IDMaps, GNP : *IDMaps* et *Global Network Positioning* [57, 87] ont pour objectif de déterminer la distance entre deux hôtes d'un réseau. La distance est cependant principalement basée sur la latence mesurée à l'aide de *pings* réguliers et ce type de modèle ne répond donc pas à nos besoins.

Mercator : Mercator [62] est un programme utilisant le même type de méthodologie que `traceroute` pour établir des cartes d'Internet. Il n'y a donc toujours pas d'information sur la capacité des liens et des routeurs.

7.5.3 Effective Network View

À la différence des approches basées sur SNMP, nous avons comme objectif, non pas de récupérer la topologie telle qu'elle est déclarée par les administrateurs du réseau mais telle qu'elle est *perçue* par les applications. De même, les informations obtenues par `traceroute` sont de niveau 3 alors que les applications sont de niveau supérieur. La réalité capturée par ces programmes peut donc être très différente de celle vécue par les applications. Souhaitant reproduire un environnement ayant le même type de comportement que celui que les applications peuvent percevoir, nous nous sommes intéressés à un outil proposant une approche de la mesure de la topologie de niveau supérieur : ENV (*Effective Network View*).

7.5.3.1 Présentation

ENV [100] a été développé par Shao, Berman et Wolsky à l'Université de Californie, San Diego. Cet outil a pour fonction de découvrir la topologie d'un réseau telle que peut la percevoir une machine (le maître dans le cas d'une application maître/esclave). Les données ainsi obtenues dépendent donc du choix du maître. Cet outil est original car il n'utilise que des techniques d'observation du réseau de niveau utilisateur et infère des informations de niveau 2 ou 3 sans utiliser de fonctions bas-niveau. La figure 7.3 représente une partie du réseau de l'ENS Lyon. Cette figure est simplifiée car en pratique, certaines routes ne sont pas symétriques et un VLAN est utilisé. La figure 7.4 montre la vue que la machine *the-doors* a du réseau selon ENV.

Ces deux figures nous permettent de voir qu'ENV simplifie la réalité et ne détecte pas tous les routeurs. Seuls *routlhpc* et *routeur_backbone* apparaissent sur la topologie générée avec ENV. Ceci est due à l'asymétrie de certaines routes et à des erreurs de routage dans des tables statiques. Par contre, ENV détecte bien que les machines *popc0*, *myri0* et

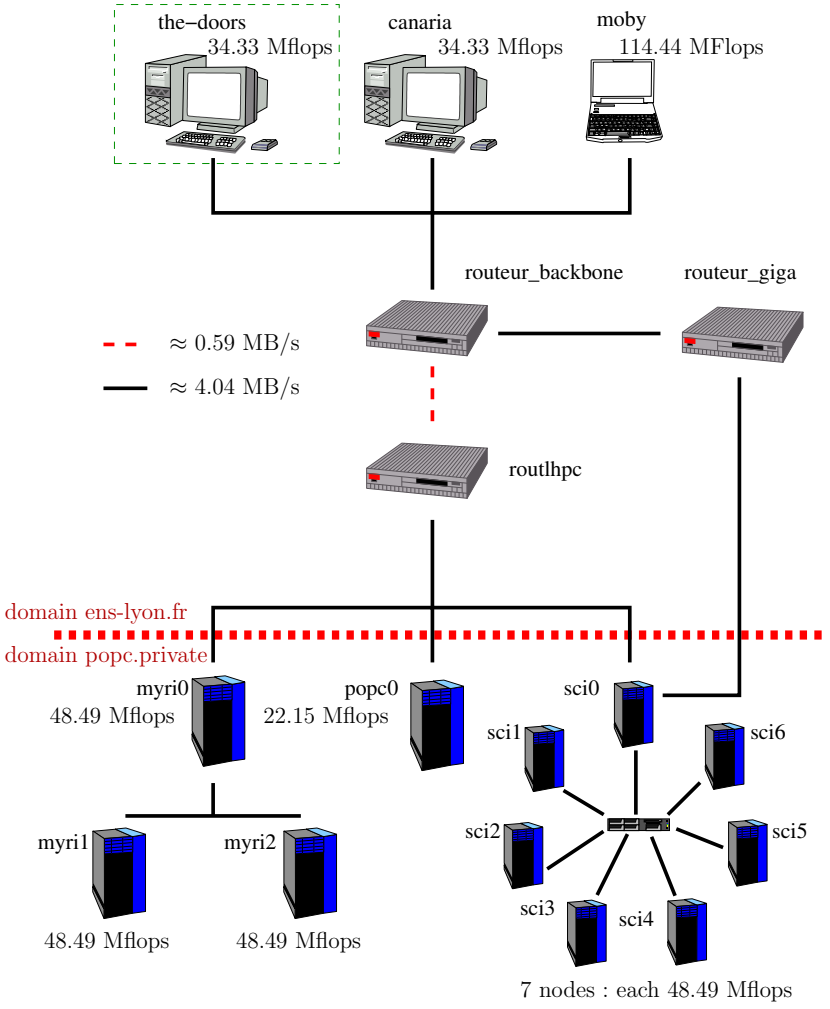


FIG. 7.3 – Topologie physique de notre réseau test à l'ENS.

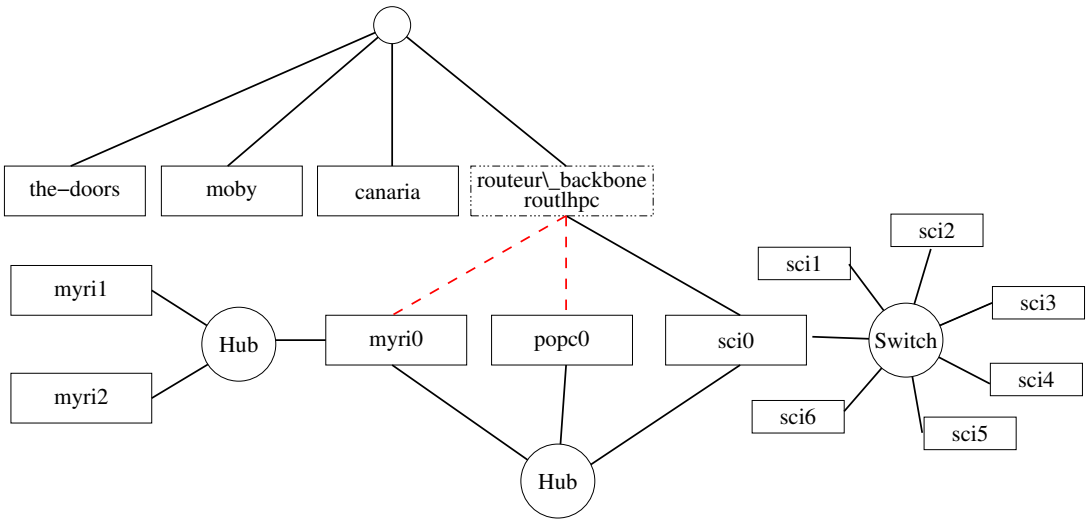


FIG. 7.4 – Topologie effective calculée par ENV avec *the-doors* comme machine maître.

sci0 sont sur des réseaux *switchés* à 100 Mb/s alors que pour atteindre *popc0* et *myri0*, *the-doors* traverse un goulot d'étranglement à 10 Mb/s. Ce type d'information est crucial pour les différents algorithmes d'ordonnancement tels que ceux que nous avons présentés dans les chapitres précédents.

7.5.3.2 Principes

L'acquisition des données par ENV se décompose en deux parties. La première est indépendante du choix du maître à la différence de la seconde.

La première phase collecte des informations sur les différentes machines telles que leur adresse IP, le type de processeur, le système d'exploitation, le nombre de processeurs,etc. Nous avons rajouté une mesure de la puissance du processeur à l'aide d'une fonction d'étalonnage fournie dans LINPACK [26]. Un `traceroute` vers une machine externe au réseau est également effectué, de façon à déterminer une première topologie structurelle et un premier arbre d'interconnexion. Dans le cas de la plate-forme de Lyon, on obtient l'arbre de la figure 7.5. Cet arbre n'est pas suffisant car il ne montre pas clairement la nature des différents liens. Un certain nombre de tests vont être effectués pour raffiner cette vue.

La seconde phase dépend du maître. Un nouvel arbre est généré en effectuant une série de tests :

Bande passante vers chaque hôte : la bande passante entre le maître et chacune des machines de la plate-forme est mesurée en effectuant un simple transfert. Partant du principe qu'*a priori*, si deux machines sont proches l'une de l'autre elles doivent avoir la même bande-passante vers le maître, les groupes de machines sont divisés si leurs bandes-passantes ne sont pas assez proches.

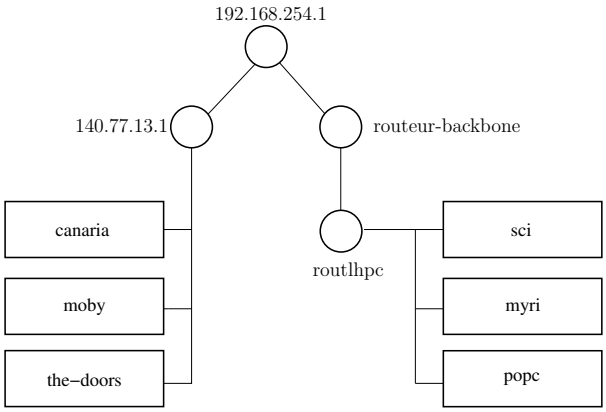
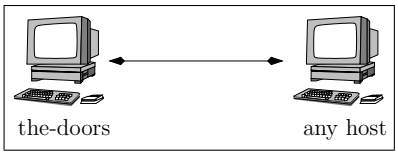
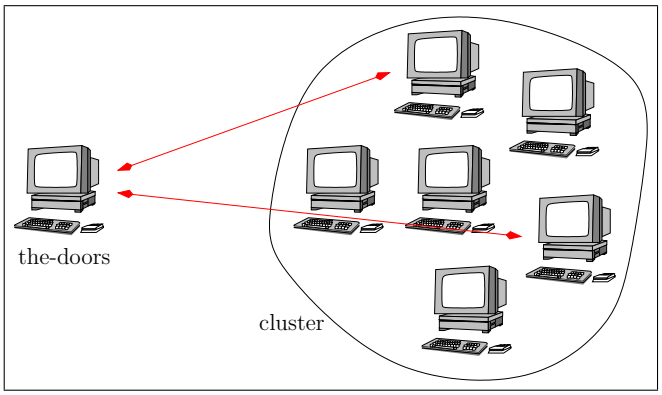


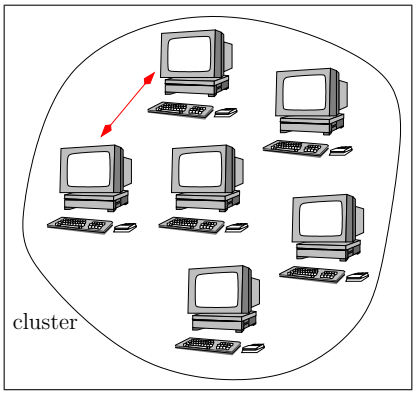
FIG. 7.5 – Arbre initial d'ENV



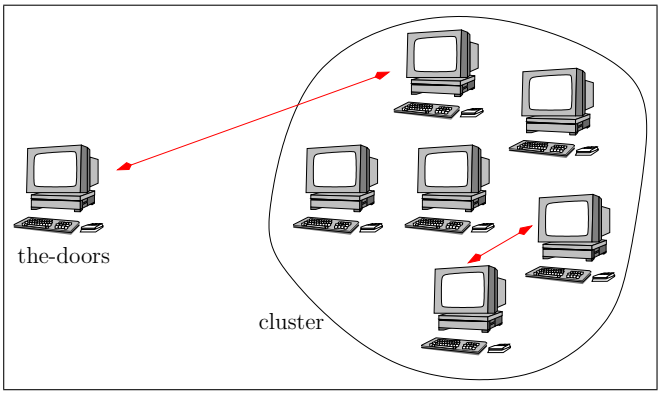
(a) Bande passante vers chaque hôte



(b) Bande passante par paire d'hôtes



(c) Bande passante interne



(d) Interférence avec l'extérieur

Bande passante par paire d'hôtes : pour chaque groupe ainsi dégagé, on mesure la bande passante entre le maître et toute paire de machines du groupe. Les deux mesures sont faites en même temps, de façon à détecter une interférence. Les groupes sont alors à nouveau divisés en groupe d'hôtes partageant le même type de caractéristiques : si le ratio entre la bande passante entre le maître et un hôte et la bande passante mesurée en parallèle avec un autre transfert est inférieur à une certaine valeur, l'hôte est déclaré indépendant.

Bande passante interne : pour chaque groupe ainsi dégagé, la bande passante est mesurée entre chaque paire de machines du groupe. Cela permet d'obtenir la bande passante au sein d'un groupe de machines car la bande passante interne n'a pas de raison d'être égale à celle avec mesurée entre le maître et les différentes machines du groupe. Par exemple, le groupe contenant *popc* dans notre exemple a une bande passante de 100 Mps en interne alors qu'il faut passer par un goulot d'étranglement pour l'atteindre.

Interférence avec l'extérieur : dans chaque groupe, la bande passante entre le maître et une machine est mesurée alors qu'un transfert entre une paire d'autres machines du groupe est effectué. Cette mesure est effectuée 5 fois et le rapport entre la bande passante mesurée initialement et les nouvelles est calculé. La moyenne des ces rapports est comparée à deux seuils *shared_threshold* et *indep_threshold* :

- si cette moyenne est inférieure à *shared_threshold* (fixé empiriquement à 0,7) le lien reliant les différentes machines du groupe est partagé (comme pour un hub) et les communications internes interfèrent entre elles,
- si cette moyenne est supérieure à *indep_threshold* (fixé empiriquement à 0,9) alors les communications internes sont indépendantes,
- dans les autres cas, le type du réseau connectant ce groupe est indéterminé.

7.5.3.3 Limitations

Outre un certain nombre de problèmes techniques qui se posent lors de la mise en œuvre d'un tel outil et qui sont détaillés dans [75], un certain nombre d'autres problèmes plus fondamentaux se pose.

Pertinence : la question de savoir si les résultats donnés par ENV sont précis ou non n'est pas vraiment importante. Nous n'en sommes pas arrivés à utiliser ENV pour sa précision mais parce qu'il permet de mesurer simplement une topologie du niveau qui nous intéresse. En ce qui nous concerne, les différentes fois où nous l'avons utilisé, les résultats étaient tout à fait pertinents. En revanche, on peut discuter de la méthodologie employée. Le problème principal réside dans la durée de validité des différentes mesures. S'il y a un grand nombre de mesures à effectuer et que la charge de la plate-forme évolue entre les différentes séries de tests, les résultats d'ENV peuvent être faussés. Il n'y a pas vraiment de solution à ce problème à part se dépêcher d'effectuer les différentes mesures. Sur une plate-forme comme celle de Lyon, 5 minutes sont nécessaires pour effectuer l'ensemble des mesures et produire

une topologie cohérente. Sur une si petite période de temps, il n'est pas absurde de supposer que l'environnement est suffisamment stable pour produire des résultats pertinents. Rien n'est moins évident sur une grosse plate-forme. Néanmoins, toute connaissance de non-interférence des différentes parties d'une plate-forme, peut être utilisée pour diminuer efficacement la durée des expériences : on peut supposer sans trop de risque que dans une plate-forme composée de machines appartenant à deux universités, on peut traiter chacune des deux universités indépendamment l'une de l'autre.

Paradigme Maître/Esclave : ENV a été conçu en ayant à l'esprit d'utiliser les topologies ainsi créées pour faire de l'ordonnancement maître/esclave. C'est la raison pour laquelle une des machines joue un rôle particulier. Cela permet de diminuer considérablement le nombre de tests à effectuer mais cela limite la vision de la plate-forme. Les topologies ainsi créées sont donc toujours très arborescentes et certains liens entre différents groupes de machines peuvent ne pas apparaître du tout.

7.5.4 Network Weather Service

Une fois obtenue une topologie simple et susceptible de nous permettre de reconstruire un environnement tel que celui que peuvent percevoir nos applications, il nous reste à instancier la charge externe des différentes ressources de la plate-forme. Nous avons utilisé NWS [116] pour collecter ces données. L'objectif de ce projet est de fournir des données précises sur les performances au cours du temps d'une plate-forme distribuée. NWS ne permet pas seulement de collecter la charge des différentes ressources mais également de prédire leur valeur dans un futur proche à l'aide d'une batterie de méthodes statistiques [115]. Cette partie n'a pas d'intérêt pour nous car nous souhaitons seulement enregistrer ces valeurs pour les réutiliser par la suite.

7.5.4.1 Organisation de NWS

NWS est composé d'un certain nombre d'entités ayant différents rôles :

- entrepôt (*Persistent State*) : ce processus récupère et stocke l'ensemble des données produites par les autres processus ;
- serveur de nom (*Name Server*) : ce processus permet de connaître les différents processus disponibles sur un hôte donné ;
- senseur (*Sensor*) : ce type de processus mesure les performance d'une ressource ;
- prédicteur (*Forecaster*) : ce type de processus effectue de la prédiction de performance en sélectionnant dynamiquement la méthode statistique donnant les meilleurs résultats.

Le déploiement des différents éléments n'est pas automatique et c'est donc généralement une tâche relativement longue et fastidieuse. Comme au moment de déployer NWS, ENV a déjà été exécuté, nous connaissons la topologie de la plate-forme et nous pouvons l'utiliser pour déployer intelligemment NWS.

7.5.4.2 Observation du réseau

Le mesure de la bande passante et de la latence entre n machines peut paraître simple au premier abord. La solution la plus simple consiste à effectuer $n(n-1)/2$ mesures en supposant que les connections sont symétriques. Cependant, si deux hôtes partagent un même lien physique, les mesures ne doivent pas être effectuées en même temps sous peine d'être faussées. De plus, quand n devient grand, le nombre de mesures à effectuer entraîne une perte de bande-passante non négligeable. Les mesures doivent donc être faites au bon moment et au bon endroit. NWS résout ces problèmes en utilisant des *cliques*. Une clique est un sous-ensemble de k ordinateurs pour lequel les $k(k-1)/2$ mesures sont effectuées. L'ensemble des machines est donc recouvert par une arborescence de cliques. Un hôte peut appartenir à autant de cliques que nécessaire mais il y a toujours au plus un hôte dans l'intersection de deux cliques distinctes. La bande passante entre deux hôtes appartenant à la même clique est effectivement mesurée. En revanche, la bande passante entre deux hôtes n'appartenant pas à la même clique doit être calculée à l'aide des mesures des autres cliques. La bande passante entre deux machines A et B peut par exemple être approchée en prenant le minimum des bandes passantes entre les machines apparaissant sur la route (dans l'arborescence de cliques) pour aller de A à B.

Les mesures ne sont donc pas faites en même temps. Un jeton passe d'hôte en hôte dans chaque clique, donnant la permission de faire des mesures [116]. La définition des cliques est cruciale pour obtenir des valeurs pertinentes. L'arborescence de cliques doit être aussi proche de la topologie réelle que possible. Notre déploiement de NWS est donc effectué automatiquement à partir de la topologie donnée par ENV.

Ainsi, pour chaque groupe/sous-groupe trouvé par ENV, nous construisons au moins deux cliques :

- Si le groupe est *partagé* (c'est-à-dire que toutes les communications interfèrent), nous supposons que toutes les communications s'effectuent sur le même lien physique (voir figure 7.3) et donc que la mesure de la latence et de la bande passante entre une paire d'hôtes de ce groupe est suffisante pour connaître pour celle des autres machines. Nous choisissons donc deux hôtes dans ce groupe en faisant une simple clique. On choisit également un autre hôte pour construire une clique entre ce groupe et le groupe parent s'il existe.
- Si le groupe est *switché* (c'est-à-dire qu'aucune des communications n'interfère), nous construisons une clique contenant l'ensemble des hôtes de ce groupe et une autre pour connecter la passerelle de ce groupe au groupe parent s'il existe.

Dans le cas de la plate-forme de l'ENS, on obtient le déploiement représenté sur la figure 7.6.

7.5.4.3 Observation des processeurs

L'observation des capacités d'un processeur sur un système UNIX est bien plus difficile que celle du réseau. En effet, les variations peuvent être bien plus rapides et bien plus importantes car il n'y a pas autant de trafic que sur un réseau et que la loi des grands

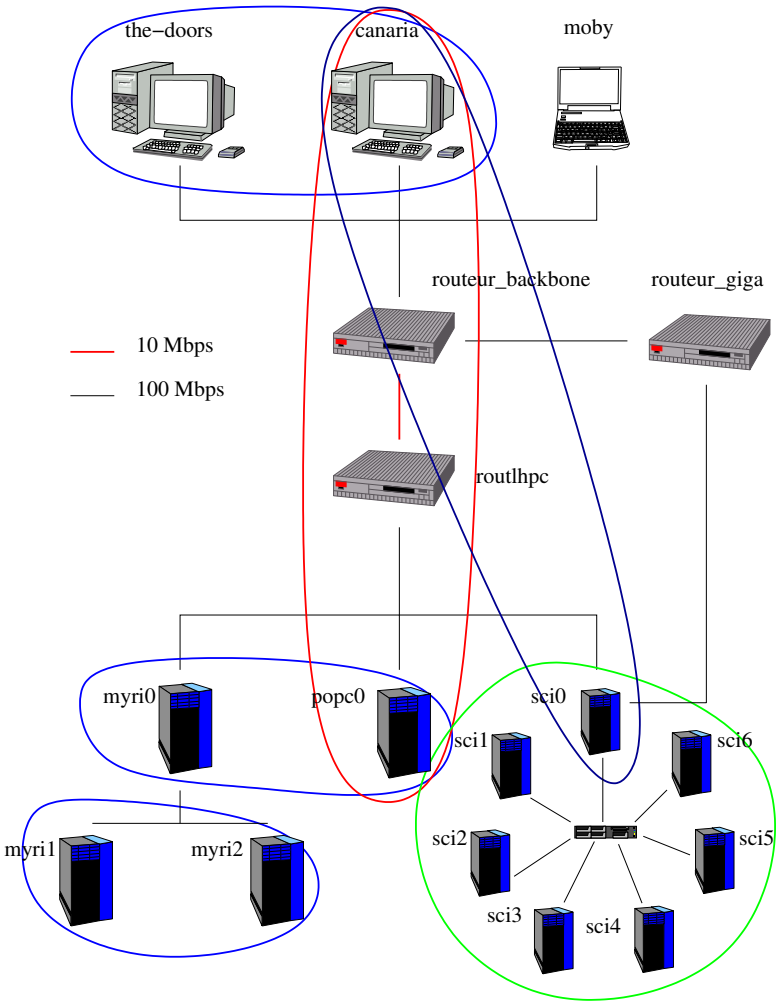


FIG. 7.6 – Définition des cliques NWS sur le réseau test de l'ENS.

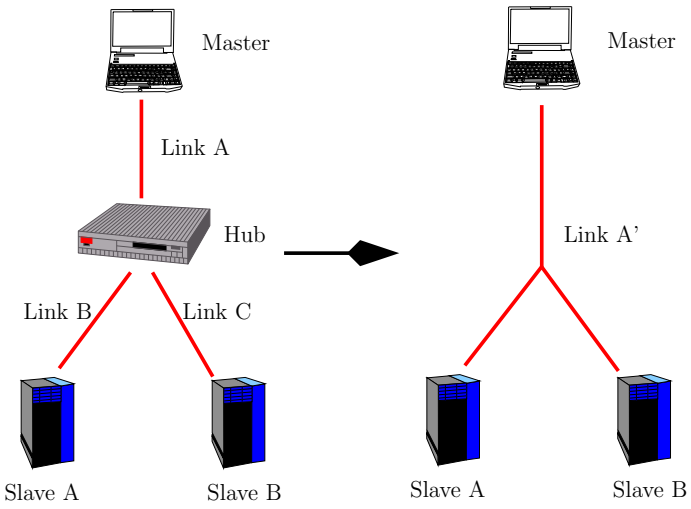


FIG. 7.7 – Modélisation d'un *hub* dans SIMGRID

nombres ne s'appliquant pas, ce type de ressource est plus difficile à modéliser et donc à prévoir. La charge dépend des personnes utilisant la machine, des démons exécutés sur la machine, etc. Elle peut donc sembler relativement chaotique au premier abord. Il est donc nécessaire d'effectuer des mesures plus fréquemment mais elles doivent être les plus légères et discrètes possibles pour ne pas perturber la ressource. Le senseur NWS chargé de l'observation des processeurs est relativement précis (au pire 10% d'erreur [117]). Cette précision est suffisante pour nos besoins car notre objectif n'est pas de prédire le temps que prendra une application mais de pouvoir obtenir un environnement d'exécution relativement réaliste pour comparer deux algorithmes.

7.5.5 Intégration dans SimGrid

Dans cette section, nous expliquons comment nous utilisons les différentes mesures collectée avec ENV et NWS pour créer un ensemble de ressources modélisant un environnement réaliste.

Nous appelons *hub*, *switch* et *routeur* et les routeurs trouvés respectivement au niveau 1, 2 et 3 du modèle OSI. Ces routeurs sont plus ou moins bien détectés par ENV :

Hub : ce sont les groupes de machines dont le lien les connectant est partagé, c'est-à-dire les groupes de machines où les communications internes interfèrent toutes entre elles. Dans ce cas, le *hub* est modélisé par une simple ressource de communication partagée par les différents hôtes du groupe (voir figure 7.7).

Switch : un groupe *switché* est un groupe de machines telle que les communications internes n'interfèrent pas entre elles. Le *switch* est alors modélisé par un graphe complet (voir figure 7.8). Ce modèle n'est pas parfait car par exemple, si le lien A a une capacité de 100 Mb/s, dans ce modèle, il est possible de transférer avec une

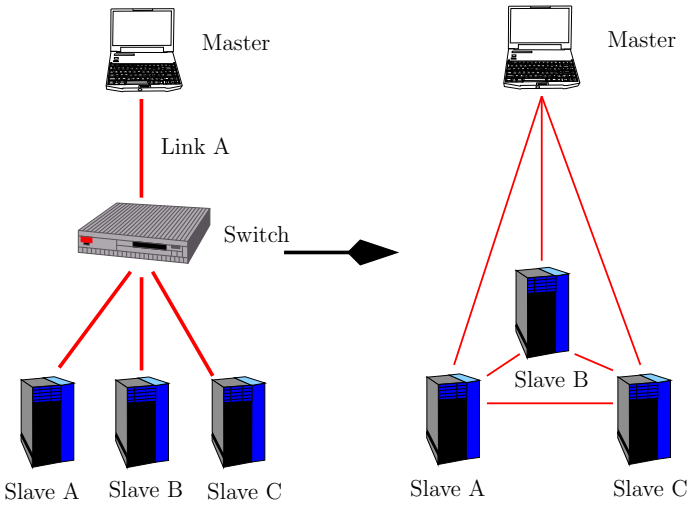


FIG. 7.8 – Modélisation d'un *switch* dans SIMGRID

bande passante de 200 Mb/s en utilisant les liens A' et C'. Pour avoir une bonne modélisation, il faudrait être capable de mesurer la bande passante entre chaque machine et le *switch*, ce qui n'est pas possible.

Routeur : il arrive qu'ENV n'arrive pas à déterminer le type du réseau connectant un groupe donné. On distingue alors deux modélisations selon que la racine du groupe est un hôte sur lequel ENV s'exécute ou pas :

- Prenons le cas de *myri0*, *popc0*, et *routlhpc*. ENV n'étant pas exécuté sur *routlhpc*, nous l'enlevons tout simplement. Le résultat d'une telle transformation est donnée en figure 7.9. Nous pourrions introduire un hôte modélisant le routeur et se chargeant de faire passer les paquets, mais il aurait fallu pouvoir mesurer la bande passante entre ce routeur et les autres machines.
- Prenons le cas du groupe constitué de *myri0*, *myri1*, et *myri2*. Le nombre de machines n'était pas suffisant pour statuer sur le type de réseau les connectant. *myri0* servant de routeur pour cette grappe, on utilise le même type de modélisation que dans le cas des hubs. Dans le cas contraire, on utilise la modélisation de la figure 7.10.

7.6 Conclusion

Dans ce chapitre, nous avons présenté quelques réflexions sur le thème de la modélisation de plates-formes distribuées. Nous avons présenté SIMGRID, un simulateur permettant de simuler simplement des algorithmes d'ordonnancement centralisés ou non, et une technique permettant d'obtenir des plates-formes réalistes utilisables par ce simulateur.

Actuellement, SIMGRID réprésente environ 15 000 lignes de code C, se compile et fonctionne sans difficulté sur à peu près n'importe quel système UNIX (linux, solaris, MacOS

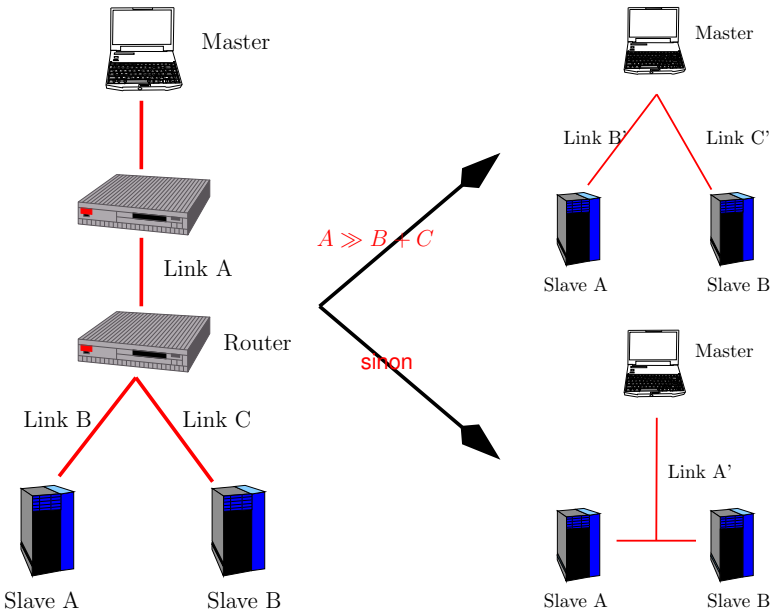


FIG. 7.9 – Modélisation d'un routeur dans SIMGRID

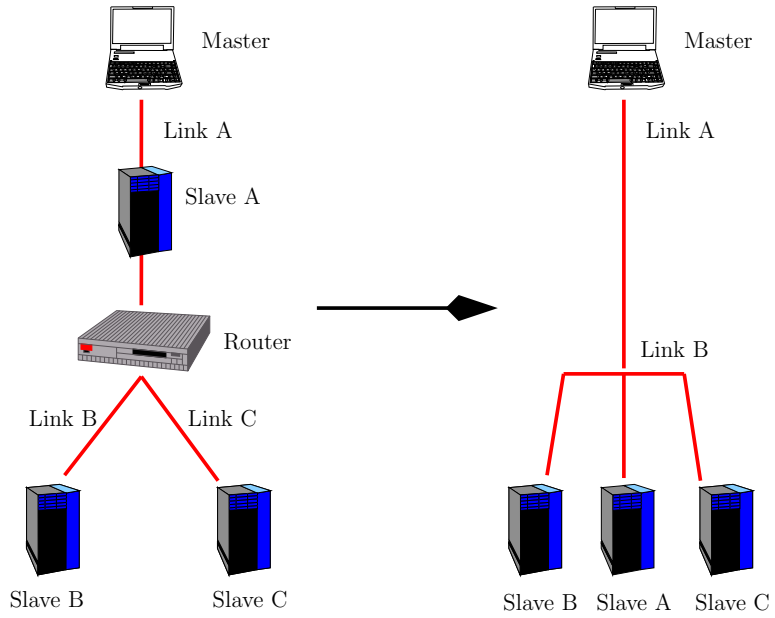


FIG. 7.10 – Modélisation d'un routeur dans SIMGRID

X, AIX, IRIX). Il est relativement simple d'utilisation et est librement téléchargeable. Il est utilisé à notre connaissance par plus d'une trentaine de chercheurs et a également été utilisé dans le cadre du cours d'algorithmique parallèle du Magistère d'Informatique et de Modélisation de l'ENS Lyon.

Aucun des problèmes présentés dans ce chapitre n'est complètement résolu. La simulation reste approximative mais nous pensons que cela est suffisant pour nos besoins. Un de nos objectifs est de proposer des jeux de description de plates-formes et ainsi permettre de comparer *simplement* et *équitablement* des algorithmes d'ordonnancement en leur faisant passer une batterie de tests. La génération de plates-formes réalistes peut se faire de deux façons. Soit en les générant aléatoirement, soit en filmant des plates-formes réelles. Les plates-formes que nous avons pu obtenir avec la première approche n'ont pas conduit à des comportements proches de ceux que nous pouvions observer dans la réalité à la différence de celles obtenues avec la seconde. Nous pensons donc que la seconde approche est la bonne. Elle nous a fait découvrir un certain nombre de problèmes intéressants tels que la découverte de topologies utilisables par des algorithmes d'ordonnancement ou la mesure des différentes caractéristiques des composants de la plate-forme.

Chapitre 8

Conclusion

Les différents chapitres de cette thèse ont permis de mesurer les difficultés algorithmiques liées à l'introduction de l'hétérogénéité de la plate-forme dans les modèles classiques. Des problèmes aussi simples que les noyaux d'algèbre linéaire denses ou la distribution de tâches indépendantes requièrent la mise en œuvre d'algorithmes sophistiqués. Les problèmes d'ordonnancement étaient déjà difficiles en environnement homogène, il n'y a aucune raison qu'ils se simplifient en environnement hétérogène. Néanmoins, nous espérons que les techniques développées dans ces différents chapitres ouvriront la voie à de nouvelles façons d'appréhender la conception efficace d'applications parallèles destinées à de tels environnements.

Nos principales contributions sont les suivantes :

Noyaux d'algèbre linéaire dense : nous avons proposé des distributions équivalentes aux distributions cycliques par bloc bidimensionnelles mais adaptées aux plates-formes hétérogènes. Nous avons également proposé des mécanismes de redistributions légères permettant de rééquilibrer optimalement la charge en cours de calcul. Ces résultats sont présentés en section 2.3 et 2.4.

Ordonnancement maître/esclave : nous avons proposé en section 3.3 un algorithme polynomial pour déterminer l'allocation optimale de tâches indépendantes de caractéristiques identiques sur une plate-forme de type bus lorsqu'une seule communication est nécessaire avant le traitement des tâches sur les différents processeurs. Ce problème, initialement formulé par Andonie, Chronopoulos, Grosu et Galmeanu [4], est ensuite étendu au cas où des communications avant et après le traitement de l'ensemble des tâches sont nécessaires. Nous avons montré que ce problème était NP-complet en section 3.4.1 et nous avons proposé en section 3.4.2 une heuristique garantie pour le résoudre.

Nous nous sommes intéressés au problème de l'allocation optimale de tâches indépendantes de caractéristiques identiques lorsqu'une communication est nécessaire avant le traitement de chacune des tâches sur les différents processeurs. Nous montrons en section 3.5 que ce problème peut être résolu en temps polynomial si la plate-forme est une étoile.

Ordonnancement en régime permanent : les difficultés rencontrées pour résoudre les problèmes précédents étant en grande partie liées à la métrique utilisée (le *makespan*), nous avons changé quelque peu les règles du jeu et nous nous sommes intéressés au calcul de la puissance de la plate-forme en régime permanent.

Le problème de l'allocation optimale de tâches indépendantes de caractéristiques identiques lorsqu'une communication est nécessaire avant le traitement de chacune des tâches sur les différents processeurs est alors résoluble beaucoup plus facilement et conduit à des ordonnancements asymptotiquement optimaux. Ces travaux sont présentés dans le chapitre 4.

Nous avons ensuite expliqué dans le chapitre 5 comment étendre partiellement ces résultats au cas où les tâches recèlent du parallélisme interne. L'affinité des sous-tâches avec les différents processeurs est aisément prise en compte, même si l'introduction de dépendances complique notablement le problème.

Enfin, nous avons adapté les techniques précédentes au modèle des tâches divisibles. Notre approche nous a permis d'établir de nouveaux résultats d'optimalité pour les distributions en une seule tournée sur les étoiles et d'optimalité asymptotique pour les distributions en plusieurs tournées sur des plates-formes quelconques. Ces résultats sont présentés dans le chapitre 6.

Modélisation et simulation de plates-formes hétérogènes : afin d'évaluer honnêtement nos algorithmes sur des plates-formes hétérogènes, nous avons développé un simulateur, en collaboration avec Henri Casanova de l'Université de Californie, San Diego, adapté aux besoins des ordonnanceurs ou des algorithmiciens. Un effort tout particulier a porté sur la simplicité et la portabilité de cette application. Nous avons également proposé une méthode simple permettant d'observer une plate-forme et de simuler un comportement réaliste. Le chapitre 7 présente brièvement la problèmatique de la simulation, notre simulateur, et les difficultés posées par l'acquisition de paramètres réalistes.

Avec des environnements hétérogènes, il est nécessaire de désynchroniser les algorithmes, de concevoir des stratégies souples capables de remettre régulièrement en cause leurs décisions pour prendre en compte les modifications de leur environnement. Les démarches purement dynamiques ont déjà montré leur faiblesse dans un cadre hétérogène. Leur incapacité à anticiper la lenteur de certains processeurs entraîne souvent une synchronisation de l'ensemble des processeurs sur le processeur le plus lent, tout particulièrement lorsqu'il y a des dépendances (même régulières) entre les tâches. La bonne démarche, selon nous, pour concevoir une application adaptée à un environnement hétérogène consiste d'abord à ignorer l'instabilité de la plate-forme en trouvant un algorithme statique et à essayer de garantir son efficacité. Alors seulement, il est possible d'en déduire un algorithme dynamique capable de s'adapter aux variations de charges et exhibant les même bonnes propriétés d'optimalité que l'algorithme statique. Dans le cas où les performances de la plate-forme sont imprévisibles, il est de toutes façons impossible de garantir quoi que ce soit concernant les performances d'un algorithme. Il est donc important qu'il soit efficace dès que l'état de la plate-forme est stabilisé et que l'on soit en mesure de quantifier son efficacité.

On comprend bien que dans ces conditions des mesures classiques telles que le *makespan* ne sont pas adaptées à la conception de tels algorithmes. Par exemple, si la NP-complétude de l'allocation de tâches indépendantes sur un arbre est très intéressante d'un point de vue théorique, elle n'est pas significative en pratique. En effet, la métrique utilisée doit être suffisamment robuste pour pouvoir prendre en compte l'instabilité de la plate-forme. Du fait de son caractère continu, c'est le cas de l'optimisation du débit en régime permanent. De plus, nous avons vu que de nombreux problèmes sont beaucoup plus simples et qu'il est possible d'obtenir des algorithmes asymptotiquement optimaux et bien plus aisément adaptables à une plate-forme réelle.

Les difficultés que nous avons rencontrées lors de l'acquisition de traces pour pouvoir effectuer des simulations réalistes nous ont permis de mesurer l'étendue du fossé qui sépare les modèles servant de base à la majorité des algorithmes et la réalité. Sans même parler de l'instabilité latente des plates-formes, il est extrêmement difficile d'avoir une vision claire de l'organisation et des capacités de la plate-forme. Dans ces conditions, on peut mesurer la vanité de l'introduction de modèles de plus en plus complexes que l'on est dans l'incapacité d'instancier (comme les approches basées sur les signatures d'applications [80, 110]) ou d'un algorithme fournissant le meilleur ordonnancement, à la seconde près en se basant sur évaluation extrêmement précise de la capacité des ressources. Pour avoir une quelconque utilité, ces algorithmes doivent s'appuyer sur des modèles simples dont les paramètres sont effectivement mesurables, ce qui est (bien sûr !) le cas des modèles et des algorithmes présentés dans cette thèse.

Annexe A

Notations

Définition A.1 (argmin). *Étant donnée une fonction f de X dans $\mathbb{R}$, on note* $\operatorname{argmin}_{x\in X} f(x)$ *un élément $\tilde{x}$ de X tel que $f(\tilde{x}) = \min_{x\in X} f(x)$.*

Cet opérateur renvoie un *des éléments qui atteint le minimum.*

La fonction argmax *est définie de façon similaire.*

Définition A.2 ($[\![a, b]\!]$). *Pour a, b dans $\mathbb{Z}$, on définit $[\![a, b]\!]$ par*

$$[\![a, b]\!] = [a, b] \cap \mathbb{Z}$$

Définition A.3 ($\lfloor x \rfloor$ et $\lceil x \rceil$). *Pour x dans $\mathbb{R}$, on définit $\lfloor x \rfloor$ comme l'entier dans $\mathbb{Z}$ vérifiant*

$$\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1,$$

et $\lceil x \rceil$ comme l'entier dans $\mathbb{Z}$ vérifiant

$$\lceil x \rceil - 1 < x \leqslant \lceil x \rceil,$$

Les notations utilisées au cours de cette thèse ne sont pas forcément uniformes d'un chapitre à l'autre car elles suivent celles utilisées dans la littérature. Nous rappelons donc brièvement chapitre par chapitre les différentes notations utilisées.

Chapitre 2

- $t_c r^3$ représente le temps nécessaire au produit de deux blocs carrés de taille r. Il représente donc le temps de cycle d'un processeur. Lorsque le temps de cycle varie d'un processeur à l'autre, on utilise indifféremment la notation t_i ou $t_c^{(i)}$;
- τr^2 représente le temps nécessaire à la communication d'un bloc carré de taille r ;
- s_i représente la fraction (en terme de surface) de matrice dont le processeur P_i est responsable ;
- h_i représente la hauteur normalisée (c'est-à-dire que $h_i \in]0, 1]$) de la zone dont le processeur P_i est responsable ;

- w_i représente la largeur normalisée (c'est-à-dire que $h_i \in]0,1]$) de la zone dont le processeur P_i est responsable ;
- r_i représente le nombre normalisé de lignes dont les processeurs $P_{i,j}$ sont responsables dans le cas d'une distribution en grille ;
- c_j représente le nombre normalisé de colonnes dont les processeurs $P_{i,j}$ sont responsables dans le cas d'une distribution en grille ;
- C_Σ représente le coût d'un produit de matrice sur une plate-forme où toutes les communications sont séquentielles ;
- $C_{\max}$ représente le coût d'un produit de matrice sur une plate-forme où toutes les communications sont parallèles ;
- $\hat{C}$ est la somme des demi-périmètres d'une partition du carré unitaire ;
- $\hat{M}$ est le plus grand des demi-périmètres d'une partition du carré unitaire ;
- $D = (\delta_1, \delta_2, \ldots, \delta_p)$ représente le déséquilibre d'une distribution donnée ;
- Pour $l_1 < l_2$, $\llbracket\, l_2, l_1 \,\rrbracket$ représente l'ensemble des processeurs dont une migration de ligne de l_1 vers l_2 modifie le coût d'un rééquilibrage complet par pousse-pousse.

Chapitre 3

- d_i (ou bien d ou t_{com} dans le cas d'un bus) représente le temps nécessaire au transfert d'une tâche du maître vers le processeur P_i ;
- w_i représente le temps nécessaire au traitement d'une tâche sur le processeur P_i ;
- c_i est le nombre de tâches allouées au processeur P_i.

Chapitre 4 et 5

- la plate-forme est représentée à l'aide d'un graphe $G_P = (V_P, E_P)$ appelé *graphe de plate-forme* ;
- chaque arête $P_i \rightarrow P_j$ est étiquetée par $c_{i,j}$: le temps nécessaire à l'envoi d'un message de taille unitaire entre P_i et P_j ;
- l'application est représentée à l'aide d'un graphe de tâches $G_A = (V_A, E_A)$, le *graphe d'application* ;
- chaque arête $e_{k,l} : T_k \rightarrow T_l$ est étiquetée par un poids $data_{k,l}$ qui représente la quantité de données attachée à cette dépendance ;
- $w_{i,k}$ représente le temps d'exécution de la tâche T_k sur le processeur P_i.
- $\mathcal{A} = (\pi, \sigma)$ représente une allocation du graphe de tâche G_A sur la plate-forme G_P ;
- (t_π, t_σ) représente un ordonnancement associé à une allocation (π, σ) ;
- $sent(P_i \rightarrow P_j, e_{k,l})$ représente le nombre moyen de fichiers de type $e_{k,l}$ envoyés de P_i à P_j par unité de temps. ;
- $s(P_i \rightarrow P_j, e_{k,l})$ représente le temps moyen passé par P_i à envoyer des fichiers de type $e_{k,l}$ à P_j ;
- $cons(P_i, T_k)$ représente le nombre moyen de tâches de type T_k calculées par P_i ;
- $\alpha(P_i, T_k)$ représente le temps moyen passé par P_i à calculer des tâches de type T_k ;

– ρ représente le nombre moyen de graphes de tâches traités par unité de temps ;

Ces différentes notations peuvent se simplifier en fonction de la plate-forme.

Dans le chapitre 5, on adjoint à chaque tâche et à chaque fichier des listes de dépendances L. Les notations évoluent en conséquence et on utilise donc $sent(P_i \rightarrow P_j, e^L_{k,l})$, $s(P_i \rightarrow P_j, e^L_{k,l})$, $cons(P_i, T^L_k)$ et $\alpha(P_i, T^L_k)$.

Chapitre 6

– w_{total} représente la quantité totale de travail à distribuer ;
– α_i représente la quantité de tâches allouées au processeur P_i ;
– la communication de α_i tâches entre le maître et un esclave P_i s'écrit $G_i + \alpha_i g_i$;
– le temps nécessaire au processeur P_i pour traiter α_i tâches s'écrit $W_i + \alpha_i w_i$;

Annexe B

Bibliographie

[1] R. Agarwal, F. Gustavson et M. Zubair. «A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication». *IBM J. Research and Development* **38**, numéro 6 (1994), 673–681.

[2] D. Altilar et Y. Paker. «An optimal scheduling algorithm for parallel video processing». Dans *IEEE Int. Conference on Multimedia Computing and Systems* (1998), IEEE Computer Society Press.

[3] D. Altilar et Y. Paker. «Optimal scheduling algorithms for communication constrained parallel processing». Dans *Euro-Par 2002* (2002), LNCS 2400, Springer Verlag, pp. 197–206.

[4] R. Andonie, A. T. Chronopoulos, D. Grosu et H. Galmeanu. «Distributed backpropagation neural networks on a PVM heterogeneous system». Dans *Parallel and Distributed Computing and Systems Conference (PDCS'98)* (1998), IASTED Press, pp. 555–560.

[5] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela et M. Protasi. *Complexity and Approximation*. Springer, Berlin, Germany, 1999.

[6] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu et D. Zappala. «Improving Simulation for Network Research». Rapport technique 99-702, University of Southern California, 1999. Available at http://citeseer.nj.nec.com/bajaj99improving.html.

[7] H. Bal et M. Haines. «Approaches for integrating task and data parallelism». *IEEE Concurrency* **6**, numéro 3 (1998), 74–84.

[8] A. Barabási et R. Albert. «Emergence of scaling in random networks». *Science* **59** (1999), 509–512. Available at http://www.nd.edu/~networks/Papers/science.pdf.

[9] S. Bataineh, T. Hsiung et T.G.Robertazzi. «Closed form solutions for bus and tree networks of processors load sharing a divisible job». *IEEE Transactions on computers* **43**, numéro 10 (octobre 1994), 1184–1196.

[10] O. Beaumont, V. Boudet, A. Legrand, F. Rastello et Y. Robert. «Heterogeneity Considered Harmful to Algorithm Designers». Rapport technique RR-2000-24, LIP, ENS Lyon, juin 2000. Available at `www.ens-lyon.fr/LIP/`.

[11] O. Beaumont, V. Boudet, F. Rastello et Y. Robert. «Partitioning a Square into Rectangles: NP-Completeness and Approximation Algorithms». Rapport technique RR-2000-10, LIP, ENS Lyon, février 2000.

[12] O. Beaumont, V. Boudet, F. Rastello et Y. Robert. «Matrix-Matrix Multiplication on Heterogeneous Platforms». Rapport technique RR-2000-02, LIP, ENS Lyon, janvier 2000. Short version appears in the proceedings of ICPP'2000.

[13] O. Beaumont, L. Carter, J. Ferrante, A. Legrand et Y. Robert. «Bandwidth-centric allocation of independent tasks on heterogeneous platforms». Dans *International Parallel and Distributed Processing Symposium IPDPS'2002* (2002), IEEE Computer Society Press. Extended version available as LIP Research Report 2001-25.

[14] O. Beaumont, A. Legrand et Y. Robert. «Data Allocation Strategies for Dense Linear Algebra on two-dimensional Grids with Heterogeneous Communication Links». Rapport technique 2001-14, LIP, apr 2001.

[15] O. Beaumont, A. Legrand et Y. Robert. «The master-slave paradigm with heterogeneous processors». Dans *Cluster'2001* (2001), D. S. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki et R. Buyya, éditeurs, IEEE Computer Society Press, pp. 419–426. Extended version available as LIP Research Report 2001-13.

[16] D. Bertsekas et R. Gallager. *Data Networks*. Prenctice Hall, 1987.

[17] M. D. Beynon, T. Kurc, A. Sussman et J. Saltz. «Optimizing execution of component-based applications using group instances». *Future Generation Computer Systems* **18**, numéro 4 (2002), 435–448.

[18] V. Bharadwaj, D. Ghose, V. Mani et T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.

[19] A. Bierman et K. Jones. «Physical Topology MIB». RFC2922, september 2000.

[20] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker et R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[21] J. Blazewicz, M. Drozdowski et M. Markiewicz. «Divisible task scheduling - concept and verification». *Parallel Computing* **25** (1999), 87–98.

[22] T. Bonald et L. Massoulié. «Impact of fairness on Internet performance». Dans *SIGMETRICS/Performance* (2001), pp. 82–91. Available at http://citeseer.nj.nec.com/bonald00impact.html.

[23] J.-P. Boufflet et J. Carlier. «An exact method for minimizing the makespan of an application processed on a master slave bus oriented multiprocessor system». *Discrete Applied Mathematics* **94**, numéro 1-3 (1999), 51–76.

[24] P. Boulet, J. Dongarra, F. Rastello, Y. Robert et F. Vivien. «Algorithmic issues for heterogeneous computing platforms». Rapport technique RR-98-49, LIP, ENS Lyon, 1998. Available at www.ens-lyon.fr/LIP.

[25] T. D. Braun, H. J. Siegel et N. Beck. «Optimal use of mixed task and data parallelism for pipelined computations». *J. Parallel and Distributed Computing* **61** (2001), 810–837.

[26] R. P. Brent. «The LINPACK Benchmark on the AP1000: Preliminary Report». Dans *CAP Workshop 91* (1991), Australian National University. Website http://www.netlib.org/linpack/.

[27] H. Burch, B. Cheswick et A. Wool. «Internet Mapping Project». http://www.lumeta.com/mapping.html.

[28] M. den Burger, T. Kielmann et H. E. Bal. «TOPOMON: A Monitoring Tool for Grid Network Topology». Dans *International Conference on Computational Science (ICCS 2002)* (Amsterdam, April 21-24 2002), volume 2330, LNCS, pp. 558–567.

[29] K. L. Calvert, M. B. Doar et E. W. Zegura. «Modeling Internet Topology». *IEEE Communications Magazine* **35**, numéro 6 (juin 1997), 160–163. Available at http://citeseer.nj.nec.com/calvert97modeling.html.

[30] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson et F. Suter. «A Scalable Approach to Network Enabled Servers». Dans *Proceedings of EuroPar 2002* (Paderborn, Germany, 2002). (short paper). Available at http://www.ens-lyon.fr/~fsuter/pages/metacomputing.html.

[31] L. Carter, H. Casanova, J. Ferrante et B. Kreaseck. «Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-task Applications». Dans *International Parallel and Distributed Processing Symposium IPDPS'2003* (2003), IEEE Computer Society Press.

[32] H. Casanova et F. Berman. «GRID'2002». Dans *Parameter sweeps on the grid with APST* (2002), F. Berman, G. Fox et T. Hey, éditeurs, Wiley.

[33] H. Casanova et L. Marchal. «A Network Model for Simulation of Grid Application». Rapport technique 2002-40, LIP, oct 2002.

[34] S. Chakrabarti, J. Demmel et K. Yelick. «Models and scheduling algorithms for mixed data and task parallel programs». *J. Parallel and Distributed Computing* **47** (1997), 168–184.

[35] A. K. Chandra, D. S. Hirschberg et C. K. Wong. «Approximate algorithms for some generalized knapsack problems». *Theoretical Computer Science* **3** (1976), 293–304.

[36] S. Charcranoon, T. Robertazzi et S. Luryi. «Optimizing computing costs using divisible load analysis». *IEEE Transactions on computers* **49**, numéro 9 (septembre 2000), 987–991.

[37] Y. Chen, M. Winslett, S. Kuo, Y. Cho, M. Subramaniam et K. E. Seamons. «Performance Modeling for the Panda Array I/O Library». Dans *Proceedings of Supercomputing '96* (1996), ACM Press and IEEE Computer Society Press. Available at http://cdr.cs.uiuc.edu/pubs/super96.ps.

[38] D. N. Chiu. «Some Observations on Fairness of Bandwidth Sharing». Rapport technique, Sun Microsystems, 1999.

[39] P. Chrétienne, E. G. C. Jr., J. K. Lenstra et Z. Liu, éditeurs. *Scheduling Theory and its Applications* (1995), John Wiley and Sons.

[40] T. H. Cormen, C. E. Leiserson et R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[41] P. E. Crandall et M. J. Quinn. «Block data decomposition for data-parallel programming on a heterogeneous workstation network». Dans *2nd International Symposium on High Performance Distributed Computing* (1993), IEEE Computer Society Press, pp. 42–49.

[42] D. E. Culler et J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1999.

[43] P. D. D. Lu. «GridG: Generating Realistic Computational Grids». *Performance Evaluation Review* **30**, numéro 4 (mars 2003). Available at http://www.cs.nwu.edu/~pdinda/Papers/lu-dinda-per.pdf.

[44] W. Day et S. Hill. «Farming: Towards a Rigorous Definition and Efficient Transputer Implementation». Rapport technique 1-93*, University of Kent, Canterbury, UK, February 1993. Available at http://citeseer.nj.nec.com/day92farming.html.

[45] P. M. Dickens, P. Heidelberger et D. M. Nicol. «A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs». Dans *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)* (1994). Available at http://www.csam.iit.edu/~pmd/pubs/Distributed.ps.

[46] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste et D. Sutherland. «The architecture of the Remos system». Dans *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)* (août 2001).

[47] M. Doar. «A Better Model for Generating Test Networks». Dans *Proceedings of Globecom '96* (novembre 1996). Available at http://citeseer.nj.nec.com/doar96better.html.

[48] A. B. Downey. «Using Pathchar to Estimate Internet Link Characteristics». Dans *Measurement and Modeling of Computer Systems* (1999), pp. 222–223. Available at http://citeseer.nj.nec.com/downey99using.html.

[49] M. Drozdowski. *Selected problems of scheduling tasks in multiprocessor computing systems*. Thèse de doctorat, Instytut Informatyki Politechnika Poznanska, Poznan, 1997.

[50] P.-F. Dutot. «Master-slave Tasking on Heterogeneous Processors». Dans *International Parallel and Distributed Processing Symposium IPDPS'2003* (2003), IEEE Computer Society Press.

[51] P.-F. Dutot. «Complexity of Master-slave Tasking on Heterogeneous Trees». *European Journal of Operational Research* (2003). Special issue on the Dagstuhl meeting on Scheduling for Computing and Manufacturing systems (to appear).

[52] H. El-Rewini, T. G. Lewis et H. H. Ali. *Task scheduling in parallel and distributed systems*. Prentice Hall, 1994.

[53] M. Faloutsos, P. Faloutsos et C. Faloutsos. «On Power-law Relationships of the Internet Topology». Dans *SIGCOMM* (1999), pp. 251–262. Available at http://citeseer.nj.nec.com/faloutsos99powerlaw.html.

[54] S. Floyd et K. Fall. «Promoting the use of end-to-end congestion control in the Internet». *IEEE/ACM Transactions on Networking* **7**, numéro 4 (1999), 458–472. Available at http://citeseer.nj.nec.com/article/floyd99promoting.html.

[55] I. Foster et C. Kesselman, éditeurs. *High-performance schedulers*. Morgan-Kaufmann, 1999.

[56] G. Fox, S. Otto et A. Hey. «Matrix algorithms on a hypercube I: matrix multiplication». *Parallel Computing* **3** (1987), 17–31.

[57] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt et L. Zhang. «IDMaps: A Global Internet Host Distance Estimation Service». *IEEE/ACM Transactions on Networking* (octobre 2001). Available at http://citeseer.nj.nec.com/francis01idmaps.html.

[58] M. R. Garey et D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.

[59] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek et V. Sunderam. *PVM Parallel Virtual Machine: A Users'Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1996.

[60] M. Gondran et M. Minoux. *Graphs and Algorithms*. John Wiley & Sons, 1984.

[61] J. P. Goux, S. Kulkarni, J. Linderoth et M. Yoder. «An enabling framework for master-worker applications on the computational grid». Dans *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)* (2000), IEEE Computer Society Press.

[62] R. Govindan et H. Tangmunarunkit. «Heuristics for Internet Map Discovery». Dans *IEEE INFOCOM 2000* (Tel Aviv, Israel, mars 2000), IEEE, pp. 1371–1380. Available at citeseer.nj.nec.com/govindan00heuristics.html.

[63] W. Gropp, E. Lusk, N. Doss et A. Skjellum. «High-performance, portable implementation of the MPI Message Passing Interface Standard». *Parallel Computing* **22**, numéro 6 (1996), 789–828. Available at http://www-unix.mcs.anl.gov/mpi/mpich/.

[64] T. Hagerup. «Allocating independent tasks to parallel processors: an experimental study». *J. Parallel and Distributed Computing* **47**, numéro 2 (1997), 185–197.

[65] E. Heymann, M. A. Senar, E. Luque et M. Livny. «Adaptive scheduling for master-worker applications on the computational grid». Dans *Grid Computing - GRID 2000* (2000), R. Buyya et M. Baker, éditeurs, Springer-Verlag LNCS 1971, pp. 214–227.

[66] J. E. Hopcroft et R. M. Karp. «An $n^{5/2}$ algorithm for maximum matching in bipartite graphs». *SIAM J. Computing* **2**, numéro 4 (1973), 225–231.

[67] S. F. Hummel, E. Schonberg et L. Flynn. «Factoring: a method for scheduling parallel loops». *Communications of the ACM* **35**, numéro 8 (1992), 90–101.

[68] V. Jacobson, apr 1997. Presented at the Mathematical Science Research Institute. Available at ftp://ftp.ee.lbl.gov/pathchar/.

[69] M. Kaddoura, S. Ranka et A. Wang. «Array Decomposition for Nonuniform Computational Environments». *Journal of Parallel and Distributed Computing* **36** (1996), 91–105.

[70] A. Kalinov et A. Lastovetsky. «Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers». Dans *HPCN Europe 1999* (1999), P. Sloot, M. Bubak, A. Hoekstra et B. Hertzberger, éditeurs, LNCS 1593, Springer Verlag, pp. 191–200.

[71] F. Kelly. «Charging and rate control for elastic traffic», 1997. Available at http://citeseer.nj.nec.com/kelly97charging.html.

[72] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman et K. Verstoep. «Programming Environments for High-Performance Grid Computing: the Albatross Project». *Future Generation Computer Systems* (2002). Available at http://www.cs.vu.nl/~kielmann/papers/fgcs01.ps.gz.

[73] V. Kumar, A. Grama, A. Gupta et G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[74] C. Lee et M. Hamdi. «Parallel image processing applications on a network of workstations». *Parallel Computing* **21** (1995), 137–160.

[75] J. Lerouge et A. Legrand. «Towards realistic scheduling simulation of distributed applications». Rapport technique 2002-28, LIP, ENS Lyon, France, juillet 2002.

[76] M. Litzkow, M. Livny et M. W. Mutka. «Condor - A Hunter of Idle Workstations». Dans *Proceedings of the 8th International Conference of Distributed Computing Systems* (1988), IEEE Computer Society Press, pp. 104–111.

[77] J. Liu et D. M. Nicol. *DaSSF 3.1 User's Manual*, avril 2001. Available at http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/papers/dassf-manual-3.1.ps.

[78] M. Livny et R. Raman. *High-performance schedulers*. Morgan-Kaufmann, 1999, pp. 311–337.

[79] B. Lowenkamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste et J. Subhlok. «A resource query interface for network-aware applications». Dans *Proceedings of the Seventh International Symposium on High Performance Distributed Computing* (1998).

[80] C. Lu. «Application Signatures for Scientific Codes». Rapport de DEA, Univ. of Illinois at Urbana-Champaign, 2002. Available at http://citeseer.nj.nec.com/lu02application.html.

[81] L. Massoulié et J. Roberts. «Bandwidth Sharing: Objectives and Algorithms». Dans *INFOCOM (3)* (1999), pp. 1395–1403. Available at http://citeseer.nj.nec.com/massoulie99bandwidth.html.

[82] M. Mathis, J. Semke et J. Mahdavi. «The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm». *Computer Communications Review* **27**, numéro 3 (1997). Available at http://citeseer.nj.nec.com/mathis97macroscopic.html.

[83] A. Medina, I. Matta et J. Byers. «On the Origin of Power Laws in Internet Topologies». *ACM Computer Communication Review* **30**, numéro 2 (avril 2000). Available at http://citeseer.nj.nec.com/medina00origin.html.

[84] M. Mihail et C. H. Papadimitriou. «On the Eigenvalue Power Law». Available at http://citeseer.nj.nec.com/mihail02eigenvalue.html.

[85] N. Miller et P. Steenkiste. «Collecting Network Status Information for Network-Aware Applications». Dans *INFOCOM'00* (2000), pp. 641–650.

[86] A. Munier. *Contribution à l'étude des ordonnancements cycliques*. Thèse de doctorat, Université Paris 6, février 1991.

[87] E. Ng et H. Zhang. «Predicting Internet network distance with coordinates-based approaches». Dans *InfoCom'O2* (2002), IEEE Computer Society Press. Available at citeseer.nj.nec.com/article/ng01predicting.html.

[88] J. Padhye, V. Firoiu, D. Towsley et J. Krusoe. «Modeling TCP Throughput: A Simple Model and its Empirical Validation». Dans *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication* (1998), pp. 303–314. Available at http://citeseer.nj.nec.com/article/padhye98modeling.html.

[89] V. Paxon et S. Floyd. «Why we don't know how to simulate the internet». Dans *Proceedings of the Winder Communication Conference* (décembre 1997). Available at http://citeseer.nj.nec.com/article/floyd99why.html.

[90] Prime. URL: http://www.mersenne.org.

[91] S. Ramaswamy, S. Sapatnekar et P. Banerjee. «A framework for exploiting task and data parallelism on distributed memory multicomputers». *IEEE Trans. Parallel and Distributed Systems* **8**, numéro 11 (1997), 1098–1116.

[92] H. Renard, Y. Robert et F. Vivien. «Static load-balancing techniques for iterative computations on heterogeneous clusters.». Dans *Proceedings of Euro-Par 2003* (août 2003), volume 2790, LNCS, pp. 148–159.

[93] J. Roberts et L. Massoulié. «Bandwidth sharing and admission control for elastic traffic», 1998. Available at http://citeseer.nj.nec.com/roberts98bandwidth.html.

[94] A. L. Rosenberg. «Sharing partitionable workloads in heterogeneous NOWs: greedier is not better». Dans *Cluster Computing 2001* (2001), D. S. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki et R. Buyya, éditeurs, IEEE Computer Society Press, pp. 124–131.

[95] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

[96] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 des *Algorithms and Combinatorics*. Springer-Verlag, 2003.

[97] S. Seshan, M. Stemm et R. H. Katz. «SPAND: Shared Passive Network Performance Discovery». Dans *USENIX Symposium on Internet Technologies and Systems* (1997). Available at http://citeseer.nj.nec.com/seshan97spand.html.

[98] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. Thèse de doctorat, Dept. of Computer Science, University Of California at San Diego, 2001.

[99] G. Shao, F. Berman et R. Wolski. «Master/slave computing on the grid». Dans *Heterogeneous Computing Workshop HCW'00* (2000), IEEE Computer Society Press.

[100] G. Shao, F. Berman et R. Wolski. «Using effective network views to promote distributed application performance». Dans *International Conference on Parallel and Distributed Processing Techniques and Applications* (juin 1999), CSREA Press. Available at http://apples.ucsd.edu/pubs/pdpta99.ps.

[101] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker et J. Dongarra. *MPI the complete reference*. The MIT Press, 1996.

[102] J. Sohn, T. Robertazzi et S. Luryi. «Optimizing computing costs using divisible load analysis». *IEEE Transactions on parallel and distributed systems* **9**, numéro 3 (mars 1998), 225–234.

[103] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura et A. A. Chien. «The MicroGrid: a Scientific Tool for Modeling Computational Grids». Dans *Supercomputing* (2000). Available at http://www.sc2000.org/techpapr/papers/pap.pap286.pdf.

[104] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman et J. Saltz. «Executing multiple pipelined data analysis operations in the grid». Dans *2002 ACM/IEEE Supercomputing Conference* (2002), ACM Press.

[105] J. Subhlok, J. Stichnoth, D. O'Hallaron et T. Gross. «Exploiting task and data parallelism on a multicomputer». Dans *Fourth ACM SIGPLAN Symposium on Priciples & Practices of Parallel Programming* (mai 1993), ACM Press.

[106] J. Subhlok et G. Vondran. «Optimal use of mixed task and data parallelism for pipelined computations». *J. Parallel and Distributed Computing* **60** (2000), 297–319.

[107] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker et W. Willinger. «Network topology generators: Degree-based vs structural». Dans *Proceedings of SIGCOMM'02* (août 2002), ACM Press. Available at http://citeseer.nj.nec.com/tangmunarunkit02network.html.

[108] K. Taura et A. A. Chien. «A heuristic algorithm for mapping communicating tasks on heterogeneous resources». Dans *Heterogeneous Computing Workshop* (2000), IEEE Computer Society Press, pp. 102–115.

[109] The PM2 Team. *Getting Started with PM2*, 2001. http://www.pm2.org.

[110] F. Vraalsen, R. A. Aydt, C. L. Mendes et D. A. Reed. «Performance Contracts: Predicting and Monitoring Grid Application Behavior». Dans *GRID* (2001), pp. 154–165. Available at http://citeseer.nj.nec.com/vraalsen01performance.html.

[111] R. Wang, A. Krishnamurthy, R. Martin, T. Anderson et D. Culler. «Modeling communication pipeline latency». Dans *Measurement and Modeling of Computer Systems (SIGMETRICS'98)* (1998), ACM Press, pp. 22–32.

[112] B. M. Waxman. «Routing of Multipoint Connections». *IEEE Journal on Selected Areas in Communications* **6**, numéro 9 (December 1988), 1617–1622.

[113] J. B. Weissman. «Scheduling multi-component applications in heterogeneous wide-area networks». Dans *Heterogeneous Computing Workshop HCW'00* (2000), IEEE Computer Society Press.

[114] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 1996.

[115] R. Wolski. «Dynamically forecasting network performance using the Network Weather Service». *Cluster Computing* **1**, numéro 1 (1998), 119–132. Available at http://www.cs.ucsd.edu/users/rich/papers/nws-tr.ps.gz.

[116] R. Wolski, N. T. Spring et J. Hayes. «The network weather service: a distributed resource performance forecasting service for metacomputing». *Future Generation Computer Systems* **15**, numéro 5-6 (1999), 757–768. Available at http://www.cs.ucsd.edu/users/rich/papers/nws-arch.ps.gz.

[117] R. Wolski, N. T. Spring et J. Hayes. «Predicting the CPU availability of time-shared Unix systems on the computational grid». *Cluster Computing* **3**, numéro 4 (2000), 293–301. Available at http://www.cs.ucsd.edu/users/rich/papers/nws-cpu.ps.gz.

[118] Y. Yang et H. Casanova. «Multi-round algorithm for scheduling divisible workload applications: analysis and experimental evaluation». Rapport technique CS2002-0721, Dept. of Computer Science and Engineering, University of California, San Diego, 2002.

[119] W. Yu. *The two-machine flow shop problem with delays and the one-machine total tardiness problem*. Thèse de doctorat, Technishe Universiteit Eidhoven, juin 1996.

[120] «SETI@home project». Website http://setiathome.ssl.berkeley.edu/.

Annexe C

Liste des publications

J'ai écrit, en collaboration avec Yves ROBERT, un livre d'enseignement destiné au second cycle universitaire intitulé *Algorithmique Parallèle*. Ce livre est paru dans le courant de l'année 2003 aux éditions DUNOD.

Revues internationales avec comité de lecture

[1] O. Beaumont, A. Legrand, L. Marchal et Y. Robert. «Scheduling strategies for mixed data and task parallelism on heterogeneous clusters». *Int. Journal of High Performance Computing Applications* (2003 (to appear)).

[2] O. Beaumont, A. Legrand et Y. Robert. «Scheduling divisible workloads on heterogeneous platforms». *Parallel Computing* (2003 (to appear)).

[3] O. Beaumont, A. Legrand et Y. Robert. «The master-slave paradigm with heterogeneous processors». *IEEE Trans. Parallel Distributed Systems* (2003 (to appear)).

[4] O. Beaumont, A. Legrand, F. Rastello et Y. Robert. «Dense linear algebra kernels on heterogeneous platforms : redistribution issues». *Parallel Computing* **28** (2002), 155–185.

[5] O. Beaumont, A. Legrand et Y. Robert. «Static scheduling strategies for heterogeneous systems». *Computing and Informatics* **21** (2002), 413–430.

[6] O. Beaumont, A. Legrand, F. Rastello et Y. Robert. «Static LU decomposition on heterogeneous platforms». *Int. Journal of High Performance Computing Applications* **15**, numéro 3 (2001), 310–323.

Revues nationales avec comité de lecture

[7] A. Legrand. «Équilibrage de charge statique pour noyaux d'algèbre linéaire sur plate-forme hétérogène». *Technique et Science Informatique (TSI), Numéro spécial RenPar'13* (2002), 711–734.

Chapitres de livre

[8] O. Beaumont, V. Boudet, A. Legrand, F. Rastello et Y. Robert. «Static Data Allocation and Load Balancing Techniques for Heterogeneous Systems». Dans *Annual Review of Scalable Computing*, C. Yuen, éditeur, volume 4. World Scientific, 2002, chapitre 1, pp. 1–37.

Conférences internationales avec comité de lecture

[9] H. Casanova, A. Legrand et L. Marchal. «Scheduling Distributed Applications : the SimGrid Simulation Framework». Dans *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)* (mai 2003).

[10] O. Beaumont, A. Legrand et Y. Robert. «Scheduling strategies for mixed data and task parallelism on heterogeneous clusters and grids». Dans *PDP'2003, 11th Euromicro Workshop on Parallel, Distributed and Network-based Processing* (2003), IEEE Computer Society Press, pp. 209–216.

[11] O. Beaumont, A. Legrand et Y. Robert. «Optimal algorithms for scheduling divisible workloads on heterogeneous systems». Dans *HCW'2003, the 12th Heterogeneous Computing Workshop* (2003), IEEE Computer Society Press.

[12] O. Beaumont, L. Carter, J. Ferrante, A. Legrand et Y. Robert. «Bandwidth-centric allocation of independent tasks on heterogeneous platforms». Dans *International Parallel and Distributed Processing Symposium IPDPS'2002* (2002), IEEE Computer Society Press.

[13] C. Banino, O. Beaumont, A. Legrand et Y. Robert. «Scheduling strategies for master-slave tasking on heterogeneous processor grids». Dans *PARA'02 : International Conference on Applied Parallel Computing* (2002), LNCS 2367, Springer Verlag, pp. 423–432.

[14] O. Beaumont, A. Legrand et Y. Robert. «Static scheduling strategies for heterogeneous systems». Dans *ISCIS XVII, Seventeenth International Symposium On Computer and Information Sciences* (2002), CRC Press, pp. 18–22.

[15] O. Beaumont, A. Legrand et Y. Robert. «A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs». Dans *ISCIS XVII, Seventeenth International Symposium On Computer and Information Sciences* (2002), CRC Press, pp. 115–119.

[16] O. Beaumont, V. Boudet, A. Legrand, F. Rastello et Y. Robert. «Heterogeneous Matrix-Matrix Multiplication, or Partitioning a Square into Rectangles : NP-Completeness and Approximation Algorithms». Dans *EuroMicro Workshop on Parallel and Distributed Computing (EuroMicro'2001)* (2001), IEEE Computer Society Press, pp. 298–305.

[17] O. Beaumont, A. Legrand et Y. Robert. «The master-slave paradigm with heterogeneous processors». Dans *Cluster'2001* (2001), IEEE Computer Society Press, pp. 419–426.

[18] O. Beaumont, V. Boudet, A. Legrand, F. Rastello et Y. Robert. «Heterogeneity Considered Harmful to Algorithm Designers». Dans *Cluster'2000* (2000), IEEE Computer Society Press, pp. 403–404.

Conférences nationales avec comité de lecture

[19] A. Legrand. «Équilibrage de charge statique pour la décomposition LU sur une plate-forme hétérogène». Dans *13ième Rencontres Francophones du Parallélisme des Architectures et des Systèmes* (Paris, La Villette, 24-27 Avril 2001).

[20] O. Beaumont, A. Legrand et Y. Robert. «Ordonnancement en régime permanent pour plates-formes hétérogènes». Dans *GRID'2002, Actes de l'école thématique sur la globalisation des ressources informatiques et des données* (2002), INRIA Lorraine, pp. 325–334.

[21] A. Legrand. «Simulation pour l'ordonnancement distribué». Dans *GRID'2002, Actes de l'école thématique sur la globalisation des ressources informatiques et des données* (2002), INRIA Lorraine, pp. 155–164.

Résumé :
Les travaux présentés dans cette thèse portent sur les difficultés algorithmiques soulevées par l'introduction de l'hétérogénéité des plates-formes modernes dans le calcul parallèle et distribué. Les contributions de cette thèse se situent à trois niveaux : 1) Algorithmique Parallèle : distributions hétérogènes pour les noyaux d'algèbre linéaire denses (produit de matrice, décomposition LU), technique de rééquilibrage, légère et efficace en cas de petites variations de charge des processeurs ; 2) Modélisation et simulation : l'instabilité latente des plates-formes de calcul distribuées à grande échelle interdit toute validation expérimentale grandeur nature d'un algorithme ou d'une politique d'ordonnancement. Nous avons proposé des modèles simples et un simulateur réaliste pour palier ce problème ; 3) Ordonnancement : un certain nombre d'applications sont constituées d'un grand nombre de tâches indépendantes et de caractéristiques identiques. Nous avons établi des résultats de complexité et proposé des approximations pour différentes modélisation de ce problème.
Mots-clés :
Algorithmique parallèle, ordonnancement, hétérogénéité, régime permanent, simulation et modélisation de plates-formes de calcul.

Abstract:
The results summarized in this document deal with the algorithmic difficulties raised by the heterogeneity of modern parallel and distributed computing platforms. The contributions of this work are divided in three parts: 1) Parallel algorithms: efficient heterogeneous distributions for most dense linear algebra kernels (matrix product, LU decomposition), lightweight redistribution techniques to cope with load variations; 2) Simulation and modeling: the genuine instability of wide-area distributed computing platforms forbids the use of real experiments to to test the behavior of an algorithm or of a scheduling policy. Therefore we have proposed simple models and have developed a realistic simulator to cope with this problem; 3) Scheduling: lots of applications are constituted of a very large number of independent identical tasks. We have established some complexity results and have proposed some approximations for different models of this situation.
Keywords:
Parallel algorithms, scheduling, heterogeneity, steady-state, simulation and modeling of computing platforms.