Chapitre 1

Machines virtuelles et techniques d'ordonnancement

Jean-Claude Konig (LAMI) – Jean-Louis Roch (LMC-IMAG)

De manière générale, que ce soit dans un cadre dynamique ou statique, l'ordonnancement d'un programme parallèle prend en entrée un graphe de tâches et un modèle de machine et fournit en sortie un ordonnancement à un facteur borné (aussi proche de 1 que possible) de l'optimal. Dans le cadre dynamique (ou à la volée), le graphe est inconnu car construit au cours de l'exécution; la durée d'une tâche ou d'une communication n'est connue qu'après sa terminaison. Dans le cadre statique, le graphe est supposé connu avant l'exécution du programme. Dans cet article, nous introduisons les différents modèles de machines considérés et les techniques d'ordonnancement qui leur sont associées dans les cadres statique et dynamique.

1.1 Introduction

Le placement des tâches sur l'architecture cible dans le but de minimiser la durée d'exécution est un problème complexe qui intègre un grand nombre de paramètres qui seront étudiés dans cette école et qui sont liés à des choix algorithmiques (routage par exemple) ou des contraintes physiques (mode de communication, bande passante par exemple).

Le placement et l'ordonnancement des tâches doivent donc être décomposés en plusieurs étapes pour limiter à la fois la complexité et augmenter la portabilité et la généricité. La démarche classique est donc de placer et d'ordonnancer les tâches sur une machine virtuelle et d'émuler ensuite la machine virtuelle par la machine

cible. Or il semble difficile (impossible) de concevoir une machine virtuelle simple et universelle qui soit acceptable (comme l'est la machine de von Neumann pour le calcul séquentiel) c'est-à-dire qui puisse servir de référence pour mesurer la complexité. Une raison clé de cet échec est l'impossibilité de faire abstraction des transferts de données (de coûts non uniformes de part la hiérarchie mémoire) pour obtenir des résultats de complexité ayant un sens expérimental.

Ainsi les modèles (de type PRAM synchrone [30] ou asynchrone [11, 4]) permettent d'analyser le coût d'un algorithme à partir de paramètres caractéristiques des graphes associés à ses exécutions : nombre d'opérations (les tâches sont séquentielles), volume d'accès en mémoire partagée (les arêtes de dépendance de données correspondent potentiellement à des accès distants), ... Mais pour que les résultats ainsi obtenus puissent être exploitables et validés expérimentalement, il est nécessaire de montrer comment l'ordonnancement du graphe (indépendant du nombre de processeurs) peut être réalisé. Il s'agit d'une part de montrer qu'un ordonnancement optimal du programme apporte l'efficacité et d'autre part d'exhiber un algorithme permettant de calculer un ordonnancement optimal ou à un facteur de l'optimal (α-compétitif). Ces deux études sont complémentaires; ce chapitre est surtout centré sur la présentation d'algorithmes α-compétitifs. Dans ce cadre, un ordonnancement optimal pour un programme donné peut éventuellement être séquentiel.

L'ordonnancement trivial sur une machine PRAM théorique (principe de Brent) ne peut fournir une émulation raisonnable sur une machine, même simple, car il repose sur des hypothèses fausses comme l'indépendance entre les temps de latence pour accéder à la mémoire partagée et la possibilité de répartir équitablement sans surcoût de temps l'ensemble des instructions d'une étape sur les processeurs.
Une façon de rendre le modèle plus fiable est de montrer la possibilité d'ordonnancer des classes de graphes spécifiques sur des modèles de machine plus réalistes, comme :

- granularité globale : le nombre d'accès potentiels en mémoire distante est négligeable devant le volume de calcul;
- granularité grossière : toutes les tâches sont de même durée et ont chacune un volume d'accès distants négligeable devant leur nombre d'opérations;
- topologie particulière : le graphe de dépendance est un arbre dont les tâches sont de même coût.

De tels graphes sont suffisament génériques pour être effectivement rencontrés dans des applications irrégulières : les classes précédentes englobent en effet de nombreux cas où par exemple durées de tâches et volumes d'accès à des données distantes sont inconnus ou variables, ou encore le graphe est inconnu et construit au cours de l'exécution par la création de nouvelles tâches.

Nous nous proposons dans ce chapitre de présenter les résultats connus pour l'ordonnancement dynamique et statique de tels graphes sur un modèle simpliste

et d'étudier leur évolution lorsqu'on intègre au modèle certains paramètres caractérisant l'architecture cible comme :

- le nombre de processeurs
- la latence liée aux communications
- l'hétérogénéité des processeurs

Il est clair qu'un modèle intégrant seulement ces trois paramètres pour être à peu prêt valide nécessite de nombreuses hypothèses du type : seule l'application considérée s'exécute sur le système, la bande passante est suffisante pour ne pas entrainer une saturation des débits ... Malgré celà un tel modèle devrait permettre de mieux analyser le parallélisme exploitable d'une application et de donner un ordonnancement plus efficace à l'exécution.

Organisation. Après la définition d'un problème d'ordonnancement, la première section précise les mécanismes systèmes nécessaires à sa mise en œuvre ainsi que les critères permettant d'évaluer ses performances. Puis, les algorithmes classiques d'ordonnancement lorsque les processeurs sont supposés identiques et les communications non prises en compte, sont rappelés. L'analyse de leurs performances permet d'introduire les techniques de base utilisées pour l'évaluation théorique d'un algorithme d'ordonnancement. Dans les sections suivantes, les principaux algorithmes permettant de prendre en compte les communications sont présentés : ordonnancements dynamiques (section 3) et ordonnancements statiques (section 4). Enfin, les principaux résultats concernant l'ordonnancement sur des machines hétérogènes sont présentés.

1.2 Le problème d'ordonnancement

Dans cette partie, nous définissons le formalisme de graphe le plus général pris en entrée d'un algorithme d'ordonnancement. L'exécution d'un programme parallèle décrit (de manière dynamique dans le cadre le plus général) un graphe de dépendance. Ce graphe est orienté sans cycle : les nœuds correspondent à des blocs d'instructions (tâches) et les arêtes à des relations de dépendance sur l'accès à une donnée (transitions).

1.2.1 Caractérisation du problème : graphe de dépendance

De manière générale, l'exécution d'un programme parallèle peut-être décrite par un graphe de flots de données (macro data-flow) :DFG. Ce graphe est biparti, orienté et sans cycle avec comme ensemble de nœuds $J = \{j_1, \ldots, j_n\}$ correspondant aux blocs d'instructions séquentielles (j signifiant *job*, nous dirons tâche) et $T =$

$\{t_1, \ldots, t_m\}$ correspondant aux opérations de lecture et écriture d'une donnée (vue comme un bloc de mots) en mémoire globale (t signifiant *transition*).

Tous les nœuds d'entrée (i.e. de degré entrant nul) et de sortie (i.e. de degré sortant 0) sont des transitions; les premiers correspondent aux données en entrée du programme (fichiers en lecture par exemple) alors que les derniers correspondent à ses sorties.

Le graphe définit un ordre partiel sur J, dit de précédence et noté $\prec$. Pour tout $1 \leq i, k \leq n$, $j_i \prec j_k$ si et seulement s'il existe un chemin orienté de j_i vers j_k.

N.B. Dans toute la suite, nous nous limitons à l'ordonnancement de graphes *d'arité bornée* (qui est celui des algorithmes dits EREW). Dans le cas général (algorithme CRCW), nous supposons donc que le graphe (le programme) a été transformé en une description sous forme graphe de tâches à arité bornée. Une telle transformation peut augmenter d'un facteur quadratique au plus le nombre de sommets (tâches et transitions) du graphe.

Contraintes de placement. Généralement, des fonctions de contraintes $\phi_t : T \mapsto \mathcal{P}(\{M_i, 1 \leq i \leq p\})$ et $\phi_j : J \mapsto \mathcal{P}(\{P_i, 1 \leq i \leq p\})$ permettent de spécifier des contraintes de localisation pour certaines transitions ou pour certaines tâches. Une transition en entrée peut par exemple n'être disponible que sur un module mémoire, une tâche ne peut être exécutée que sur un sous-ensemble de processeurs. Dans le cadre de cet article, nous nous limitons au cas où aucune contrainte n'est spécifiée, i.e. $\phi_t(x) = (\{M_i, 1 \leq i \leq p\})$ et $\phi_j(y) = (\{P_i, 1 \leq i \leq p\})$ pour tout $x \in T$, $y \in J$.

Déterminisme des exécutions. Le graphe de dépendance est directement associé à une exécution; il ne peut souvent être complètement déterminé qu'après exécution compléte du programme. Cependant, dans tous les cas, nous supposerons que l'exécution reste *déterministe* : le graphe de dépendance peut être construit complétement dynamiquement, mais reste le même pour les mêmes valeurs en entrée quel que soit l'ordonnancement effectué (même séquentiel). Cette contrainte est liée à la difficulté (impossibilité si $P \neq NP$) de trouver un ordonnancement polynômial pour un algorithme de coût polynômial sur une machine non déterministe (i.e. dont toute exécution admet un ordonnancement polynômial).

1.2.2 Ordonnancement du graphe sur une machine

Une architecture distribuée est vue comme un ensemble d'unités de calcul P_i chacune associée à un module de mémoire M_i. L'ordonnancement consiste alors à affecter à chaque transition t (qui correspond à une donnée "en assignation unique") un module mémoire $M(t)$ et à chaque job j le processeur $P(j)$ sur lequel il sera exécuté. L'ordonnanceur spécifie en outre quels évènements conditionnent le démarrage ou la préemption d'une tâche (généralement fin de communication et fin de tâche de calcul).

1.2.2.a Ordonnancement global et ordonnancement système

Deux niveaux d'ordonnancement peuvent être distingués :

- *macroscopique* : ordonnancement global des nœuds du graphe de dépendance (tâches et données) sur les ressources (processeurs et modules mémoire).
- *microscopique* : ordonnancement local des processus concurrents sur le processeur et des instructions atomiques pour minimiser les temps d'attente (par exemple anticipation du chargement d'une ligne dans le cache ou avancement des communications).

Nous ne nous intéressons dans ce chapitre qu'à l'ordonnancement macroscopique, l'ordonnancement microscopique étant supposé réalisé localement sur chaque processeur par le système.

Calculer un ordonnancement consiste donc à allouer à chaque nœud du graphe (donnée ou tâche) une ressource en complétant la relation de précédence $\prec$ initiale. Chaque exécution du programme après ordonnancement définit une relation d'ordre partiel $\prec_i$ sur l'ensemble des différentes instructions atomiques exécutées. Dans tous les cas, $\prec_i$ doit être compatible avec $\prec$: l'ordonnancement calculé est dit "admissible" si, pour toute exécution et pour tout couple d'instructions atomiques (i_1, i_2) appartenant respectivement aux jobs j_1 et j_2 alors :

$$(j_1 \prec j_2) \Longrightarrow (i_1 \prec_i i_2).$$

1.2.2.b Opérations utilisées pour réaliser l'ordonnancement

Pour réaliser et contrôler l'ordonnancement calculé, l'ordonnanceur utilise certaines primitives supposées fournies par le système. Sauf précisé, toutes ces primitives sont supposées de coût constant.

Exécution d'une tâche . Une tâche (un nœud du DFG) peut être exécutée sur un processeur quelconque.

Communication. Un processeur peut lire ou écrire un objet dans n'importe quel module mémoire distant. Le coût de l'accés à n données est supposé borné par $h.n$ où h est une constante de la machine [30, 9]. Les accès mémoire sont asynchrones. Pour contrôler l'ordonnancement (via les transitions, une tâche accède en lecture à des données écrites par ses prédécesseurs), des outils de synchronisation globaux sont supposés disponibles (verrous, sémaphores).

Duplication. La duplication consiste à exécuter plusieurs fois une même tâche sur différents processeurs; ceci peut permettre de construire des ordonnancements minimisant les coûts de communication.

Migration. Certains algorithmes d'ordonnancement utilisent en outre une opération de préemption, qui permet de suspendre une tâche au cours de son exécution pour la relancer, éventuellement plus tard, sur un autre processeur [29, 3]. Deux types de préemption sont distingués :

- *migration par redémarrage :* lorsqu'une tâche est déplacée sur un autre processeur, son exécution reprend depuis son début.
- *migration :* lorsqu'une tâche est déplacée sur un autre processeur, son exécution reprend à partir de sa dernière instruction exécutée.

Un algorithme d'ordonnancement *non-préemptif* n'utilise aucune de ces opérations : il perd tout contrôle sur une tâche dés qu'il l'a démarrée sur un processeur, sauf information sur sa terminaison. Le coût d'une opération de migration est supposé constant.

Remarque. L'utilisation de la migration est trés liée au modèle de programmation (comment sont implémentées les tâches) et au modèle de machine. Par exemple, si les tâches sont sans bloquage (c'est le cas dans le graphe de dépendance), la migration est inutile lorsque la machine est composée de processeurs identiques. Par contre, elle peut s'avérer utile sur des machines hétérogènes [29].

1.2.3 Notation des problèmes d'ordonnancement

Nous allons utiliser pour classifier les problèmes d'ordonnancement la notation introduite dans [14] et étendue dans [34] pour les problèmes avec communications. Nous l'étendons pour préciser le cas où le graphe de dépendance est construit en cours d'exécution, les coûts des tâches et des communications étant inconnus jusqu'à leur terminaison.

Cette notation est constituée de trois champs : α, β, γ. Ces trois paramètres permettent de règler le niveau d'abstraction de la machine virtuelle et les hypothèses sur le type de graphes considérés.

Le paramètre α définit le type des processeurs et leur nombre. Pour nous :

- $\alpha = P$: signifie que les processeurs sont identiques et que leur nombre m est une entrée du problème.
- $\alpha = \overline{P}$: signifie que les processeurs sont identiques, leur nombre étant suffisant ou non borné.
- $\alpha = P_m$: indique un nombre fixé de processeurs identiques.

Le paramètre β définit le type du graphe de précédence, les temps d'exécution des tâches, le coût de communication, l'autorisation ou pas de la duplication, l'autorisation ou pas de la préemption.
Donc $\beta = \beta_1\beta_2\beta_3\beta_4$ où :

- $\beta_1 \in \{prec?, prec, arbre, chaine, ..., \bullet\}$: Type du graphe de précédence des tâches.
  - $\beta_1 = prec$: le graphe de précédence des tâches est quelconque mais de structure connue;
  - $\beta_1 = prec?$: le graphe de précédence des tâches est quelconque mais inconnu (construit en cours d'exécution);
  - $\beta_1 = arbre$: le graphe de précédence des tâches est un arbre;
  - $\beta_1 = \bullet$: les tâches sont indépendantes.

- $\beta_2 \in \{com, com?, c_{jk}, c, \bullet\}$: communication.
  - $\beta_2 = com$: temps de communication donnés par le graphe de précédence des tâches.
  - $\beta_2 = com?$: temps de communication inconnu avant fin de la communication effective mais constant.
  - $\beta_2 = com? < \alpha \sum p_i$: temps de communication inconnu mais volume total de données communiquées borné par une constante fois le volume total de calcul.
  - $\beta_2 = c_{jk}$: le temps de communication entre la tâche j et la tâche k est c_{jk}.
  - $\beta_2 = c$: temps de communication partout égaux.
  - $\beta_2 = \bullet$: temps de communications nuls.
- $\beta_3 = \in \{dup, mig, \bullet\}$: la migration et la duplication sont autorisées ou non
- $\beta_4 \in \{p_i = 1, p_i = ?, \bullet\}$: durée des tâches;
  - $\beta_4 = (p_i = 1)$: toutes les tâches ont une durée d'execution unitaire.
  - $\beta_4 = (p_i = ?)$: les tâches sont de durées quelconques et inconnue avant exécution.
  - $\beta_4 = \bullet$: les durées des tâches sont définies par le graphe de précédence des tâches.

Le paramètre γ définit l'objectif. Dans ce chapitre, le but sera toujours de minimiser la durée d'exécution, ce but est noté $\gamma = C_{max}$.

1.2.4 Évaluation du coût d'un algorithme d'ordonnancement

Dès que l'on considère des machines virtuelles non triviales, le problème de trouver un ordonnancement optimal d'un graphe devient NP-difficile. On s'intéresse

donc à rechercher des heuristiques polynômiales (voire linéaire) pouvant fournir un ordonnancement proche de l'optimal.

L'efficacité d'une heuristique peut-être évaluée de diverses façons. Dans ce cours, nous nous contenterons de mesurer la qualité d'un algorithme d'ordonnancement par deux critères :

- **ratio** (ou compétivité) : le maximum du rapport sur tous les graphes entre la durée obtenue par l'ordonnancement de l'heuristique et la durée obtenue par un ordonnancement optimal.
- **surcoût** d'ordonnancement : le nombre d'opérations requises pour la mise en œuvre effective de l'ordonnancement.

1.2.4.a Ratio de performance ou compétitivité

Définition 1.2.1 *Soit h un algorithme d'ordonnancement. Le ratio de performance de h, noté $R(h)$, est le maximum du rapport sur tous les graphes entre la durée donnée par l'ordonnancement fourni par h et la durée de l'ordonnancement optimal, c'est-à-dire* $R(h) = Max\frac{C_{max}(h)}{C_{max}^{opt}}$.
On dit qu'un algorithme h est ρ-approché (ou encore ρ-compétitif) si $R(h) \leq \rho$.

Un algorithme h est d'autant plus efficace que son ratio $R(h)$ est petit. La démarche dans le cas d'un problème NP-complet est donc de trouver la meilleure heuristique possible. En fait la plupart du temps nous ne sommes capables que de donner une borne supérieure du ratio de la meilleure heuristique (en étudiant le ratio d'une heuristique connue) et une borne inférieure de ce ratio.

Une technique classique pour obtenir une borne inférieure du ratio est basée sur le résultat suivant, facile à démontrer et appelé "théorème de l'impossibilité" [25].

Théorème 1.2.1 *Etant donné un problème d'ordonnancement et un entier c, si la question de savoir si un graphe G peut-être ordonnancé en c unités de temps ou moins est NP-complet alors on ne peut pas espérer avoir une heuristique pour ce problème d'ordonnancement ayant un ratio inférieur à* $\frac{c+1}{c}$.

La valeur du ratio de performance permet de cerner la complexité pratique du problème et de traduire l'augmentation de la complexité du problème en fonction des nouveaux paramètres intégrés au modèle.
D'un point de vue pratique, l'heuristique de meilleur ratio n'est pas forcément celle à intégrer si le surcoût induit par sa mise en œuvre s'avère trop important.

1.2.4.b Surcoût d'ordonnancement

Définition 1.2.2 *Soit h un algorithme d'ordonnancement et G un graphe de dépendance quelconque. Le surcoût d'ordonnancement de h, noté $\sigma(h)$, est le nombre d'opérations effectuées par h pour exécuter G.*

Il est souvent difficile de séparer a priori le surcoût lié à la mise en œuvre de l'ordonnancement de la longueur de l'ordonnancement lui-même (caractérisé par le ratio). Aussi, son étude est basée sur l'analyse du ratio et complétée par la prise en compte des opérations pour l'ordonnancement.

Le surcoût peut être séparé en deux parties : d'une part les opérations requises pour le calcul de l'ordonnancement et d'autre part celles utilisées pour exécuter le graphe de tâches en garantissant le respect de la relation de précédence.

Cadre statique. Dans le cadre statique, le graphe étant connu à la compilation, le calcul de l'ordonnancement n'implique pas de surcoût. Par contre son contrôle nécessite un surcoût lié – au moins – à la taille du graphe. La réalisation d'un ordonnancement pré-calculé étant triviale, nous considérerons uniquement le ratio de performance pour l'analyse des algorithmes d'ordonnancement statique.

Cadre dynamique. Dans le cadre dynamique, au surcoût de contrôle s'ajoute aussi celui du calcul de l'ordonnancement. Dans tous les cas, chaque nœud et chaque transition devant être placés, il est borné inférieurement par la taille du graphe.

Notation. Pour exprimer le surcoût d'ordonnancement, dans la suite nous désignerons par N_a le nombre de tâches et par N_t le nombre de transitions. Comme tous les nœuds sont supposés d'arité bornée, nous désignerons par $n = N_a + N_t$ le nombre de nœuds dans G et par n_d le degré de G. Une borne sur la taille du graphe est donc nn_d.

1.3 Ordonnancements sans communication

Nous présentons d'abord les résultats préliminaires concernant l'ordonnancement (dynamique ou statique) sans communication.

1.3.1 Problème $P_m|prec, p_i =?|C_{max}$: algorithme de liste

Dans le cas de la régulation dynamique de la charge, l'ordonnancement le plus rencontré en pratique consiste basiquement, lorsqu'un processeur devient inactif, à lui allouer une tâche prête s'il en existe. Un tel ordonnancement est appelé "algorithme de liste" : il requiert de gérer une structure évoluant dynamiquement et permettant d'accéder aux tâches prêtes.

Dans cette section, seul le ratio de performance d'un tel algorithme pour un graphe de précédence quelconque est étudié. Le surcoût induit par sa mise en œuvre sur une architecture distribuée (gestion du graphe et contrôle de l'ordonnancement d'une part, introduction des communications d'autre part) est étudié dans la section 1.4.

Théorème 1.3.1 *[12] Si les communications ne sont pas prises en compte, tout algorithme d'ordonnancement de type liste a un ratio de performance borné par* $\left(2 - \frac{1}{m}\right)$ *sur une machine constituée de m processeurs identiques.*

Nous rappelons la preuve de ce résultat car son schéma intervient dans de nombreuses démonstrations.
Soit T_m la longueur de l'ordonnancement fourni par l'algorithme de liste sur m machines. Un algorithme de liste est tel que, à chaque instant, au moins un processeur exécute une tâche. Ainsi, si à un instant donné un processeur est inactif alors il existe au moins un processeur qui exécute une tâche. Soit t_{j_1} l'une des tâches qui s'est terminé à la date T_m et soit d_{j_1} la date de début d'exécution de t_{j_1}. Deux cas peuvent être distingués :

cas 1 : soit aucun processeur n'a été inactif avant d_{j_1}.

cas 2 : soit il existe au moins un processeur inactif à un certain instant avant d_{j_1}. Soit θ la plus grande date avant d_{j_1} à laquelle au moins un processeur est inactif. À θ, t_{j_1} n'est pas prête car sinon elle aurait été affectée à un processeur inactif. Donc il existe une tâche t_{j_2} telle que t_{j_2} est en cours d'exécution à θ et $t_{j_2} \prec t_{j_1}$. Soit d_{j_2} la date de début d'exécution de t_{j_2}.

En appliquant récursivement ce schéma jusqu'à ce que le cas 1 arrive, nous construisons une séquence de tâches $t_{j_k} \prec \ldots \prec t_{j_2} \prec t_{j_1}$ telle que, à tout instant, soit tous les processeurs sont actifs soit un processeur exécute une tâche t_{j_i} $(1 \leq i \leq k)$.

Soit T_∞ le temps minimal sur un nombre infini de processeurs et T_1 le nombre total d'opérations exécutées (temps sur un processeur séquentiel, appelé aussi travail). Le temps total d'inactivité $\#I$ est défini par $\#I = mT_m - T_1$. Pour $1 \leq i \leq k$, soit l_i la durée de la tâche t_{j_i}. Nous avons $\#I \leq (m-1)\sum_{i=1}^{k} l_i$, d'où :

$$pT_m \leq T_1 + (p-1)\sum_{i=1}^{k} l_i.$$

En outre, les tâches t_{j_i}, $1 \leq i \leq k$ étant sur un chemin critique, $\sum_{j=1}^{k} l_i \leq T_\infty$. On a finalement :

$$T_m \leq \frac{T_1}{m} + \left(1 - \frac{1}{m}\right) T_\infty \tag{1.1}$$

Par ailleurs, soit $T_m^\star$ la longueur de l'ordonnancement optimal. On a $T_\infty \leq T_m^\star$ et, comme T_1 opérations doivent être exécutées par tout ordonnancement, $mT_m^\star \geq T_1$. Remplaçant dans 1.1, on obtient : $T_m \leq \left(2 - \frac{1}{m}\right) T_m^\star$. □

Le théorème précédent 1.3.1 est donné dans une version restreinte. [3]. De manière plus générale, la borne 1.1 reste inchangée lorsque la relation de précédence $\prec'$ considérée par l'algorithme de liste est plus forte que celle $\prec$ utilisée pour calculer l'ordonnancement optimal. La preuve est directe puisque on aura encore $t_{j_k} \prec \ldots \prec t_{j_2} \prec t_{j_1}$. Clairement, la même remarque s'applique si la durée des tâches est augmentée artificiellement. Celà implique que ni l'ajout de contraintes de précédence telles que des barrières de synchronisation pour obtenir un graphe de dépendance bien structuré (par exemple semi-parallèle), ni l'insertion artificielle d'opérations de découpe pour que toutes les tâches aient la même durée ne peuvent améliorer le ratio d'un algorithme d'ordonnancement en ligne.

Remarque : principe de Brent constructif. En corollaire de l'encadrement 1.1, on obtient une démonstration constructive générale du principe de Brent pour des tâches de longueur arbitraire (et non forcément unitaires).

Corollaire 1.3.1 *Si un programme parallèle effectue T_1 opérations (temps séquentiel) et peut s'exécuter en temps paralléle minimal T_∞ sur un nombre infini de processeurs, alors il peut être exécuté sur m processeurs en temps*

$$Max\left\{\left\lceil\frac{T_1}{m}\right\rceil, T_\infty\right\} \leq T_m \leq \left\lceil\frac{T_1}{m} + T_\infty\right\rceil \tag{1.2}$$

1.3.2 Bornes inférieures sur le ratio

Une question naturelle est alors de déterminer s'il est possible d'obtenir un ratio de performance inférieur à $\left(2 - \frac{1}{m}\right)$, soit sur le même problème soit en utilisant des opérations plus puissantes pour réaliser l'ordonnancement, comme le tirage aléatoire ou la migration. Ce problème est étudié dans [29] où la proposition suivante est prouvée.

Théorème 1.3.2 *[29] Si les temps de communication ne sont pas pris en compte, $\left(2 - \frac{1}{m}\right)$ est une borne inférieure pour le ratio d'un algorithme d'ordonnancement déterministe avec ou sans migration.*
Un algorithme probabiliste sans préemption ne peut avoir un ratio de performance inférieur à $\left(2 - \frac{1}{\sqrt{m}}\right)$, ratio qui est obtenu par un algorithme de liste dans lequel les tâches affectées aux processeurs inactifs sont tirées au hasard parmi l'ensemble des tâches prêtes.

Seule la preuve de la première partie est ici présentée car elle utilise la construction d'un adversaire, technique fréquemment utilisée pour obtenir des bornes inférieures

sur un algorithme à la volée. Il s'agit de construire une instance du problème réalisant le pire cas et de montrer que ce pire cas ne peut être évité par aucun algorithme d'ordonnancement.

Le pire cas est obtenu pour l'instance suivante due à Graham [12]. G contient $1 + p(p-1)$ tâches indépendantes; une tâche α est de longueur p tandis que les β_k, $1 \leq k \leq p(p-1)$ autres sont de longueur 1. L'ordonnancement optimal est de longueur p; il est obtenu en exécutant α_1 sur un processeur et les $p(p-1)$ tâches β_k sur les $p-1$ autres processeurs.

Aucun algorithme d'ordonnancement à la volée ne peut faire d'hypothèse sur la longueur d'une tâche qui est inconnue tant que la tâche n'est pas terminée. La technique de l'adversaire consiste alors à faire en sorte que la la tâche α soit démarrée le plus tard possible. Chaque fois que l'ordonnanceur lance l'exécution d'une tâche sur un processeur, nous supposerons donc que cette tâche est une tâche β_k, de longueur 1. Ainsi l'ordonnanceur ne peut demander l'exécution de la tâche α qu'après que toutes les tâches β_k aient été exécutées, donc au plus tôt au top $(p-1)$. La longueur de l'ordonnancement délivré est alors de longueur au moins $(p-1)+p$, ce qui montre la borne inférieure. Il est clair que la préemption ou la migration ne peuvent améliorer ce ratio. □.

En conséquence, ni la migration ni l'introduction d'aléatoire ne peuvent améliorer le ratio de performance en comparaison d'un algorithme de liste.

1.3.3 Problème : $P|indep, p_j|C_{max}$

Si la durée des tâches est supposée connue, trier la liste (par exemple en ordonnançant en priorité les tâches les plus longues) ne permet pas d'améliorer la borne de Graham. Par contre, il est possible d'obtenir une amélioration en restreignant la classe de graphes considérés.

Ainsi si les tâches sont indépendantes le problème reste NP-complet (réduction au problème de la partition) mais l'heuristique LPTF – *largest processing time first* – (algorithme de liste avec la liste des tâches triée en ordre décroissant de durée) permet d'obtenir un ratio de $\frac{4}{3} - \frac{1}{3m}$ [13].

Une restriction plus intéressante est celle où toutes les tâches ont la même durée, ce sous problème étudié dans la section suivante servira de base de comparaison lors de l'étude de l'impact des communications.

1.3.4 Problème : $P|prec, p_j = 1|C_{max}$

Théorème 1.3.3 *Il n'existe pas d'algorithme d'ordonnancement pour le problème* $P|\ prec,\ p_j = 1|C_{max}$ *ayant un rapport de performance inférieur à* $\frac{4}{3}$ *si* $P \neq NP$.

Ce résultat est déduit du résultat suivant par le théorème de l'impossibilité :

Lemme 1.3.1 *[22] Le problème de décider pour un graphe de précédence quelconque s'il existe un ordonnancement valide de longueur au plus 3 est NP-complet.*

Nous allons utiliser une reduction du problème CLIQUE qui est un problème connu NP-complet. On rappelle qu'une *clique* un graphe dont tous les sommets sont adjacents. Il est évident que si $G_1 = (V_1, E_1)$ est une clique et si $G_2 = (V_2, E_2)$ a le même nombre de sommets que G_1 ($|V_1| = |V_2|$) mais n'est pas une clique, alors $|E_1| > |E_2|$. Le problème CLIQUE est le suivant :

- **Instance :** Un graphe non-orienté $G = (V, E)$ et un entier k.
- **Question :** Existe-t-il dans G une clique de taille k ?

Nous réduisons une instance de CLIQUE à une instance de $P|prec, p_j = 1|C_{max}$ de la manière suivante (la figure 1.1 illustre la réduction). Soit $l = \frac{(k-1)k}{2}$ le nombre d'arêtes dans une clique de taille k; Soit $k' = |V| - k$ et $l' = |E| - l$.
On considère l'instance suivante de $P|prec, p_j = 1|C_{max}$:

- Le nombre de machines est $m = \max\{k, l + k', l'\} + 1$.
- Pour chaque sommet $v \in V$, on introduit une tâche T_v et pour chaque arête $e \in E$ une tâche L_e. On a $T_v \prec L_e$ si v est une extremité de e.
- On ajoute $(3m-l-k-l'-k')$ tâches $X_x (x = 1, ..., m-k), Y_y (y = 1, ..., m-l-k')$ et $Z_z (z = 1, ..., m - l')$ avec les précédences $X_x \prec Y_y \prec Z_z$, $\forall x, y, z$.

Le nombre total de tâches est $3m$. Un ordonnancement valide avec $C_{max} = 3$ ne peut donc pas avoir de temps d'inactivité.

- Supposons que G contient une clique de taille k, alors il existe un ordonnancement valide de longueur 3 (voir figure 1.1).
- Supposons qu'on a un ordonnancement valide de longueur 3, alors G contient une clique de taille k.

Il est évident que les tâches qui appartiennent à des chemins de longueur 3 doivent être exécutées au plus tôt. Ainsi X_x, Y_y et Z_z sont exécutés aux instants 1, 2 et 3 respectivement.

A l'instant 1 on peut exécuter au plus k tâches de type T. Supposons que seulement $k_1 < k$ tâches forment une clique. Ceci dit qu'il y aura certains processeurs qui vont rester inactifs pendant la deuxième unité de temps puisque les tâches de type L (c'est-à-dire les arêtes adjacentes aux k sommets exécutés à l'instant 1) qui sont liberées sont moins de l. Par conséquent, l'ordonnancement ne peut pas terminer à $t = 3$. □

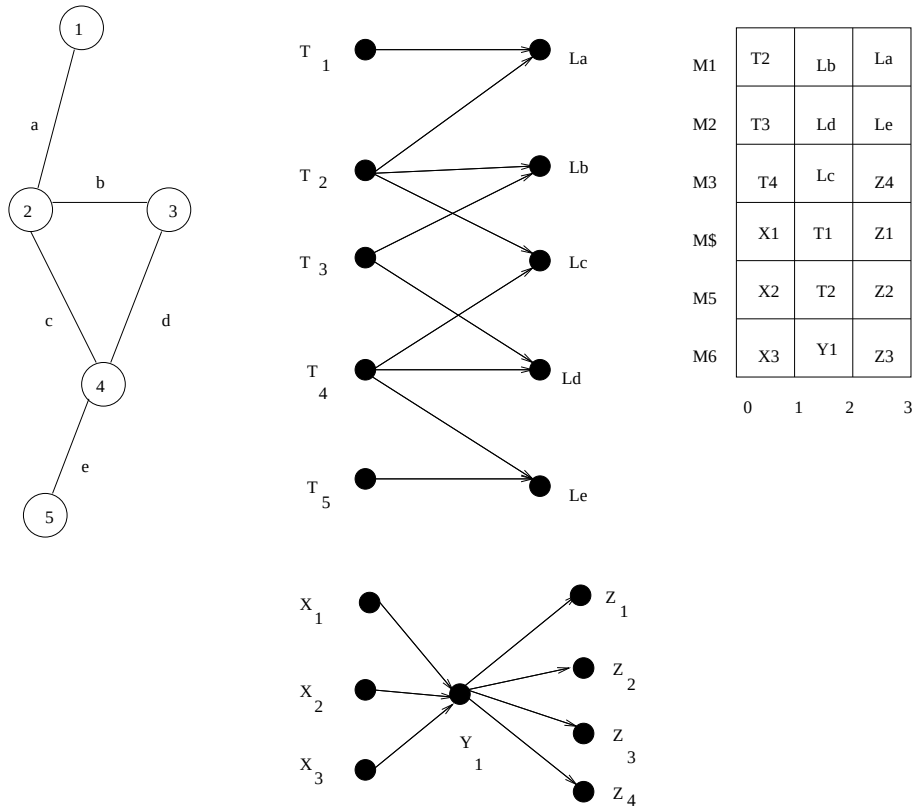

Figure 1.1 : Le graphe de précédence des tâches, l'instance correspondante du problème d'ordonnancement où $m = 6$, et un ordonnancement en trois unités de temps.

1.3.5 Une borne supérieure pour $P|prec, p_j = 1|C_{max}$: l'algorithme de Coffman-Graham

La solution du problème utilise un algorithme d'étiquettage qui associe à chaque tâche du graphe un entier. L'étiquettage se fait de la façon suivante [18]. R désigne l'ensemble des tâches non étiquettées dont tous les successeurs immediats sont déjà étiquettés.

- **Etape 1** : On distribue de façon arbitraire des étiquettes aux tâches qui n'ont pas de successeurs.
- **Etape 2** : A chacune des tâches de R on fait correspondre la séquence décroissante des étiquettes de ses successeurs immédiats S_i. La tâche qui a la plus petite (lexicographiquement) séquence S_i prend l'étiquette suivante.

- **Etape 3 :** On itére l'étape 2 jusqu'à ce que toutes les tâches de G soient étiquettées.
- **Etape 4 :** Construction d'une liste de priorité en ordre décroissant des étiquettes.

Théorème 1.3.4 *[21] L'algorithme de CG a un ratio de performance borné par*

$$2 - \frac{2}{m}$$

La preuve assez technique de ce résultat est omise. Elle permet d'obtenir le corollaire suivant.

Corollaire 1.3.2 *L'algorithme de CG est optimal pour* $P2|prec, p_j = 1|C_{max}$.

Etant donné que la complexité en temps de l'algorithme de CG est en $O(n^2)$, il est évident qu'on a intérêt de l'utiliser seulement pour des petites valeurs de m. Pour des grandes valeurs de m les gains par rapport à un algorithme de liste quelconque deviennent minimes.

Remarque. La complexité du problème $P_m|prec, p_j = 1|C_{max}$ n'est pas connue quand $m \geq 3$.

1.4 Ordonnancements dynamiques avec communications

Nous nous intéressons à la mise en œuvre d'un algorithme de liste sur une architecture distribuée qui fournit un ordonnancement de longueur presque optimale sur un nombre arbitraire de processeurs identiques (cf corollaire 1.3.1). L'intérêt d'un tel algorithme est d'être compatible avec une création dynamique du graphe de dépendance : une tâche t peut créer lors de son exécution de nouvelles tâches t_k, en nombre quelconque, en ajoutant de nouvelles dépendances. Il est donc particulièrement adapté à des applications irrégulières où la génération de parallélisme est souvent réalisée en cours d'exécution.

Deux problèmes sont alors à considérer :

- l'implémentation de l'algorithme de liste lui-même : gestion du graphe et contrôle de l'ordonnancement par affectation de tâches prêtes aux processeurs inactifs.
- la prise en compte du coût des communications.

Les résultats présentés sont appliqués à l'ordonnancement de programmes dont le nombre d'opérations est grand devant le nombre de tâches et de dépendances (*granularité globale*).

Remarque : Hypothèse sur la construction à la volée du graphe de tâches. Pour faciliter la gestion du graphe de dépendance (qui induit une relation de précédence entre les tâches), nous supposerons que :

- les tâches t_k créées ont une relation de précédence avec la tâche t qui les crée, c'est à dire : $\forall k$, si t crée t_k alors $t \prec t_k$;
- les nouvelles dépendances de données ajoutées au graphe ne peuvent concerner que les tâches ayant préalablement une relation de précédence avec t, c'est à dire : soit t_1 et t_2 deux tâches et $t_1 \prec t_2$ une dépendance ajoutée par t, alors, avant l'ajout de cette dépendance, on avait $t \prec t_1$ et $t \prec t_2$.

Ces contraintes, bien que non restrictives en ce qui concerne la description de l'exécution d'un algorithme sous forme de graphes de tâches (cf les chapitres sur la construction de graphes de tâches), ne sont pas fondamentales : leur introduction est motivée par l'obtention d'une implémentation concise en ce qui concerne l'algorithme de listes.

1.4.1 Coût des communications : simulation d'une mémoire partagée

Pour prendre en compte le coût des communications (contention et latence), nous simulons une mémoire partagée. La simulation utilise des fonctions de hachage (choisies aléatoirement dans une classe de fonctions de hachage universelles) pour distribuer les objets accessibles par tous les processeurs (i.e. les transitions dans le graphe de dépendance) sur les différents modules mémoire [28, 20].

Le délai d'une simulation est le temps requis pour l'accès à une donnée de taille unité. Il est lié à l'évaluation de la fonction de hachage, à la contention mémoire (quand plusieurs accès sont effectués vers un même module) et à la latence (temps de routage). Dans [28], une simulation de délai $\Theta(\log p)$ sur un réseau "butterfly" est donnée pour des accès exclusifs (de type EREW). Dans [19], une simulation de délai $O(\log \log p \log^\star p)$ est donnée pour des accès aléatoires (de type EREW ou CRCW) sur une architecture disposant d'un réseau d'inteconnexion complet (la latence n'est pas prise en compte); pour des accès concurrents (CRCW), la simulation est à un facteur $\log^\star p$ de l'optimal.

Pour obtenir une simulation optimale, ces délais doivent être masqués par des calculs. La technique classique (virtualisation ou "parallel slackness" [20, 31, 19]) consiste à simuler q ($q > p$) processeurs virtuels sur les p processeurs de l'architecture physique, en utilisant généralement des processus légers. La simulation est alors dite optimale si le délai pour un accès est proportionnel à q/p.

Dans la suite de cette section, nous supposerons disposer d'une simulation optimale de délai h; le nombre de processeurs m requis pour une telle simulation peut

alors être supérieur au nombre de processeurs p de l'architecture. L'ordonnancement sera donc effectué sur m processeurs et non p, mais comparé à l'ordonnancement optimal sur p processeurs. Un processeur (parmi les m) peut lire ou écrire un objet dans n'importe quel module mémoire distant. Le coût de l'accés à une donnée de taille n est supposé borné par $h.n$ où h est une constante de la machine.

Synchronisation. Les accès en mémoire distante ne permettent pas directement de synchroniser les processeurs (à la différence des communications). Sur une architecture distribuée asynchrone, des outils spécifiques de synchronisation (verrous, sémaphores) doivent alors être utilisés. Nous supposerons aussi que le coût d'accés à un verrou global, lorsque celui-ci est libre, est lui aussi borné par h (il requiert un aller-retour).

1.4.2 Problème $P_m|prec?, p_i =?|C_{max}$: algorithme de liste

Lorsque le graphe de dépendance est construit et déterminé en cours d'exécution, l'implémentation d'un algorithme de liste nécessite de gérer une structure permettant de déterminer les tâches prêtes (i.e. dont les prédécesseurs sont terminés) et d'affecter des tâches distinctes à chacun des processeurs devenus inactifs.

Directe en séquentiel, cette affectation est cependant complexe en distribué. Dans [7], un algorithme basé sur des graphes d'expansion est utilisé pour ordonnancer n tâches indépendantes effectuant au total $T_1 = O(n)$ opérations en temps $O(\log n)$ sur $\frac{n}{\log n}$ processeurs d'une EREW PRAM; outre le caractère théorique d'un tel ordonnancement (qui suppose une synchronisation globale à chaque instruction), la constante masquée dans le O est loin d'être négligeable (> 237) devant le ratio de performance (proche de 2)[1].

Dans ce qui suit, nous considérons dans un but simplificateur une gestion séquentielle de la structure permettant de calculer les tâches prêtes (i.e. la liste) et de les affecter aux processeurs inactifs. À défaut d'être la plus performante sur un grand nombre de processeurs, elle permet d'estimer précisément le surcoût lié à la gestion de l'ordonnancement. Pour éviter de dédier un processeur à la gestion de cette structure, elle est stockée en mémoire (virtuellement) partagée et accédée par les processeurs en exclusion mutuelle; l'exclusion est réalisée par un verrou global.

Théorème 1.4.1 *Soit un programme paralléle dont l'exécution génère n tâches et n_d dépendances (transitions). Soit T_1 le temps d'une exécution séquentielle (sur un processeur) et T_∞ le temps minimal sur un nombre infini de processeurs. Alors, si les communications ne sont pas prises en compte, G peut être ordonnancé sur m*

[1] Une constante beaucoup plus faible peut être obtenue si l'ordonnancement est réalisé sur moins de processeurs. Par exemple, l'utilisation d'un calcul de préfixe permet de calculer un ordonnancement à un facteur 2 de l'optimal sur $\frac{n}{\log^2 n}$ processeurs [6].

processeurs identiques en temps T_m borné par :

$$T_m \leq \frac{T_1}{m} + T_\infty + O(n + n_d + m).$$

Comme pour le théorème 1.3.1, la preuve est basée sur la construction d'un ensemble de tâches situées sur un chemin critique dans G. Pour contrôler l'ordonnancement, nous utilisons une structure globale, visible par tous les processeurs et accédée en exclusion mutuelle grâce à un verrou global `verrou-graphe`; outre le graphe G, cette structure contient :

- la liste L_t des tâches prêtes (initialisée avec les tâches de G de degré entrant nul);
- la liste L_i des processeurs inactifs; à l'initialisation, seul un processeur (disons P_0) est supposé artficiellement actif – il simule la terminaison d'une tâche fictive –, les autres $m-1$ processeurs étant mis dans L_i.

L'ordonnancement des tâches est géré de la façon suivante : lorsqu'un processeur p termine une tâche t (par exemple P_0 au top 0), il exécute les instructions de *post-traitement* liées à t suivantes :

1. Prendre le verrou `verrou-graphe`.
2. Mettre à jour G en ajoutant les nouvelles tâches créées et les dépendances associées.
3. Mettre à jour la liste L_t des tâches prêtes en ajoutant les tâches successeurs immédiats de t dont tous les prédécesseurs sont terminés.
4. Tant que ($L_i \neq$ vide) et ($L_t \neq$ vide) faire
   Enlever une tâche t' de L_t et un processeur p' de L_i;
   Lancer l'exécution de t' sur p'.
5. Si ($L_t \neq$ vide) alors
   Enlever une tâche t'
   Relâcher le verrou `verrou-graphe`;
   Exécuter t' (sur le processeur p)
6. sinon
   Ajouter p à la liste des processeurs inactifs
   Relâcher le verrou `verrou-graphe`;

De manière évidente, l'ordonnancement ainsi construit est admissible, i.e. ne viole aucune contrainte de précédence dans G. Soit T_m le temps d'exécution avec cet ordonnancement sur m machines. En reprenant le schéma de la preuve précédente (1.3.1), il s'agit de borner les tops d'inactivité *potentielle*. Les tops $\{1, ..., T_m\}$ sont partitionnés en trois sous-ensembles, A (activité), O (ordonnancement) et I (inactivité) de la façon suivante :

- à tout top dans A, tous les processeurs exécutent une tâche, i.e. une instruction de l'application.
- à tout top dans O, une instruction de post-traitement (i.e. gestion de l'ordonnancement) est en cours.
- à tout top dans I, la liste L_i des processeurs inactifs est non vide mais aucune instruction de post-traitement n'est en cours d'exécution.

En remarquant que, lorsqu'un processeur est en attente du verrou, une instruction de post-traitement est nécessairement en cours sur un autre processeur, il n'y a pas dans $A \bigcup C$ de tops où un processeur est en attente du verrou.

$\#A$ peut être trivialement borné par $\frac{T_1}{m}$. L'exécution en exclusion mutuelle des instructions de post-traitement permet de borner facilement $\#O$. En effet, une fois le verrou pris, le post-traitement ne consiste qu'en la mise à jour des listes (les opérations de base sont en coût constant) et au lancement de nouvelles exécutions éventuelles. Lorsque le verrou est libre, le temps nécessaire à sa prise est borné par h. Lorsqu'il n'est pas libre, il existe nécessairement un autre processeur en train d'exécuter une instruction de post-traitement (éventuellement en train de prendre ou de relâcher le verrou). Par suite, le nombre de tops dans $\#O$ est borné par $O(h(n + n_d + m))$ (i.e. la taille du graphe).

$\#I$ est borné de manière analogue à la preuve de Graham, en remarquant que lorsqu'un processeur est inactif et aucun en cours de post-traitement, il existe nécessairement une tâche sur un chemin critique qui est en cours d'exécution. Par suite, $\#I < T_\infty$.

Pour conclure, $mT_m = m\#O + m\#A + (m-1)\#I$. D'où[2] $T_m \leq \frac{T_1}{m} + T_\infty + O(n + n_d + m)$. □

Conclusion sur le surcoût d'ordonnancement. Borner le surcoût requiert d'analyser l'algorithme qui contrôle l'ordonnancement. Dans tous les cas, ce surcoût fait apparaître un terme lié à la taille du graphe de dépendance. Pour le diminuer, on peut s'intéresser à structurer ce graphe; mais les remarques énoncées à la section précédente montre que celà ne peut qu'entraîner une perte en terme de ratio.

[2]Le ratio $(2 - 1/m)$ de Graham se retrouve directement dans le cas où $\#O = 0$.

Implémentation d'un algorithme de liste. En pratique, la gestion centralisée des listes et l'accès en exclusion mutuelle est évité. D'une part, le graphe est stocké de manière distribuée; d'autre part, lorsqu'un processeur est inactif, il scrute les autres processeurs pour trouver une nouvelle tâche à exécuter (on parle de "vol" de tâche). Une telle stratégie est analysée en moyenne d'un point de vue théorique dans [5, 17] et permet d'obtenir un résultat similaire à 1.4.1.

1.4.3 Problème $P_m|prec?, p_i =?, com?|C_{max}$: communications

L'algorithme précédent ne prend pas en compte le délai entre l'affectation d'une tâche prête à un processeur et le moment où elle démarre son exécution, les données dont elle dépend étant disponibles localement sur ce processeur. De part la technique de simulation utilisée pour la mémoire partagée (§1.4.1), ce délai est borné par h fois le volume des données accédées. Le résultat s'étend donc directement.

Théorème 1.4.2 *Soit une machine simulant m processeurs identiques accédant avec un délai h à une mémoire virtuellement partagée.*
Soit un programme paralléle dont l'exécution génère n tâches et n_d dépendances (transitions). Soit T_1 le temps de l'exécution séquentielle (sur un processeur), W_c le volume total maximal d'accès distants et T_∞ le temps minimal sur un nombre infini de processeurs (avec communications). Alors, G peut être ordonnancé sur m processeurs identiques en temps T_m borné par :

$$T_m \leq \frac{T_1}{m} + h\frac{W_c}{m} + T_\infty + hO(n + n_d + m).$$

Le résultat s'obtient à partir de la preuve précédente (théorème 1.4.1) en bornant $m\#A$ par $T_1 + W_c$ et $\#I$ par T_∞ (qui prend en compte les communications sur le chemin critique). □

1.4.4 Granularité globale : $P_p|prec?, p_i =?, com? < \alpha \sum p_i|C_{max}$

Le théorème précédent ne permet pas d'évaluer le ratio de performance par rapport à l'ordonnancement optimal. D'une part, ce dernier peut éventuellement supprimer toutes les communications et d'autre part, la simulation de délai optimal utilisé requiert un nombre de processeurs m supérieur au nombre de processeurs p physiques. Le temps effectif sur p processeurs est alors $\frac{m}{p}T_m$; il ne peut être garanti à un facteur constant de l'optimal que si hW_c négligeable devant T_1.

Lorsque le volume maximal de communications est borné par αT_1 (problème $P_p|prec?, p_i =?, com? < \alpha \sum p_i|C_{max}$), l'algorithme de liste précédent (théorème 1.4.2) permet d'obtenir un ordonnancement de longueur bornée par :

$$\frac{m}{p}\left((1 + \alpha h)\frac{T_1}{m} + T_\infty + hO(n + n_d + m)\right).$$

Si pour une entrée de taille suffisante, les valeurs de α (borne supérieure du rapport entre volume de communications et volume de calcul), T_∞ (temps parallèle minimal) et $(n+n_d)$ (nombre de tâches et de dépendances) sont asymptotiquement négligeables devant le temps séquentiel d'exécution, le ratio de performance est asymptotiquement optimal (i.e. 1). D'où le théorème suivant.

Théorème 1.4.3 *Soit un réseau de processeurs constitué de p processeurs identiques communiquants via un réseau d'interconnexion.*
Soit π un programme parallèle dont l'exécution construit un graphe de dépendance avec un volume de dépendances de données asymptotiquement négligeable devant le temps séquentiel T_1 et un nombre d'arêtes de dépendances asymptotiquement négligeable devant le temps parallèle sur un nombre infini de processeurs. Alors π peut être exécuté en temps T_p asymptotiquement optimal par un algorithme de liste :

$$T_p \leq (1+\epsilon)\frac{T_1}{p}.$$

On obtient donc un ordonnancement asymptotiquement optimal pour des problèmes très parallèles et requérant peu de communications. Ce cadre est plus général que celui de la granularité grossière (certaines tâches pouvant requérir un temps de communication grand devant le temps de calcul [10]). Il est à noter que la création dynamique de tâches et la gestion du graphe n'implique pas de surcoût pour autant que la taille du graphe des tâches reste négligeable devant le temps parallèle optimal.

1.5 Ordonnancements statiques avec communications

En reprenant le mème modèle de communication qu'au paragraphe 1.4.1, nous considérons que le temps de communication entre deux tâches ayant une dépendance directe est nul si les deux tâches sont exécutées sur le même processeur et est de durée constante et connue (proportionnelle à la taille de la donnée communiquée) sinon. Soit d_i la date de début de la tâche t_i sur le processeur P_i; si (i,j) est un arc du graphe de précédence et P_i le processeur qui exécute la tâche i, alors

$$\begin{cases} d_j - d_i \geq p_i, & si \quad P_i = P_j \\ d_j - d_i \geq p_i + c_{ij}, & si \quad P_i \neq P_j \end{cases}$$

où c_{ij} est le coût de la communication interprocesseur entre les tâches i et j.

1.5.1 Granularité grossière : $\bar{P}|prec, c=1, p_j=1|C_{max}$

Théorème 1.5.1 *Le problème de décider si une instance de $\bar{P}|prec, c=1, p_j=1|C_{max}$ a un ordonnancement de longueur au plus 6 est NP-complet.*

La preuve est basée sur une transformation par le problème 3-SAT (voir [16]).

Corollaire 1.5.1 *Il n'existe pas d'heuristique ρ-approchée telle que* $\rho \leq \frac{7}{6}$.

Ainsi nous savons que nous sommes dans l'impossibilité de trouver une heuristique qui permette de garantir que quelque soit le graphe en donnée la solution obtenue par l'heuristique est à moins de 16% de la solution optimale.

Ce problème avec $c = 0$ était un problème facile. Une heuristique triviale pour $c = 1$ est d'intercaler entre 2 étapes de calcul une phase de communication. Cette heuristique a trivialement un ratio égal à deux. Récemment dans [26], les auteurs donnent une heuristique de ratio $\frac{4}{3}$. Cette heuristique est basée sur la résolution d'un problème de programmation linéaire.

En effet, soit G=(V,E) le graphe représentant une instance du problème $\bar{P}|prec, c = 1, p_j = 1|C_{max}$. Le problème peut-être formulé par le problème linéaire en nombres entiers suivant :

$$Min \quad w$$
$$\forall i \in V, \quad t_i \geq 0$$
$$\forall (i,j) \in E, \quad x_{(i,j)} \in \{0,1\} \quad (*)$$
$$\forall (i,j) \in E, \quad t_i + 1 + x_{(i,j)} \geq t_j$$
$$\forall i \in V, \quad \sum_{j \in \Gamma^+(i)} x_{(i,j)} \geq |\Gamma^+(i)| - 1$$
$$\forall i \in V, \quad \sum_{j \in \Gamma^-(i)} x_{(j,i)} \geq |\Gamma^-(i)| - 1$$
$$\forall i \in V, \quad t_i + 1 \leq w$$

Avec cette formulation $x_{(i,j)} = 0$ implique que les tâches i et j sont exécutées sur le même processeur. Lorqu'on relâche les contraintes d'intégrité (c'est-à-dire on remplace $(*)$ par $\forall (i,j) \in E, \quad x_{(i,j)} \in [0,1]$) on obtient un problème de programmation linéaire qui peut être résolu en temps polynômial.

Intuitivement, dans la solution réelle obtenue plus $x_{(i,j)}$ est proche de 0 plus on a intérêt à supprimer la communication entre la tâche i et la tâche j. Dans [26], les auteurs montrent qu'effectivement en mettant à 0 tous les $x_{(i,j)}$ strictement inférieurs à $\frac{1}{2}$ et à 1 les autres, on obtient une solution réalisable à moins de 33% de la solution réelle (qui est elle-même une borne inférieure de la solution optimale).

Théorème 1.5.2 *L'heuristique proposée est un algorithme* $\frac{4}{3}$*-approché.*

Dans [8], il est montré que ce problème devient facile si on autorise la duplication des tâches.

1.5.2 Le problème $P|prec, c = 1, p_j = 1|C_{max}$

Théorème 1.5.3 *Le problème de décider si une instance de $P|biparti, c = 1, p_j = 1|C_{max}$ a un ordonnancement de longueur au plus 4 est NP-complet.*

La preuve est omise; elle utilise une réduction du problème de la clique [1].

Ainsi nous savons que nous sommes dans l'impossibilité de trouver une heuristique qui permettent de garantir que quelque soit le graphe en donnée la solution obtenue par l'heuristique est à moins de 25% de la solution optimale pour le problème $P|prec, c = 1, p_j = 1|C_{max}$.

Une solution classique pour obtenir un ordonnancement sur un nombre limité de processeurs est de partir d'un ordonnancement sur un nombre illimité de processeurs et de plier cet ordonnancement sur le nombre de processeurs considéré. Le théorème suivant permet d'évaluer une telle démarche.

Théorème 1.5.4 *A partir de n'importe quelle heuristique de rapport de performance ρ pour le problème $\bar{P}|prec, c = 1, p_j = 1|C_{max}$ on peut obtenir une heuristique pour le problème $P|prec, c = 1, p_j = 1|C_{max}$ de rapport de performance $1 + \rho$.*

Soit l'heuristique h suivante : on exécute les tâches dans le même ordre que pour l'heuristique, notée h^* dans la suite, utilisée pour le problème $\bar{P}|prec, c = 1, p_j = 1|C_{max}$. Plus précisément, si X_i est l'ensemble des tâches exécutées à l'instant i avec l'heuristique h^*, et β_i le cardinal de X_i, alors dans h, si $\beta_i \neq 0$, on exécute les tâches de X_i en $\lceil \frac{\beta_i}{m} \rceil$ instants où m est le nombre de processeurs. Dans le cas où $\beta_i = 0$, on conserve l'instant d'inactivité.

On remarque que le nombre d'instants où il y a au moins un processeur inactif est inférieur à la durée de l'ordonnancement avec un nombre illimité de processeurs. Les autres instants, tous les processeurs sont actifs. Ainsi l'écart entre le temps donné par l'heuristique et le temps optimal est borné par la durée de l'ordonnancement avec un nombre illimité de processeurs. Si l'on note t^* la durée de l'ordonnancement donné par h^* et t^m la durée de l'ordonnancement donné par h alors $t^m \leq t^* + t^m_{opt}$. Comme par hypothèse $t^* \leq \rho t^*_{opt}$ et comme $t^*_{opt} \leq t^m_{opt}$, la conclusion s'impose. □

Le corollaire immédiat de ce théorème est l'existence d'une heuristique de ratio de performance $\frac{7}{3}$ pour le problème $P|prec, c = 1, p_j = 1|C_{max}$. Cette heuristique est la meilleure connue à l'heure actuelle.

Dans les sections suivantes nous allons citer quelques familles de graphes ayant un comportement spécifique vis-à-vis de ce modèle.

1.5.3 Le problème $P2|arbre, p_j = 1, c = 1|C_{max}$

Une famille des graphes de précédence qui a un intérêt important est la famille des arbres. Depuis 1961, on sait que le problème $P|arbre, p_j = 1|C_{max}$ est un problème polynômial. En fait, T.C. Hu a proposé un algorithme simple qui consiste à

construire une liste de priorité où la tâche dont la distance de la racine (le niveau) de l'anti-arbre est la plus grande, est la tâche la plus prioritaire.

Par contre, quand on prend en compte les communications Lenstra et al [24, 33] ont montré que le problème $P|arbre, p_j = 1, c = 1|C_{max}$ devient NP-complet. Lorsqu'on fixe le nombre de processeurs, un algorithme optimal de programmation dynamique de complexité $O(n^{2(m-1)})$ a été présenté dans [32].

Restrictions sur la structure du graphe. Parmi les autres classes de graphes pour lesquels un ordonnancement optimal est facilement calculable figure les graphes ordonnés par intervalles. Un graphe de dépendance est dit ordonné par intervalles si à toute tâche t peut être associé un intervalle réel I_t tel que $t \prec t'$ si et seulement si $\forall x \in I_t$, $\forall y \in I_{t'}$ on a $x < y$. Por de tels graphes, le calcul d'un ordonnancement optimal peut être ramené à un tri [27].
Étant donné un graphe quelconque, le problème est alors d'ajouter un nombre minimal de dépendances qui, sans limiter de manière trop importante le parallèlisme, permettent de le ramener à une famille de graphes pour laquelle le calcul de l'ordonnancement optimal est facile.

1.5.4 Granularité fine : $P|prec, c, p_j = 1|C_{max}$

Sur ce modèle, les résultats connus sont très décevants et de nombreux travaux théoriques et pratiques sont en cours.

Les meilleurs résultats connus pour la borne inférieure du ratio de performance est $1 + \frac{1}{c+3}$ [2]. Dès que c devient grand et donc que la granularité est relativement fine, cette borne ne représente aucun intérêt. Réciproquement les meilleures heuristiques connues donnent un ratio de performance égal à $c + 2$. Là aussi, la borne ne représente aucun intérêt puisque ce ratio est le même qu'une heuristique triviale consistant à prendre un ordonnancement négligeant les communications et à introduire entre deux étapes de calcul une inactivité des processeurs de durée c pour permettre les communications.

1.5.5 Tableau récapitulatif

Le tableau ci-dessous résume les différents résultats pour l'ordonnancement statique lorsque les tâches sont de même durée ($p_i = 1$) et les communications de durée c.

	m entrée		m fixé	m suffisant	
	B_{inf}	B_{sup}		B_{inf}	B_{sup}
c = 0	$\frac{4}{3}$	2	m = 2 (Classe P) m = 3 (Pb Ouvert)	1	1
c = 1	$\frac{5}{4}$	$\frac{7}{3}$	m = 2 (Pb Ouvert)	$\frac{7}{6}$	$\frac{4}{3}$
c grand	$1+\frac{1}{c+3}$	c + 2	m = 2 (NP-Complet)	1	1 + c

1.6 Systèmes hétérogènes

Le rôle croissant que joue les réseaux de stations de travail dans le domaine du calcul parallèle, nous oblige à intégrer ce paramètre dans nos modèles. La plupart de ces réseaux sont des réseaux constitués de stations qui n'ont pas tous la même puissance et il est évident que cette hétérogénéité pose de nouveaux problèmes sur le plan de la modélisation et de l'analyse des problèmes d'ordonnancement associés.

Dans le cas général, même si l'on ne considère pas les coûts de communication, le temps d'une tâche ne peut être connu qu'après sa terminaison et dépend non seulement du processeur auquel elle a été affectée mais encore de l'état de ce processeur durant son exécution. Le problème apparaît donc complexe à modéliser. Le but de cette section est d'étudier cette complexité même sous des hypothèses simplificatrices.

1.6.1 Difficulté d'un ordonnancement optimal

Dans cette section nous nous limitons au cas de systèmes comportant une machine puissante, que l'on va noter P_1 et un nombre illimité de machines de faible puissance, que nous appelerons de type P_2. Ce problème est noté $P_1 \cup \bar{P}_2 | prec, p_i^1, p_i^2 | C_{max}$ où p_i^1 (resp. p_i^2) représente la durée d'exécution sur P_1 (resp. sur un processeur de type P_2).

Un ordonnancement consiste ici en deux types d'information :

- l'affectation des tâches au processeur (choix du type de processeur)
- l'ordre d'exécution sur chaque processeur (ici sur P_1)

En fait, il a été montré que même si on se limite à des graphes bipartis de profondeur 1, le problème est déjà NP-difficile. Même lorsque l'affectation des tâches au processeur est imposée le problème reste NP-complet dans des versions très simples (par exemple lorsque le graphe est une réunion de chemins de longueur 3). Pour plus de détails voir [2].

1.6.2 Machines uniformes et non-uniformes

Lorsque la durée d'une tâche est inconnue mais ne dépend que du processeur sur lequel elle a été affectée, deux modèles de machine sont distingués (m désigne le nombre de processeurs de la machine) :

- machine *uniforme* : les processeurs sont de vitesses proportionnelles, le rapport des vitesses étant éventuellement inconnu).
- machine *non-uniforme* : le temps d'une tâche dépend du processeur sur lequel elle s'exécute.

Dans ces deux cas, $\Omega(\log m)$ est une borne inférieure pour le ratio de perfomance d'un algorithme d'ordonnancement dynamique avec ou sans préemption [29, 15].

Par ailleurs, des algorithmes polynomiaux ont été proposés pour ces deux modèles qui permettent d'atteindre un ratio $O(\log n)$, n étant le nombre de tâches [23]. En regroupant les tâches affectées à un même processeur, [29] donnent un algorithme d'ordonnancement dynamique de ratio $O(\log \min(m, R))$ où R le rapport de vitesse entre le processeur le plus rapide et le plus lent. Il est à noter que tous ces ratio peuvent être obtenus par des algorithmes d'ordonnancement non-préemptifs; la migration n'apporte un gain que d'un facteur constant dans le ratio.

1.6.3 Simulation d'une machine à processeurs identiques

Une approche pragmatique pour construire des algorithmes ayant une justification théorique (par exemple un algorithme de liste) est de simuler une machine constituée de processeurs identiques sur une architecture hétérogène. Une telle simulation est facilitée par la manipulation de processus légers dont le nombre sur chaque processeur peut varier en fonction de l'état du processeur.

1.7 Conclusion

Les différents résultats énoncés dans ce chapitre montrent la difficulté d'obtenir des résultats analytiques non triviaux lorsqu'on tente d'intégrer dans les modèles d'ordonnancement certains paramètres primordiaux d'un point de vue pratique comme les grands délais de communication. Dans le cas général, il n'existe pas d'algorithmes d'ordonnancement ayant un ratio de performance constant (indépendant de l'architecture) sur une architecture hétérogène. Il est donc important de considérer des classes spécifiques de problèmes à ordonnancer.

Une approche harmonieuse semble être de se restreindre à des graphes de dépendance (éventuellement construits dynamiquement en fonction des entrées) dont le volume maximal de communication est négilgeable devant le volume de calcul. Aussi bien dans le cadre dynamique (algorithmes de liste) que statique (granularité

grossière, $p_i = c = 1$), des ordonnancements efficaces sont alors connus. Cependant, l'obtention d'un tel graphe par compilation à partir d'un programme de grain fin nécessite des algorithmes de partition efficaces qui risquent de réduire fortement le degré de parallélisme sur des problèmes ayant un comportement irréguliers.

Bibliographie

[1] Anderson (R.) et Miller (G.). – Deterministic parallel list ranking. *In : AWOC88*. pp. 81–90. – Springer-Verlag.

[2] Bampis (E.), Giannakos (A.) et Konig (J.-C.). – *From simple-machine problems to hetegogeneous scheduling.* – Rapport technique n° 25, LaMI, Université d'Evry, France, 1997.

[3] Blazewicz (J.), Exker (K.), Schmidt (G.) et Węglarz (J.). – *Scheduling in Computer and Manufacturing Systems.* – Germany, Springer-Verlag, 1993.

[4] Blelloch (G. E.), Gibbons (P. B.) et Matias (Y.). – Provably efficient scheduling for languages with fine-grained parallelism. *In : Proceedings of the 7th Symposium on Parallel Algorithms and Architectures*. pp. 1–12. – Santa-Barbara, California, 1995.

[5] Blumofe (R. D.). – *Executing Multithreaded Programs Efficiently.* – Boston, Thèse de PhD, Massachussets Institute of Technology, 1995.

[6] Briat (J.), Gautier (T.) et Roch (J.-L.). – Application irrégulière et ordonnancement en ligne. *In : Placement dynamique et répartition de charge: application aux sy stèmes répartis et parallèles*, éd. par Bernard (G.), Chassin de Kergommeaux (J.), B. (F.) et Roucairol (C.), pp. 81–106. – Collection didactique INRIA, 1996.

[7] Cole (R.) et Vishkin (U.). – Approximate Parallel Scheduling. Part I : The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM Journal on Computing*, vol. 17, n° 1, 1988.

[8] Colin (J.) et Chretienne (P.). – C.P.M. Scheduling with small communication delays and task dupli cation. *Operations Research*, vol. 39, n° 3, 1991, pp. 680–684.

[9] Culler (D. E.), Karp (R. M.), Patterson (D.), Sahay (A.), Santos (E. E.), Schausser (K. E.) et Ramesh Subramonian (T. v. E.). – LogP: A Practical Model of Parallel Computation. *Communications ACM*, vol. 39, n° 11, 1996, pp. 78–85.

[10] Gautier (T.), Roch (J.) et Villard (G.). – Regular versus irregular problems and algorithms. *In : Proc. of IRREGULAR'95, Lyon, France.* – Springer-Verlag.

[11] Gibbons (P.). – A more practical PRAM model. *In : Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures.*

[12] Graham (R.). – Bounds for Certain Multiprocessor Anomalies. *Bell System Tech J.*, vol. 45, 1966, pp. 1563–1581.

[13] Graham (R.). – Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, vol. 17, n° 2, 1969, pp. 416–426.

[14] Graham (R.), Lawler (E.), Lenstra (J.) et Rinnooy Kan (A.). – Optimization and approximation in deterministic scheduling: A survey. *Ann. Disc. Math.*, vol. 5, 1979, pp. 287–326.

[15] Hall (L. A.), Schulz (A. S.), Shmoys (D. B.) et Wei n (J.). – *Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms.* – Rapport technique n° 516/1996, Berlin, Technishe Universität, 1996.

[16] Hoogeveen (J. A.), Lenstra (J.) et Veltman (B.). – Three, four, five, six or the complexity of scheduling with communication delays. *Operations Research Letters*, vol. 16, 1994, pp. 129–137.

[17] Joerg (C.). – *The Cilk system for parallel multithreaded computing.* – Thèse de PhD, Massachussets Institute of Technology, january 1996.

[18] Jr. (C. E.) et Graham (R.). – Optimal scheduling for two-processor systems. *Acta Informatica*, no1, 1972, pp. 200–213.

[19] Karp (R. M.), Luby (M.) et auf der Heide (F. M.). – Efficient PRAM Simulation on a Distributed Memory Machine. *Algorithmica*, vol. 16, 1996, pp. 517–542.

[20] Kruskal (C. P.), Rudolph (L.) et Snir (M.). – A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, vol. 71, 1990, pp. 95–132.

[21] Lam (S.) et Sehti (R.). – Worst case analysis of two scheduling algorithms. *SIAM J. Computing*, vol. 6, n° 3, 1977, pp. 518–538.

[22] Lenstra (J.) et Rinnooy Kan (A.). – Complexity of scheduling under precedence constraints. *Operations Research*, vol. 26, 1978, pp. 22–35.

[23] Lenstra (J.), Shmoys (D.) et Tardos (.). – Approximation algorithms for scheduling unrelated parallel machines. *Math. Programming*, vol. 46, 1990, pp. 259–271.

[24] Lenstra (J.), Veldhorst (M.) et Veltman (B.). – The complexity of scheduling trees with communication delays. *Journal of Algorithms*, vol. 20, n° 1, 1996, pp. 157–173.

[25] Lenstra (J. K.) et Shmoys (D. B.). – Computing near-optimal schedules. *In : Scheduling, Theory and its Applications*, éd. par Chrétienne (P.) et al., pp. 1–14. – J. Wiley, 1995.

[26] Munier (A.) et König (J.-C.). – A heuristic for a scheduling problem with communication delays, A paraître dans Operations Research.

[27] Picouleau (C.). – *Etude des Problèmes d'Optimisation dans les Systèmes Distribués.* – Thèse de PhD, Université Paris VI, France, 1992.

[28] Ranade (A. G.). – How to emulate shared memory. *In : Proceedings 28th Annual Symposium on Foundations of Computer Science.* pp. 185–192. – IEEE.

[29] Shmoys (D. B.), Wein (J.) et Williamson (P.). – Scheduling parallel machines on-line. *SIAM Journal on Computing*, vol. 24, n° 6, 1995, pp. 1313–1331.

[30] Valiant (L. G.). – A Bridging Model For Parallel Computation. *Communications of the ACM*, vol. 33, n° 8, 1990, pp. 103–111.

[31] Valiant (L. G.). – General purpose parallel architectures. *In : Algorithms and Complexity*, éd. par van Leuwen (J.), pp. 944–971. – Elsevier, 1990.

[32] Varvarigou (T.), Roychowdhury (V.) et Kailath (T.). – Scheduling in and out forests in the presence of communication delays. *In : Proceedings International Parallel Processing Symposium*, pp. 222–229. – Newport Beach, CA, 1993.

[33] Veltman (B.). – *Multiprocessor Scheduling with Communication Delays.* – Thèse de PhD, CWI-Amsterdam, Holland, 1993.

[34] Veltman (B.), Lageweg (B.) et Lenstra (J.). – Multiprocessor scheduling with communication delays. *Parallel Computing*, vol. 16, 1990, pp. 173–182.