Application irrégulière et ordonnancement en ligne

J. Briat
T. Gautier
J.L. Roch

LMC-IMAG Institut Fourier, BP 53X
100, rue des Mathématiques
38041 Grenoble Cedex 9
Tél. : +33 (0)4 76 51 46 53
Fax : +33 (0)4 76 51 48 46
E-mails : [Jacques.Briat, Thierry.Gautier, Jean-Louis.Roch]@imag.fr

RÉSUMÉ. *Beaucoup d'applications génèrent en cours d'exécution du parallélisme de manière non prévisible. Il est alors nécessaire d'utiliser des techniques dynamiques d'ordonnancement pour obtenir des exécutions efficaces. Nous considérons, dans cet article, le problème de l'ordonnancement d'une application qui génère dynamiquement des tâches de durées inconnues ayant des relations de précédence. En particulier, nous étudions les surcoûts dus à l'ordonnancement d'un tel programme dans le modèle PRAM. Deux critères seront analysés : la* régularité *de l'application qui mesure le travail minimal pour l'ordonnancer et le* facteur de compétitivité *de l'algorithme d'ordonnancement qui mesure la qualité de l'ordonnancement calculé.*

ABSTRACT. *For many applications, parallelism is generated during execution. Dynamic scheduling is then critical to guarantee an efficient execution. We consider the problem of scheduling an application that dynamically generates tasks of unknown duration with precedence relations. We focus on the overhead related to scheduling such a program on a PRAM, considering two aspects: the* regularity *of the application which measures the minimal work to schedule it and the* competitive ratio *of the scheduling algorithm which measures the quality of the performed schedule.*

MOTS-CLÉS : *algorithme irrégulier, ordonnancement en-ligne, régulation dynamique de charge, parallélisme.*

KEYWORDS: *irregular algorithm, on-line scheduling, dynamique load-balancing, parallel computing.*

1. Introduction

Une application parallèle dynamique est caractérisée par un comportement fortement dépendant des données en entrée. De manière générale, ni le graphe de précédence des tâches séquentielles effectuées lors de l'exécution, ni la durée de ces tâches ne peut être déterminé avant la terminaison de l'exécution.

Une telle application nécessite donc un mécanisme de régulation dynamique de charge qui calcule un ordonnancement des tâches. Afin que l'exécution de l'application soit efficace, l'algorithme d'ordonnancement doit satisfaire les critères suivants :

– le surcoût lié au calcul (dynamique) de l'ordonnancement doit être négligeable devant le temps global de l'exécution parallèle. Ce surcoût est mesuré par la *régularité* de l'application [18].

– l'ordonnancement réalisé doit être aussi près que possible de l'ordonnancement optimal. Le surcoût de l'algorithme d'ordonnancement peut être mesuré [27] par sa *compétitivité* (notion introduite de manière plus générale dans le contexte des algorithmes en-ligne [28]).

Dans cet article, nous analysons ces deux critères dans le cadre du modèle PRAM [16]. Malgré le fait que le calcul d'un ordonnancement optimal d'un graphe de précédence de tâches de durées connues soit un problème $\mathcal{NP}$-complet, il est possible de calculer, par un algorithme en-ligne, un ordonnancement qui est à un facteur constant de l'optimal même lorsque les durées sont inconnues.

Les algorithmes dynamiques d'ordonnancement peuvent être classés suivant différents critères [7]. La difficulté est alors d'évaluer leurs performances. Deux approches complémentaires doivent être considérées. D'une part, des expérimentations peuvent être utilisées dans le but de justifier un algorithme d'ordonnancement pour une certaine classe d'applications sur une machine donnée [31]. D'autre part, une étue de complexité par rapport à un modèle théorique de machine, que ce soit dans un cadre statique [5, 12] ou dynamique [27, 24, 14], permet d'évaluer théoriquement une classe de stratégies de régulation. Cet article se situe dans ce dernier contexte.

Organisation de l'article. Dans la seconde partie, nous rappelons les définitions d'*efficacité* et d'*extensibilité* (*scalability*) d'un algorithme [22]. Mais ces définitions ne prennent pas en compte les surcoûts dus au routage des données (les communications) et à l'ordonnancement des tâches. Nous étudierons donc les relations entre ces deux problèmes : de leurs analogies nous proposons la définition du *problème de l'exécution parallèle* [18] comme une extension du problème du routage [30] et du problème de l'ordonnancement.

Le travail total nécessaire pour ordonnancer un algorithme est mesuré par sa *complexité d'ordonnancement* qui peut être majorée par le nombre de tâches engendrées au cours de l'exécution. De cette définition nous proposons alors celle de la *régularité* d'un algorithme comme étant le rapport entre le nombre total d'opérations effectuées et sa complexité d'ordonnancement. Nous illustrons cette notion sur deux algorithmes parallèles dérivant de l'algorithme de tri-rapide (*i.e.* quick-sort).

Dans la quatrième partie, nous rappelons les définitions des grandeurs pour l'analyse des performances des algorithmes en-ligne. Dans le paragraphe 4.5, nous présentons des résultats concernant les algorithmes d'ordonnancement en-ligne [27]. Un résultat important est qu'un algorithme d'ordonnancement du type « glouton » possède un facteur de compétitivité optimal par rapport à une large classe d'algorithmes d'ordonnancement.

Afin d'obtenir en pratique de meilleures bornes sur les complexités, il est nécessaire de considérer une classe plus restreinte d'algorithmes. Le paragraphe 5.4 présente un algorithme d'ordonnancement d'un ensemble de tâches indépendantes de durées inconnues ayant un facteur $(1+\epsilon)$ de compétitivité pour tout algorithme PRAM efficace, extensible et de régularité polynomiale. Ce résultat s'applique aux algorithmes parallèles de tri-rapide présentés auparavant.

Nous concluons cet article par la présentation générale du modèle de programmation parallèle de la bibliothèque ATHAPASCAN-1 développé au sein du projet APACHE. ATHAPASCAN-1 est une bibliothèque C++ avec laquelle le parallélisme d'une application est exprimé dynamiquement par la construction d'un graphe de tâches avec des relations de précédence. L'algorithme de tri-rapide étudié dans la troisième partie a été programmé dans cet environnement : nous présenterons ses performances en utilisant deux techniques d'ordonnancement dynamique.

2. Des algorithmes parallèles à l'écriture de programmes portables

L'écriture d'un programme parallèle portable qui calcule la solution d'un problème donné passe par la construction d'un algorithme parallèle dans un modèle de machine abstraite. Plusieurs modèles ont été proposés dans la littérature [13, 2, 30]. Parmi eux, le modèle PRAM [13] (ainsi que ses variantes) est le plus utilisé pour la construction d'algorithmes parallèles.

L'algorithmique parallèle [10, 21, 20] vise la construction, dans des modèles théoriques tel que le modèle PRAM, d'algorithmes à la fois très parallèles (*i.e. extensible* ou avec un *facteur d'accélération polynomial*) et *efficace* (*i.e.* d'*inefficacité constante*) [22]. De tels algorithmes sont succeptibles de conduire à une implantation performante sur une architecture parallèle.

Plus précisément, soit $\mathcal{P}_n$ un problème et $T_s(n)$ le temps du meilleur algorithme séquentiel calculant la solution pour n'importe quelle instance de taille n de $\mathcal{P}_n$. Soit A un algorithme parallèle qui résout $\mathcal{P}_n$ en temps parallèle $T_p(n)$ sur $P(n)$ processeurs.

L'algorithme A est dit de *facteur d'accélération polynomial* [21] (ou *extensible*) s'il existe $\epsilon < 1$ tel que :

$$T_s(n^\epsilon) = \mathrm{O}(T_p(n))$$

Soit $H(n)$ le nombre d'opérations effectuées par A (sans compter les opérations vides, *i.e.* les *nop*). $H(n)$ est borné par $W(n) = T_p(n)P(n)$, le *travail* de l'algorithme A.

La notion d'inefficacité permet de caractériser le surcoût du travail d'un algorithme parallèle par rapport à son équivalent séquentiel. A est dit d'inefficacité :

– *constante* (A est dit *efficace*) si et seulement si :

$$H(n) = \mathrm{O}(T_s(n)) .$$

– *poly-logarithmique* si et seulement si :

$$H(n) = \mathrm{O}\Big(T_s(n) \log^{\mathrm{O}(1)}(T_s(n))\Big)$$

– *polynomiale* si et seulement si :

$$H(n) = \mathrm{O}\Big(T_s(n)^{\mathrm{O}(1)}\Big)$$

Exemple. La classe de complexité $\mathcal{EP}$ est l'ensemble des problèmes pour lesquels il existe un algorithme parallèle d'inefficacité constante et de facteur d'accélération polynomial. Tout problème de la classe $\mathcal{EP}$ admet un algorithme parallèle qui est efficace donc intéressant en pratique. Considérons par exemple l'élimination de Gauss. Bien que le problème soit dans la classe $\mathcal{NC}$, une version parallèle de l'algorithme séquentiel qui effectue des éliminations sur les colonnes permet de construire un algorithme de complexité parallèle n^2 en temps avec n processeurs. Cet algorithme, bien que non poly-logarithmique en temps (*i.e.* non dans $\mathcal{NC}$), est dans la classe $\mathcal{EP}$ et d'un grand intérêt [12] en pratique.

Le problème principal provient de l'écriture d'un programme parallèle associé à un algorithme *optimal* d'un point de vue théorique. Pour une architecture parallèle donnée ce problème revient à majorer les surcoûts des communications et de l'ordonnancement : le surcoût des communications est dû à la difficulté à simuler efficacement le modèle PRAM sur une architecture réelle ; le surcoût de l'ordonnancement est, comme nous allons le voir au paragraphe 2.2, dû à la difficulté de l'application pratique du principe de Brent.

2.1. Communication : définition de la localité

Il est possible de caractériser le surcoût des communications intrinsèques d'un algorithme en définissant la notion de *localité* (*locality* ou *gross locality*), *i.e.* le rapport entre la complexité des calculs et la complexité des communications [25]. La programmation des algorithmes non locaux nécessite des réseaux de communication à haute performance. Aussi longtemps que le surcoût des communications sera significatif sur les architectures parallèles actuelles, en d'autres termes aussi longtemps que le modèle PRAM ne sera pas simulé efficacement, l'obtention d'algorithmes performants passera par la maîtrise de la localité.

La notion de localité peut aussi être utilisée comme notion d'*irrégularité*. En effet, les algorithmes parallèles qui ont des schémas *réguliers* d'accès à la mémoire sont plus facilement programmés de manière à minimiser le coût des communications. De ce point de vue, plus un algorithme est local, plus il est régulier. Par conséquence, un critère important pour définir l'irrégularité d'un algorithme est l'irrégularité des schémas de communication qu'il entraîne [11]. Cette notion de localité est reliée au fait que les modèles de calcul parallèle permettent de distinguer les tâches de calcul et celles de communication [29].

2.2. Ordonnancement : définition de l'irrégularité

La gestion des tâches est un problème important pour obtenir des exécutions efficaces de programmes parallèles. Par exemple, le calcul de solutions optimales pour de problèmes de référence, tel que la numérotation des éléments d'une liste (*list ranking*), peut reposer entièrement sur le calcul d'un ordonnancement de tâches [9].

Plus précisément, le principe de Brent [6] affirme que tout programme parallèle synchrone effectuant x opérations en temps t peut être ordonnancé sur p processeurs pour être exécuté en temps $\lceil x/p \rceil + t$. Néanmoins, le principe de Brent ne prend pas en compte le surcoût du calcul d'un tel ordonnancement. Sa démonstration est basée sur l'indexation des opérations exécutées (de manière synchrone par le programme parallèle) par rapport à leur profondeur dans le graphe de précédence associé. Une allocation cyclique (modulo p) des opérations permet alors, sans surcoût, d'obtenir l'ordonnancement désiré des tâches du programme.

Cependant, dans le cas où le graphe de précédence associé à une instance des entrées ne peut être déterminé que par l'exécution du programme, l'indexation des opérations ne peut plus être calculée sans un surcoût important.

Dans ce qui suit, nous considérons les deux critères suivants pour évaluer le surcoût d'un ordonnancement :

– la *qualité* de l'ordonnancement : sur une machine parallèle donnée, cette qualité est évaluée par la comparaison par rapport au meilleur ordonnancement sur cette même machine, ou bien par le facteur d'accélération par rapport au meilleur algorithme séquentiel.

– le *coût* du calcul de l'ordonnancement. Ce coût est lié au nombre de tâches engendrées dans l'exécution du programme parallèle et peut être formalisé dans le définition de l'*irrégularité* [18].

La prochaine partie, basée sur [18], concerne la définition de la *complexité d'ordonnancement* d'un programme donné, qui permet de donner une borne inférieure au coût de calcul d'un ordonnancement indépendamment de sa qualité.

3. Ordonnancement et irrégularité

Rappelons les définitions introduites dans [18]. Le modèle de machine parallèle que nous considérons est composé d'un ensemble $\mathcal{P}$ de p processeurs. Un processeur possède une mémoire locale, peut communiquer avec d'autres processeurs à travers une mémoire globale et exécute séquentiellement un flot d'instruction. Dans ce modèle, nous considérons qu'un programme parallèle est composé d'un ensemble T de tâches prenant en entrée une donnée x d'un ensemble X. Soit $\mathcal{O}$ le sous-ensemble de $T \times X$ des couples (t, x) tel que t s'exécute avec x comme entrée.

3.1. Problème de l'exécution parallèle

Généralement, le surcoût dû aux communications est formalisé par le *problème de routage de données* (on parle aussi *d'ordonnancement des accès en mémoire*). Le surcoût dû à la gestion des tâches est formalisé quant à lui par le *problème d'ordonnancement des tâches*. Nous faisons abstraction de ces deux problèmes en considérant le *problème de l'exécution parallèle*qui donne une solution à la fois au problème du routage et à celui de l'ordonnancement. La qualité de la solution du problème du routage détermine le temps nécessaire pour simuler une machine PRAM par une autre machine modélisant les communications, comme, par exemple, dans les modèles DCM [22], LPRAM [1], ou XRAM [30].

En suivant le formalisme proposé dans [30], nous définissons le *problème de l'exécution parallèle*, noté *PEP*, de la manière suivante. Un *PEP* est un couple $(\mathcal{P}, \mathcal{S})$, où $\mathcal{S}$ est un *séquenceur* (*i.e. scheduler*) qui manipule des objets dans $\mathcal{O}$. Un *PEP* initialisé est un quadruplet $(\mathcal{P}, \mathcal{S}, \mathcal{I}, \mathcal{D})$, où $\mathcal{I}$ est la spécification des entrées, *i.e.* une application de $\mathcal{O} \subset T \times X$ dans $\mathcal{P} \times \mathcal{P}$ qui indique où sont initialement placées les données ainsi que les tâches. De la même manière, $\mathcal{D}$ est la spécification en sortie, *i.e.* une application de $\mathcal{O} \subset T \times X$ dans $\mathcal{P}$ qui détermine sur quels processeurs sont exécutées les tâches de T et où sont acheminées les données de X.

De plus, nous supposons que les tâches sont indivisibles et que le processeur qui débute l'exécution d'une tâche l'exécute jusqu'à sa terminaison. Cela signifie que nous n'autorisons pas dans notre modèle la migration de tâche (voir à ce sujet l'article [4] pour une discussion détaillée). Ce choix n'est pas restrictif puisqu'une tâche peut toujours être décomposée en une succession de tâches élémentaires, chacune étant engendrée à la fin de la précédente.

Les problèmes de routage et d'ordonnancement sont souvent considérés séparément mais présentent, au moins d'un point de vue théorique, des propriétés semblables et sont résolus par des techniques similaires.

Le problème de routage est un sous-problème du *PEP* lorsque les objets manipulés sont uniquement des données, *i.e.* $\mathcal{O} = X$. Le séquenceur $\mathcal{S}$ gère les transferts de données (ou de manière équivalente les accès en mémoire). Les données sont alors découpées en paquets et communiquées selon une certaine politique [30]. Le surcoût des communications peut être assimilé au coût des communications effectuées (sur une machine réelle les réseaux ont une certaine bande passante, ou bien les accès à une mémoire commune peuvent être quantifiés). Habituellement, aucun surcoût n'est associé au calcul des spécifications $\mathcal{I}$ et $\mathcal{D}$. En effet, quand le problème de routage est considéré, ces spécifications sont connues. Le problème de routage peut être résolu avant l'exécution (*off-line*) ou pendant l'exécution (*on-line*). L'exécution d'un programme est généralement une succession de phases de calcul et de synchronisation [29], ou bien ces phases peuvent être exécutées de manière asynchrone [11]. Chaque phase de synchronisation peut être exprimée par des schémas structurés de communication (diffusion, réduction, permutation...) ou bien non structurés.

De même, le problème d'ordonnancement est un sous-problème du *PEP* pour lequel $\mathcal{O} = T$, l'ensemble des tâches. Le séquenceur $\mathcal{S}$ est chargé de l'exécution des tâches générées par le programme. Il est composé d'estimateurs qui mesurent la charge de la machine et d'une politique qui décide de la date d'exécution et de l'affectation des tâches [32, 7]. Habituellement, et à la différence du problème du routage, le surcoût de l'ordonnancement comprend non seulement le coût de gestion des tâches (création, affectation) mais aussi celui de l'estimation de la charge [15] (à la création des tâches les spécifications des entrées et des sorties sont déjà calculées). Cette dualité permet de définir l'irrégularité d'un programme comme le coût de l'ordonnancement : plus un séquenceur travaille, plus l'algorithme est irrégulier. Le problème de l'ordonnancement peut lui aussi être résolu avant exécution (ordonnancement statique) ou pendant l'exécution (régulation dynamique de charge ou partage de charge). L'exécution d'un programme se décompose en une succession de phases de calcul et d'ordonnancement [9] ou bien les phases peuvent être exécutées de manière asynchrone. Comme pour le problème de routage, les schémas associés peuvent être réguliers (par exemple des tâches de durées unitaires sont à un instant donné à répartir sur les processeurs), ou irréguliers si les tâches créées dynamiquement sont de durées inconnues et diverses.

3.2. Surcoût dû à l'ordonnancement et irrégularité

Dans cette partie, nous nous intéressons essentiellement aux surcoûts pendant l'exécution d'un programme. Les surcoûts dus aux communications et à l'ordonnancement sont à considérer ensemble : le premier correspond au transfert des données, le second au calcul des spécifications (créations, affectations des tâches). Dans l'article [25], la localité d'un problème est le rapport entre le travail parallèle du meilleur programme PRAM qui le résout et la complexité des communications sur deux processeurs. Pour peu que nous soyons en mesure d'exprimer la *complexité d'ordonnancement* d'un programme, il sera alors facile d'exprimer, de la même manière, son *irrégularité* en pire cas.

Le modèle de machine parallèle que nous utilisons est celui d'une p-PRAM (CREW PRAM avec p processeurs). Dans ce modèle, le parallélisme est classiquement exprimé par l'instruction :

$$\texttt{for all}\, x \in X \,\texttt{in parallel do}\, \mathit{instruction}(x)$$

qui affecte à chaque élément x de X le processeur d'indice $code(x)$ qui est uniquement déterminé, en temps constant, à partir de x.

Au lieu de généraliser l'instruction $fork$ [16, 2], nous considérons qu'un programme engendre du parallélisme grâce à l'instruction suivante :

$$\texttt{for all}\, t \in G \,\texttt{in parallel do}\, \mathit{schedule}(t)$$

où G est un graphe de précédence qui spécifie les tâches à exécuter, ces tâches étant soumises à des relations de précédence. L'exécution de cette instruction consiste à repartir les tâches de G sur p processeurs de manière à ce que le temps d'exécution soit *optimal*. L'instruction $schedule$ est donc un oracle calculant une solution au *problème de l'ordonnancement d'un graphe*, noté *POG* par la suite. Ce problème peut être défini ainsi :

entrée : p un nombre de processeurs et G un graphe orienté sans cycle (*DAG*) ayant $\#G$ nœuds. Chaque nœud de G est une tâche. Une tâche est un programme pouvant s'exécuter sur un processeur et pouvant contenir d'autres instructions du type *schedule*. La longueur d'une tâche est son temps séquentiel, *i.e.* le nombre d'unités de temps nécessaires à l'exécution de la tâche, ainsi que des tâches qu'elle génère, lorsqu'un ordonnancement séquentiel est utilisé (exécution sur un processeur). Le graphe peut ne pas être connu avant l'exécution complète du programme, mais il est indépendant de la manière dont une tâche est ordonnancée, en particulier lorsqu'elle contient des instructions du type *schedule*.

sortie : $L_p^*(G)$ le temps d'exécution optimal de G sur p processeurs.

Insistons sur le fait qu'une tâche peut contenir des instructions de branchement et du type *schedule*, c'est à dire qu'une solution au *POG* permet l'ordonnancement d'un programme parallèle pour lequel le graphe de précédence est inconnu.

Considérons comme sous-problème celui pour lequel chacune des tâches de G est séquentielle (*i.e.* ne contenant pas d'instructions du type *schedule*) et de durées connues. Dans ce cas, résoudre une instance du *POG* est un problème $\mathcal{NP}$-complet et décider si $L_p^*(G)$ est égal à un nombre entier donné est *co*-$\mathcal{NP}$-complet [17].

Complexité d'ordonnancement. Soit A un programme parallèle (contenant des instructions du type *schedule*) qui calcule la solution d'un certain problème $\mathcal{P}$. Pour toute entrée x de taille n, l'exécution de A engendre un DAG G_x qui contient au plus $\#G_x$ tâches séquentielles.

Soit $H(G_x)$ le nombre d'opérations exécutées et $L^*(G_x)$ la longueur d'un chemin critique de G_x. Alors, $H(G_x)$ est la somme des longueurs des tâches et $L^*(G_x)$ est égal au temps parallèle minimal pour l'exécution de G_x sur un nombre infini de processeurs.

Considérons maintenant l'exécution de A avec en entrée x sur une p-PRAM. Nous avons alors l'encadrement suivant pour $L_p^*(G_x)$:

$$\lceil H(G_x)/p \rceil \leq L_p^*(G_x) \leq \lceil H(G_x)/p \rceil + L^*(G_x) \qquad [1]$$

Une borne inférieure pour le calcul séquentiel du *POG* est $\#G_x$. De plus, puisque qu'aucune information n'est connue quant à l'ordonnancement optimal calculé par l'oracle, toutes les instructions *schedule* peuvent être exécutées sur le même processeur lorsque $p = \mathrm{O}(1)$. Nous supposerons donc que le coût du calcul de l'ordonnancement par l'oracle est $\#G_x$.

Définition 1 . *La* complexité d'ordonnancement *de A, noté $\sigma(n)$, est le nombre maximum de tâches pour n'importe quelle entrée x de taille n :*

$$\sigma(n) = \max\{\#G_x / \mid x \mid = n\} .$$

Soit $T_p^*(x)$ le temps parallèle optimal de l'exécution sur p processeurs de A avec x en entrée, en considérant à la fois le temps d'exécution $L_p^*(x)$ de l'ordonnancement optimal et la complexité d'ordonnancement qui est une borne inférieure au temps nécessaire pour calculer un tel ordonnancement quelque soit la longueur des tâches.

Notons $H(n), L^*(n), L_p^*(n)$ et $T_p^*(n)$ les valeurs en pire cas associées à toute entrée x de taille n. Une borne inférieure au temps d'exécution parallèle, en incluant à la fois le temps d'exécution de l'ordonnancement optimal et son calcul par l'oracle du séquenceur, est :

$$T_p^*(n) \geq \text{Max}\left(\lceil H(n)/p \rceil, L^*(n)\right) + \sigma(n) \qquad [2]$$

Soit $T_s(n)$ le temps du meilleur programme séquentiel calculant une solution de $\mathcal{P}$. En pratique, les programmes parallèles qui sont à la fois extensibles et efficaces sont dans $\mathcal{EP}$ et vérifient :

$$\begin{cases} pT_p^*(n) = \mathrm{O}(T_s(n))\,, \\ \exists \epsilon < 1 : L^*(n) = T_s(n^\epsilon) \end{cases} \qquad [3]$$

Considérons un tel programme dans $\mathcal{EP}$, nous avons $H(n) = \mathrm{O}(T_s(n))$, et [2] se réduit à :

$$p\sigma(n) = \mathrm{O}(H(n)) \qquad [4]$$

Un programme A ayant un facteur d'accélération polynomial implique que A peut théoriquement être exécuté efficacement avec un nombre polynomial de processeurs, donc que $H(n)/\sigma(n) = n^{\Omega(1)}$.

Définition 2 . *La* régularité en pire cas $\rho(n)$ *d'un programme utilisant des instructions du type $schedule$ est le rapport entre le nombre d'opérations $H(n)$ nécessaires à son exécution et sa complexité d'ordonnancement :*

$$\rho(n) = \frac{H(n)}{\sigma(n)} \qquad [5]$$

De même que pour la localité [25], les programmes dans notre modèle de calcul peuvent être classés relativement à leur régularité :

Définition 3 . *Un programme est dit :*

– irrégulier *si et seulement si* $\rho(n) = O(1)$*,*

– log-régulier *ssi* $\rho(n) = \log^{O(1)} H(n)$*,*

– poly-régulier *(ou* régulier*) ssi* $\rho(n) = H(n)^{O(1)}$.

De telles définitions s'appliquent d'avantage aux programmes « en pratique » qu'aux problèmes. Un programme efficace et dynamique, qui peut facilement être écrit en utilisant de l'ordonnancement dynamique, est irrégulier si l'utilisation du séquenceur contribue de manière significative à son efficacité. Réciproquement, tout programme qui inclue son propre séquenceur est régulier puisque seulement une instruction $schedule$ est exécutée initialement pour démarrer le programme sur chacun des p processeurs.

Par exemple, considérons le calcul d'une FFT sur n points facilement programmé avec une irrégularité $\mathrm{O}(1)$ en intégrant le séquenceur dans le programme. Si nous supposons que les coûts des opérations élémentaires sont inconnus, ce programme avec placement statique sera inefficace à cause du nombre important de synchronisations inhérentes à l'algorithme. Considérons maintenant le programme qui construit le graphe de précédence G_n du calcul de la FFT (qui ne dépend que de n) et qui exécute l'instruction $schedule(G_n)$. Puisque $\sigma(n) = n \log n$, ce programme est irrégulier mais son exécution sera efficace.

3.3. Exemple du tri-rapide parallèle

Considérons une variante de l'algorithme de tri parallèle proposé par Reischuk [26]. Cet algorithme peut être vu comme une généralisation du tri-rapide séquentiel [20].

Soit T un tableau contenant n éléments à trier. L'algorithme commence par choisir $k-1$ éléments pivots ($k = \sqrt{n}$) et les trie. Dans l'algorithme original, cette étape est réalisée en parallèle sur n processeurs, mais par souci de simplicité nous utiliserons un algorithme de tri séquentiel.

Soit $p_0 = -\infty, p_1, \ldots, p_{k-1}, p_k = +\infty$ les pivots triés : deux pivots consécutifs (p_i, p_{i+1}) définissent un sac B_i, $0 \leq i < k$. Chaque élément x de T est associé à un sac B_i tel que $p_i \leq x < p_{i+1}$. La détermination du sac associé à x est calculée en parallèle en utilisant un processeur par élément.

Le tri de T peut alors être calculé en parallèle en utilisant pour chacun des sacs un algorithme de tri qui est soit séquentiel (noté $\mathcal{A}^{(1)}$), soit récursif (noté $\mathcal{A}^{(2)}$).

Pour chacun des deux algorithmes, l'irrégularité dépend du nombre d'opérations exécutées dans l'algorithme de tri séquentiel. Si l'algorithme est celui du tri-rapide (*quick-sort*), alors pour chaque algorithme, les nombres $H^{(1)}(n)$ et $H^{(2)}(n)$ d'opérations exécutées sont encadrés par :

$$n \leq H^{(i)}(n) \leq n^2 \quad i = 1, 2.$$

La complexité d'ordonnancement de l'algorithme $\mathcal{A}^{(1)}$ est bornée par $k+1$ puisque seule une instruction $schedule$ est exécutée avec $k+1$ tâches. Donc la régularité en pire cas $\rho^{(1)}(n)$ de $\mathcal{A}^{(1)}$, qui dépend du tableau en entrée, est encadrée par :

$$n^{1/2} \leq \rho^{(1)}(n) \leq n^{3/2}$$

donc $\mathcal{A}^{(1)}$ est régulier.

Pour l'algorithme $\mathcal{A}^{(2)}$, la découpe récursive est exécutée jusqu'à ce qu'il ne reste plus qu'un élément par sac. Donc sa complexité d'ordonnancement est $H^{(2)}(n)$. L'algorithme est de régularité :

$$\rho^{(2)}(n) = 1$$

et $\mathcal{A}^{(2)}$ est irrégulier.

3.4. Autres opérations autorisées

Le problème de l'exécution parallèle (PEP) précise les entrées et les sorties pour un algorithme d'ordonnancement (*i.e.* un *séquenceur*), mais ne spécifie pas quelles opérations un séquenceur peut exécuter, mis à part l'exécution d'une tâche (un nœud élémentaire du graphe associé au PEP) sur un processeur. Le placement, par le séquenceur, d'un ensemble de tâches sur un processeur est supposé être de temps unitaire.

D'autres opérations du séquenceur peuvent être [5] :

Préemptivité : le séquenceur est dit *préemptif* s'il peut arrêter une tâche en cours d'exécution sur un processeur. De la même manière, un séquenceur est dit *non-préemptif* s'il n'a aucun contrôle sur une tâche qui s'exécute sur un processeur, excepté la possibilité de récupérer l'information de sa terminaison.

Migration : lorsque la migration est autorisée, un séquenceur préemptif peut suspendre une tâche pendant son exécution et la transférer sur un autre processeur. La tâche continue alors son exécution sur le nouveau processeur.

L'hypothèse de non-préemptivité est très forte, et il peut ne pas être réaliste de supposer qu'une fois une tâche débute, celle-ci se termine sans aucun recours jusqu'à sa terminaison qui est inconnue [27]. Un séquenceur *non-préemptif avec redémarrage* permet la suppression d'une tâche et son redémarrage sur un autre processeur [27]. Cette opération est très repandue en pratique (notamment couplée à la mise en place par l'utilisateur de points de reprise). Par exemple, le système de batch *Easy*, disponible sur la machine IBM-SP2 [23], peut tuer un processus si celui-ci dépasse son temps imparti lors de sa soumission; il est ensuite possible de le relancer, depuis son début ou son dernier point de reprise, sur un autre processeur.

4. Algorithme en-ligne et facteur de compétitivité

La précédente définition de l'irrégularité est focalisée sur le travail nécessaire à un séquenceur pour exécuter efficacement un programme. Pour une implémentation pratique, l'algorithme d'ordonnancement utilisé doit surtout être capable, dans un modèle p-PRAM, de calculer un ordonnancement *performant* du graphe de précédence généré pendant l'exécution. Quand ce graphe n'est pas entièrement connu (spécialement quand la durée des tâches n'est pas connue), l'ordonnancement doit être calculé dynamiquement en tenant compte à la fois de l'activité de la machine et de la connaissance apportée par l'application. Un tel algorithme d'ordonnancement est alors appelé *algorithme en-ligne*.

Dans le cas général, du fait de la difficulté intrinsèque du calcul d'un ordonnancement (voir la section 3.2 précédente) et du fait de la volonté de conserver un facteur d'accélération linéaire, nous nous intéressons au calcul d'un ordonnancement proche de l'optimal, c'est à dire à un facteur constant près de l'optimal. Dans le cadre des algorithmes en-ligne, cela correspond à la notion de *facteur de compétitivité*.

Dans cette section, la présentation de la théorie des algorithmes en-ligne, développé par [28], est inspirée de [3]. Dans cette théorie, l'analyse des performances d'un algorithme fait appel à la notion de *facteur de compétitivité*. Après un bref rappel des définitions, nous présentons leurs applications au cas des algorithmes d'ordonnancement en-ligne.

4.1. Algorithmes en-ligne déterministes et probabilistes

La théorie des algorithmes en-ligne a été développée dans le cadre des jeux du type question-réponse. Dans un tel jeu, un algorithme *en-ligne* doit répondre à une série de questions tout en essayant de minimiser une fonction de coût. L'algorithme est dit *en-ligne* s'il est capable de répondre à une question sans la connaissance de l'ensemble des questions ni de sa longueur. Inversement, un algorithme est dit *hors-ligne* s'il construit la séquence des réponses grâce à la connaissance de celle des questions.

Définitions préliminaires. Un jeu de question-réponse est un ensemble Q de questions, un ensemble fini R de réponses et un ensemble de fonctions de coût $f = (f_n)_{n\in\mathbb{N}}$ définies pour tout n-uplet de $Q \times R$:

$$f_n : (Q \times R)^n \longrightarrow \mathbb{R} \cup \{\infty\}.$$

f est la fonction de coût [1] associée au jeu.

Un *algorithme en-ligne déterministe* A^d est une séquence de fonctions $a_n : Q^n \longrightarrow R$, pour $n \geq 1$. Pour toute entrée $\underline{q} = (q_1, \ldots, q_n) \in Q^n$, A^d calcule en sortie la séquence $\underline{r} = (r_1, \ldots, r_n) \in R^n$ avec $r_i = a_i(q_1, \ldots, q_i)$, $i = 1 \ldots n$. Le coût de A^d avec $\underline{q}$ en entrée est $c_{A^d}(\underline{q}) = f(\underline{q}, \underline{r}) = f_n(\underline{q}, \underline{r}) = f_n(\underline{q}, A^d(\underline{q}))$.

Un *algorithme en-ligne probabiliste* A^p est une distribution d'algorithmes en-ligne déterministes A^d_X (X étant la variable aléatoire associée à la distribution). Pour toute séquence des entrées $\underline{q} \in Q^n$, la séquence des sorties $A^p(\underline{q}) = (r_1, \ldots, r_n)$ et le coût $c_{A^p}(\underline{q})$ sont des variables aléatoires : à l'étape i, chaque r_i dépend du choix aléatoire de l'algorithme déterministe A^d_x.

L'algorithme hors-ligne optimal *Opt* est défini comme l'algorithme qui, pour toute séquence de questions $\underline{q}$, calcule une séquence de réponses $\underline{r}$ telle que $f(\underline{q}, \underline{r})$ soit minimale, *i.e.* :

$$\begin{array}{rccc} Opt\colon & \bigcup_{n=1}^{\infty} Q^n & \longrightarrow & \bigcup_{n=1}^{\infty} R^n \\ & \underline{q} = (q_1, \ldots, q_n) & \longmapsto & \underline{r} = (r_1, \ldots, r_n) \end{array}$$

avec $f(\underline{q}, \underline{r}) = \min\{f_n(\underline{q}, \underline{r'}) \mid \underline{r'} \in R^n\}$. Donc, la fonction de coût c^* de l'algorithme hors-ligne optimal vérifie :

$$\forall \underline{q} \in Q^n \quad c^*(\underline{q}) = \min\{f_n(\underline{q}, \underline{r'}) \mid \underline{r'} \in R^n\}.$$

4.2. Facteur de compétitivité

La performance d'un algorithme en-ligne est caractérisé par son *facteur de compétitivité* qui mesure le rapport entre le coût de la solution qu'il calcule par rapport au coût de la solution calculée par le meilleur algorithme d'un classe donnée.

Soit A^d un algorithme en-ligne déterministe et $\mathcal{C}$ une classe d'algorithmes. A^d est dit α*-compétitif* contre $\mathcal{C}$ s'il existe une constante α telle que, pour toute séquence de questions $\underline{q}$:

$$c_{A^d}(\underline{q}) \leq \alpha \min\{c_A(\underline{q}) \mid A \in \mathcal{C}\}.$$

1. i.e. $\forall n \in \mathbb{N}\ \forall \underline{q} = (q_1, \ldots, q_n) \in Q^n \quad \forall \underline{r} = (r_1, \ldots, r_n) \in R^n : \ f(\underline{q}, \underline{r}) = f_n(\underline{q}, \underline{r})$.

Cette définition est étendue de manière naturelle au cas d'un algorithme en-ligne probabiliste. A^p est α-compétitif contre $\mathcal{C}$ s'il existe une constante α telle que, pour toute séquence de questions $\underline{q}$:

$$E_X\left(c_{A^p}(\underline{q})\right) \leq \alpha \min\{c_A(\underline{q}) \mid A \in \mathcal{C}\}$$

où l'espérance E_X est prise pour tout les choix aléatoires dans A^p.

Par extension, un algorithme en-ligne est dit *α-compétitif* s'il est α-compétitif contre l'algorithme hors-ligne optimal.

Deux techniques principales sont utilisées pour calculer des bornes sur le facteur de compétitivité. Pour calculer une borne supérieure, il suffit d'exhiber un algorithme en-ligne qui résout le problème avec cette même borne. Pour calculer une borne inférieure, la technique principale consiste à concevoir un algorithme, appelé l'*adversaire*, qui construit sa séquence de questions $\underline{q}$ de la manière suivante : la question q_{n+1} est construite à partir de la connaissance de tous les couples précédents (q_i, r_i), $1 \leq i \leq n$, dans le but de maximiser la fonction de coût $f(r_1, \ldots r_n)$.

4.3. Application à l'ordonnancement en-ligne

Plaçons nous dans le cadre du problème de l'ordonnancement et soit G le graphe d'une instance du problème de l'exécution parallèle (PEP) qui doit être ordonnancé sur une p-PRAM. La fonction de coût $f(G)$ à minimiser est la longueur de l'ordonnancement, noté par la suite $L_p(G)$. Avec la connaissance complète du graphe de précédence, l'algorithme hors-ligne optimal calcule un ordonnancement de longueur minimale, noté $L_p^*(G)$.

Soit A^d un algorithme en-ligne déterministe qui résout le problème de l'exécution parallèle, et $L^d(G)$ la longueur de l'ordonnancement calculé par A^d sur l'instance G.

L'algorithme A^d est α-compétitif si et seulement si, pour toute instance G, $L^d(G) \leq \alpha L_p^*(G)$.

5. Algorithmes d'ordonnancement en-ligne

De nombreux algorithmes d'ordonnancement en-ligne ont été proposés. Dans [27], D. Shmoys, J. Wein and D. Williamson donnent différentes bornes inférieures et supérieures au problème de l'ordonnancement en considérant de nombreux modèles de machines parallèles, en particulier le *modèle de machines identiques* (qui correspond au modèle p-PRAM que nous considérons dans cet article), le *modèle de machines uniformément liées* (le rapport de vitesse de deux machines est constant) et le *modèle de machines non liées* (le rapport de vitesse de deux machines est constant pour une tâche donnée mais peut varier d'une tâche à l'autre).

Dans la première partie, nous considérons l'algorithme en-ligne proposé par Graham [19] et nous étudions son facteur de compétitivité. Ensuite, nous présentons des bornes inférieures [27] pour résoudre ce problème.

Dans la seconde partie, nous nous intéressons au problème de l'ordonnancement d'un ensemble de tâches indépendantes de durées inconnues. Nous rappelons l'algorithme déterministe fondamental de R. Cole and U. Vishkin [9]. Nous proposons ensuite une variante à gros grain de cet algorithme qui est asymptotiquement $(1+\epsilon)$-compétitif et dont le surcoût dû au calcul de l'ordonnancement est borné par le temps parallèle.

Dans le cas de la p-PRAM (machines identiques), les principaux résultats sont :

– il existe un algorithme d'ordonnancement en-ligne déterministe et non-préemptif ayant un facteur de compétitivité de $(2-\frac{1}{p})$ [19].

– une borne inférieure au facteur de compétitivité des algorithmes en-ligne déterministes préemptifs ou non (respectivement probabilistes non-préemptifs) est $(2-\frac{1}{p})$ (respectivement $(2-\frac{1}{\sqrt{p}})$) [27].

5.1. Algorithme glouton d'ordonnancement en-ligne

Les algorithmes d'ordonnancement à base de listes sont classiquement utilisés pour l'ordonnancement des tâches. Ces algorithmes consistent à gérer des listes des tâches pouvant être exécutées. Quand un processeur termine l'exécution d'une tâche, il rend « exécutable » ses successeurs relativement aux relations de précédence. S'il existe une tâche exécutable alors il la prend et commence son exécution. Dans le cas contraire, le processeur reste inactif jusqu'à ce qu'une tâche devienne exécutable.

Généralement, un algorithme à base de listes spécifie un ordre entre les tâches par l'affectation d'une priorité à chaque tâche (qui dépend de la structure du graphe ou de la longueur des tâches). Dans le cas des algorithmes en-ligne, puisque la durée des tâches est supposée inconnue, l'algorithme appelé « glouton » affecte les tâches aux processeurs sans tenir compte de leurs priorités. De plus, le graphe peut être déterminé dynamiquement à partir de l'exécution des instructions du type $schedule$.

Notons que nous n'avons pas considéré ici le coût du calcul de l'ordonnancement, en particulier celui de l'affectation de tâches différentes à des processeurs différents. Dans le cas de l'ordonnancement d'un graphe avec n tâches, une borne inférieure à ce coût est n, *cf* paragraphe 3.2.

5.2. Bornes pour l'ordonnancement non-préemptif

Proposition 1 . *L'algorithme glouton d'ordonnancement est* $\left(2-\frac{1}{p}\right)$*-compétitif dans le modèle p-PRAM,*

Démonstration. Soit G un graphe de précédence et $L^*(G)$ le temps d'exécution de l'ordonnancement optimal obtenu par l'algorithme hors-ligne sur une p-PRAM. La durée – inconnue – d'une tâche t de G est notée $l(t)$. Soit $L(G)$ la longueur de l'ordonnancement calculé par l'algorithme glouton.

Soit $I(G)$ le temps total d'inactivité dans l'exécution de l'ordonnancement glouton. Nous avons :

$$L(G) = \frac{I(G) + \sum_{t\in G} l(t)}{p}. \qquad [6]$$

Puisque $L^*(G) \geq \frac{\sum_{t \in G} l(t)}{p}$, [6] donne :

$$L^*(G) \geq L(G) - \frac{I(G)}{p}. \qquad [7]$$

Soit C un chemin arbitraire dans le DAG de G. On a $L^*(G) \geq \sum_{t \in C} l(t)$.

L'algorithme glouton est tel qu'a tout instant au moins un processeur exécute une tâche. De plus, si à un instant donné un processeur est inactif alors il existe une tâche sur un chemin critique qui est en cours d'exécution. Dans le diagramme de Gantt de l'exécution de l'ordonnancement glouton, considérons les instants θ_i, $1 \leq i \leq \Theta$, pour lesquels au moins un processeur est inactif. Soit t_i l'opération exécutée à l'instant θ_i par une tâche sur le chemin critique. Chaque opération élémentaire t_i se trouve sur un chemin critique, nous avons donc :

$$\Theta \leq L^*(G) \qquad [8]$$

qui donne la borne suivante sur le temps d'inactivité :

$$I(G) \leq (p-1)\Theta \leq (p-1)L^*(G) \qquad [9]$$

En reportant cette expression dans [7], nous obtenons :

$$L(G) \leq L^*(G)\left(1 + \frac{p-1}{p}\right) \qquad [10]$$

ce qui termine la démonstration. $\square$

L'algorithme glouton d'ordonnancement a un facteur de compétitivité plus petit que 2 et est une solution satisfaisante (avec une inefficacité asymptotique constante) au problème de l'exécution parallèle (PEP).

Corollaire 1 . *Soit A un algorithme parallèle d'inefficacité constante, de facteur d'accélération polynomial et de régularité polynomiale qui exécute $H(n)$ opérations. Alors, $\exists M \geq 1, \exists \delta > 0$ tel que $\forall n,\ \forall p \leq n^\delta$, A peut être exécuté en temps parallèle*

$$T_p(n) \leq M\frac{H(n)}{p}$$

sur une p-PRAM.

Démonstration. Il suffit d'ordonnancer l'algorithme parallèle en utilisant une implémentation de l'algorithme glouton d'ordonnancement qui soit centralisée. A chaque fois qu'un processeur est inactif, il demande à un processeur particulier (le *maître*) une nouvelle tâche à exécuter. Pour chaque instruction $schedule$, le processeur maître centralise les tâches à exécuter et affecte à chaque processeur au plus une tâche à chaque étape.

Puisque l'algorithme est d'inefficacité constante et de facteur d'accélération polynomial, il existe M et k tels que $\forall p \leq H(n)^k$:

$$L_p^*(n) = M\frac{H(n)}{p}.$$

Puisque la régularité est polynomiale, nous avons $\rho(n) = H(n)^{O(1)}$.

Considérons l'exécution de A en utilisant l'algorithme glouton d'ordonnancement sur $p \leq \mathrm{Min}(H(n)^k, \rho(n))$ processeurs. Puisque le nombre de tâches générées par l'algorithme est majoré par $\frac{H(n)}{\rho(n)}$, le surcoût dû au calcul parallèle de l'ordonnancement glouton est majoré par le même terme.

De plus, d'après la proposition 1, $L_p(n) \leq 2L_p^*(n)$. Nous obtenons finalement :

$$T_p(n) \leq 2M\frac{H(n)}{p} + O\left(\frac{H(n)}{\rho(n)}\right) \qquad [11]$$

ce qui entraîne $T_p(n) \leq O\left(\frac{H(n)}{p}\right)$. $\square$

L'ordonnancement glouton en-ligne est donc une manière efficace de réguler dynamiquement la charge d'un programme parallèle dont le graphe de précédence est inconnu.

5.3. Minorations du facteur de compétitivité de l'algorithme glouton

Une question naturelle est de déterminer s'il est possible d'avoir un facteur de compétitivité meilleur que $(2-\frac{1}{p})$ soit en utilisant le même modèle, soit en considérant une plus grande classe d'algorithmes d'ordonnancement.

Ce problème a été étudié dans [27] où l'on trouvera la démonstration complète de la proposition suivante :

Proposition 2 *[27]. Dans le cadre du modèle p-PRAM, le facteur de compétitivité est minorée par $\left(2 - \frac{1}{p}\right)$ – respectivement $\left(2 - \frac{1}{\sqrt{p}}\right)$ – pour tout algorithme d'ordonnancement déterministe avec ou sans préemption – respectivement probabiliste sans préemption –.*

Démonstration. Donnons seulement une idée de la preuve pour le cas d'un ordonnancement déterministe sans préemption. En utilisant la technique présentée au paragraphe 4.2., l'adversaire construit l'instance suivante de G proposée par Graham [19]. G contient $1 + p(p-1)$ tâches indépendantes. Une tâche α_1 est de longueur p, tandis que les autres tâches β_k, $1 \leq k \leq p(p-1)$ sont de longueur 1.

L'ordonnancement optimal est de longueur p. La tâche α_1 est exécutée sur un processeur donné, et les $p(p-1)$ autres tâches unitaires β_k sur les $p-1$ autres processeurs.

La longueur de tout ordonnancement de G est égale à $p+t$, où t est l'instant lorsque la tâche α_1 démarre son exécution. Puisque les durées des tâches sont inconnues pour l'algorithme d'ordonnancement, la stratégie de l'adversaire consiste à rendre t aussi important que possible.

Les tâches qui sont exécutées en premier sont les $p(p-1)$ tâches β_k, qui sont exécutées en $p-1$ unités de temps, sans inactivité. à l'instant $t = p-1$, la tâche α_1 commence son exécution. La longueur de l'ordonnancement obtenu est alors $2p-1$, ce qui démontre la borne inférieure précédente. $\square$

Lorsque le graphe de précédence des tâches n'est pas connu, il est alors impossible de construire un algorithme en-ligne avec un facteur de compétitivité meilleur que celui de l'algorithme glouton.

La construction d'un algorithme d'ordonnancement en-ligne plus efficace, nécessite la définition d'un problème plus restrictif. Dans ce cas, il est intéressant, de considérer des facteurs de compétitivité asymptotiques pour certaines structures de graphes particulières, introduisant une connaissance sur les durées des tâches.

5.4. Ordonnancement de tâches indépendantes

Dans cette partie, nous considérons le problème suivant défini par R. Cole et U. Vishkin dans [9] et dénommé par la suite le problème de « *l'ordonnancement de tâches indépendantes* » (*i.e. Independant Tasks Scheduling Problem*, noté en abrégé $ITSP$ par la suite). L'$ITSP$ est un sous-problème du *PEP* défini comme suit [9]. Soit n tâches chacune de durée (longueur) inconnue mais comprise entre 1 et $c(n)$. La longueur totale des tâches est bornée par $H(n)$ ($c(n)$ et $H(n)$ sont au plus des polynômes en n).
L'$ITSP$ consiste à ordonnancer l'exécution des tâches sur une p-PRAM de manière à ce que le temps d'exécution soit O$(\max\{H(n)/p, c(n)\})$.

La solution (complexe) de [9] n'utilise ni la préemptivité ni la migration de tâche. Cette solution démontre la proposition suivante :

Proposition 3 *[9]. Si $H(n) = O(n)$ and $c(n) = O(\log n)$, alors, pour toute valeur $p \leq \log n$, il est possible de résoudre l'ITSP sur une p-PRAM en temps :*

$$T_p(n) = O(H(n)/p)$$

en comptant le surcoût dû à l'ordonnancement.

Les facteurs constants de la complexité précédente sont importants. Aussi est-il intéressant, pour des raisons pratiques, de considérer des algorithmes d'ordonnancement à « gros grain ». Dans la suite, nous considérons le problème restreint de l'$ITSP$ suivant :

entrée : p un entier et n tâches indépendantes de longueur maximale $T_M(n)$.

sortie : l'exécution des n tâches sur une p-PRAM.

Dans les paragraphes suivants, nous considérons deux algorithmes qui résolvent l'$ITSP$. Le premier permet d'obtenir un algorithme $(1 + \epsilon(n))$–compétitif sur une p–PRAM (pour $p < \frac{n}{\log^2 n}$) avec $\lim_{n\to\infty} \epsilon(n) = 0$. Le second utilise une connaissance supplémentaire sur les durées des tâches et m permet d'obtenir un algorithme à base de listes qui est $4/3$-compétitif.

5.4.1. *Un algorithme d'ordonnancement* $(1 + \epsilon)$*-compétitif*

Une stratégie souvent rencontrée en pratique est de séparer les processeurs chargés du calcul de l'ordonnancement de ceux chargés de l'exécution des tâches de l'application. Nous analysons ici la compétitivité d'une telle stratégie.

Supposons que q processeurs parmi les p soient consacrés à l'exécution des tâches et les $p-q$ autres au calcul de l'ordonnancement. Pour garantir une exécution optimale pendant t tops de calcul, il faut que chacun des q processeurs ait au moins t tâches en réserve. La redistribution de tâches entre les q processeurs se ramène à un calcul de préfixe et prend un temps minimal $\log q$ sur $\frac{q}{\log q}$ processeurs; la stratégie minimale consiste donc à choisir $q = p(1 - \frac{p}{\log p})$. Le problème $ITSP$ peut être alors résolu par l'algorithme parallèle suivant :

1. Initialisation : affectation de $2 \log p$ tâches à chacun des $q = p(1 - \frac{p}{\log p})$ processeurs et stockage des $r = n - 2q\log p$ autres tâches dans le tableau $R[1..r]$.

2. Exécution : en parallèle, chaque processeur de calcul exécute $\log p$ pas de calcul avec les tâches qui lui ont été affectées. Soit F_k le nombre de tâches terminées par le processeur k pendant cette étape.

3. Redistribution : soit $\pi_k = \sum_{i=1}^{k} F_i$ (calcul d'un préfixe pour l'opération d'addition). Soit $i_0 = r - \pi_p$.

   Si i_0 est positif, assigner à tout processeur de calcul d'indice k les tâches du tableau R d'indices $i_0 + \pi_{k-1}, \ldots, i_0 + \pi_k$. Poser $r = r - \pi_p$ et retourner à l'étape 2.
   Sinon, si i_0 est négatif, distribuer toutes les tâches restantes dans R de manière à ce que le nombre de tâches de deux processeurs distincts diffère d'au plus un. Ensuite poser $r = 0$.

4. Terminaison : Chacun des processeurs de calcul exécute les tâches en sa possession jusqu'à complétion.

Proposition 4 . *L'algorithme précédent d'ordonnancement à gros grain exécute n tâches sur $p \leq \frac{n}{\log^2 n}$ processeurs avec un facteur de compétitivité majoré par*

$$\left(1 + \frac{1}{-1 + \log p} + \frac{2p \log p T_M(n)}{H(n)}\right).$$

Démonstration. Posons $\tau = (1 - \frac{1}{\log p})$. Puisque $p\tau \log p$ opérations sont exécutées à chaque étape « Exécution » tant que r est positif, le nombre d'itérations est majoré par $\frac{H(n)}{p\tau \log p}$. Le coût d'une redistribution est borné par $\log p$ (coût de calcul d'un préfixe sur les $\frac{p}{\log p}$ processeurs dédiés à l'ordonnancement). En définitif, le temps parallèle, en incluant à la fois le coût de l'ordonnancement des tâches et de leur exécution, est majoré par

$$T_p(n) \leq \frac{H(n)}{\tau p} + 2 \log p \, T_M(n) \qquad [12]$$

Une borne inférieure de la longueur de l'ordonnancement est donnée par l'algorithme hors-ligne optimal et est minorée par $\frac{H(n)}{p}$. Nous en déduisons la majoration voulue du facteur de compétitivité. □

Application à l'algorithme parallèle de tri-rapide. Reprenons l'algorithme parallèle de tri $\mathcal{A}^{(1)}$, vu à la section 3.3. Par souci de simplicité, nous considérons le cas moyen lorsque $T_M(n) = \Omega(n^{1/2} \log n)$ et $H(n) = \Omega(n \log n)$. De la proposition 4, nous avons :

$$T_p(n) \leq \Omega\left(\frac{n \log n}{\tau p} + 2p \log p\, n^{1/2} \log n\right).$$

En utilisant l'algorithme d'ordonnancement à gros grain précédent, l'exécution de l'algorithme de tri est asymptotiquement optimale en temps pour $p \leq \frac{\sqrt{n}}{\log n}$.

5.4.2. *Ordonnancement de tâches indépendantes avec un ordre sur les coûts*

Dans ce paragraphe, nous considérons le sous-problème suivant de l'$ITSP$ intéressant en pratique : soit n tâches indépendantes t_i à ordonnancer, de durées ordonnées θ_i, $1 \leq i \leq n$ et vérifiant :

$$\theta_1 \geq \theta_2 \geq \ldots \geq \theta_n$$

Dans ce cas, nous supposons que l'algorithme en-ligne connaît l'ordre sur la durée des tâches, mais ignore la durée de ces tâches. L'algorithme *LPT* d'ordonnancement à base de listes qui affecte la tâche de durée maximale à un processeur devenant inactif a un facteur de compétitivité égal à [19] $\left(\frac{4}{3} - \frac{1}{3p}\right)$. En prenant en compte le nombre de tâches à ordonnancer, l'algorithme LPT est alors $\left(1 + \frac{1}{n}\frac{p-1}{p}\right)$–compétitif [8], ce qui montre l'optimalité asymptotique pour cet algorithme.

6. L'approche d'ATHAPASCAN

Le formalisme utilisé dans la définition du problème de l'exécution parallèle de la section 3.1. nous a ammené à définir un modèle de programmation parallèle (ATHAPASCAN-*1a*) qui permet d'ordonnancer des tâches selon leurs contraintes de précédence. Ce modèle de programmation parallèle distingue l'expression d'un algorithme donné du moyen dont il sera exécuté sur une machine donnée.

Considérons par exemple un programme parallèle d'élimination de Gauss pour une matrice dense à coefficients des nombres à virgule flottante. Sur une machine parallèle qui est reservée à l'exécution du programme, les durées des tâches peuvent être supposées connues et un ordonnancement cyclique permettra d'obtenir une exécution efficace, avec un surcoût d'ordonnancement négligeable.

Cependant, ceci n'est plus vrai si le même programme est exécuté sur un réseau de stations de travail partagées par plusieurs utilisateurs. Du fait de la non-prédictibilité de l'activité des autres utilisateurs, les durées de tâches de l'algorithme doivent être considérées comme inconnues. Dans ce cas, l'utilisation d'un algorithme d'ordonnancement en-ligne (par exemple un algorithme glouton d'ordonnancement) est nécessaire afin d'obtenir une exécution efficace quelque soit les circonstances[2]. De cette

2. qualité que nous pourrions appeler « LaTeX-résistant »...

manière, le programme est inchangé, seul l'algorithme d'ordonnancement est spécialisé.

ATHAPASCAN-1a est une bibliothèque C++ basée sur le noyau d'exécution ATHAPASCAN-0. Le noyau ATHAPASCAN-0 permet la création de *thread* à distance, des communications de groupes et l'accès à une mémoire globale. Il est construit au-dessus de la bibliothèque de communication MPI et d'un noyau de thread POSIX.

6.1. Construction du graphe de précédence, interprétation et exécution

L'exécution d'un programme consiste à la construction d'un graphe de précédence puis de son ordonnancement. Chaque nœud du graphe correspond à une tâche. Un nœud du graphe peut être un nœud d'entrée, un nœud de sortie ou bien un nœud de calcul.

6.1.1. *Construction du graphe*

L'utilisateur de la bibliothèque ATHAPASCAN-1a doit définir une classe concrète qui dérive de la classe abstraite `ExecutionGraph` qui décrit le graphe de précédence du programme. Un nœud correspond à la description de l'exécution d'une tâche et est un objet de classe `Elementary`. Chaque objet correspondant à une tâche est du type d'une classe concrète dérivant de la classe abstraite `Task`.

La classe `ExecutionGraph` offre deux types d'opérateurs : des méthodes d'accès et des méthodes de construction. Ces dernières permettent de spécifier des relations de précédence entre deux graphes G_1 et G_2 :

– composition séquentielle : `G1.After(G2)`. Le graphe G_2 ne sera ordonnancé qu'après la fin d'exécution de toutes les tâches de G_1.

– composition parallèle indépendante : `G1.IndependentParallel( G2 )`. Toutes les tâches de G_1 sont indépendantes avec toutes celles de G_2.

– composition parallèle concurrente : `G1.ConcurrentParallel( G2 )`. Les tâches de G_1 et celles de G_2 doivent être exécutées simultannément. Ce cas correspond à celui où au moins deux tâches (une de G_1 et une G_2) sont suceptibles de communiquer.

– composition fonctionnelle : `G1.LinkedUp( G2 )`. La i-ème tâche de sortie de G_1 fournit un résultat servant d'entrée à la i-ème tâche d'entrée de G_2.

6.1.2. *Interprétation*

A partir de cette classe `ExecutionGraph`, la bibliothèque ATHAPASCAN-1a offre des classes qui implémentent des schémas de calcul classiques. Par exemple, le schéma de décomposition parallèle du type *découpe-calcul-fusion* (*i.e. Split-Compute-Merge*) peut être facilement décrit par spécialisation de la classe abstraite `SplitComputeMerge`.

Par dérivation, l'utilisateur spécifie seulement les méthodes de découpe (*i.e.* la méthode `Split`) et de fusion (*i.e.* la méthode `Merge`) et le type de la tâche à exécuter. De même, par dérivation de la classe abstraite `Task`, l'utilisateur définit les méthodes d'initialisation (*i.e.* définition du *stub* serveur correspondant au *stub* client de la méthode `Split`), de calcul et de terminaison (*i.e.* définition du *stub* serveur associé au *stub* client de la méthode `Merge`).

6.1.3. *Exécution*

Chaque objet `G` de la classe `ExecutionGraph` est ordonnancé avec un objet séquenceur `S` grâce à la méthode `S.Execute(` **in** `G ).`

Les objets de type séquenceur sont des instances d'une classe concrète dérivant de la classe abstraite `Scheduler` qui offre une interface standard pour l'implémentation des algorithmes d'ordonnancement. Celle-ci est :

– `Execute(` **in** `Graph )` : cette méthode est redéfinie dans chaque classe concrète dérivant de la classe `Scheduler` et implémente l'algorithme d'ordonnancement.

– `ExecuteOn(`**in** `Elementary,`**in** `Site )` : cette méthode est utilisée dans les classes dérivées pour exécuter la tâche associée à un nœud du graphe sur le processeur choisi.

En outre, l'implémentation d'un algorithme d'ordonnancement utilise les opérateurs d'accès aux nœuds du graphe.

6.2. Extensions : localité et granularité

Le modèle de programmation ATHAPASCAN-1a offre la possibilité de spécifier des informations sur la localité et la granularité associées à un algorithme :

spécification de la localité : chaque nœud élémentaire du graphe peut être attribué par une information de localité (typiquement le site sur lequel une tâche antérieure au nœud a été précédemment exécutée).

adaptation de la granularité : l'intervalle possible du choix du facteur de découpe des objets de classe dérivant de `SplitComputeMerge` peut être défini. Cela est particulièrement intéressant pour les algorithmes réguliers : un algorithme d'ordonnancement peut choisir le facteur de découpe approprié en fonction du nombre de processeurs disponibles.

Outre ces spécifications, des informations de coût peuvent être associées aux tâches à exécuter. La sémantique de ces informations dépend de l'algorithme d'ordonnancement choisi. Par exemple, pour un algorithme à base de listes, ces informations de coût peuvent correspondre à des priorités.

6.3. Résultats expérimentaux sur le tri-rapide

Nous considérons ici l'implémentation de stratégies de régulation pour l'algorithme $\mathcal{A}^{(1)}$ (*cf* paragraphe 3.3.). Nous avons vu que cet algorithme est de régularité polynômiale et que l'algorithme d'ordonnancement à gros grain (§5.4.1.) est théoriquement optimal pour tout $p \leq \frac{\sqrt{n}}{\log n}$.

Pour l'implémentation, deux stratégies d'ordonnancement sont considérées : la première utilise un placement en tenant compte des informations de coût, la seconde correspond à un algorithme glouton centralisé avec des seuils. Les expérimentations ont été effectuées pour différentes tailles de tableau sur un IBM-SP2 à 32 processeurs.

Algorithme avec prédiction de coût. Si n_i est le nombre d'éléments du sac B_i, le coût du calcul du tri en séquentiel est estimé à $n_i \log n_i$, le coût moyen de l'algorithme de tri-rapide.

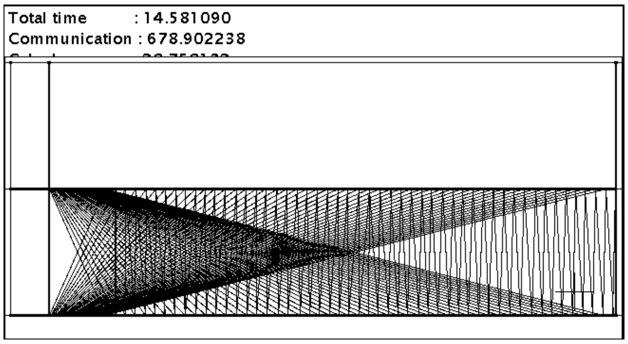

FIG. 1 – *Placement de l'algorithme sur 2 processeurs. Les deux lignes en gras (horizontales) représentent les threads, les autres lignes les communications. L'axe horizontal représente le temps.*

La figure 1 montre l'exécution de l'algorithme avec un algorithme d'ordonnancement à base de listes. Le coût des communications est masqué par celui des calculs. Une petite variation de l'algorithme séquentiel de tri-rapide permet de rendre cette stratégie efficace sur une machine uniforme.

Algorithme glouton centralisé. Les tâches à exécuter sont centralisées sur un processeur, appelé le *maître* et la charge des processeurs *esclaves* représente le nombre de tâches s'exécutant. Lorsque le nombre r_i de tâches d'un processeur esclave devient plus petit que le seuil S_m, le processeur demande au maitre $S_M - r_i$ tâches. La figure 2 montre les performances obtenues. Le choix des valeurs S_m et S_M est très important pour l'efficacité de l'exécution. Cette stratégie donne de bons résultats : 10 millions d'entiers sur 32 bits ont été trié en $156s$ en séquentiel et $8s$ sur 29 processeurs, l'efficacité correspondante est de 67%.

7. Conclusions

La théorie des algorithmes en-ligne peut être utilisée pour l'analyse des performances des algorithmes dynamiques d'ordonnancement. Deux surcoûts ont été considérés : le coût de la détermination d'un ordonnancement et sa qualité. Ces deux coûts sont liés à l'algorithme parallèle exécuté.

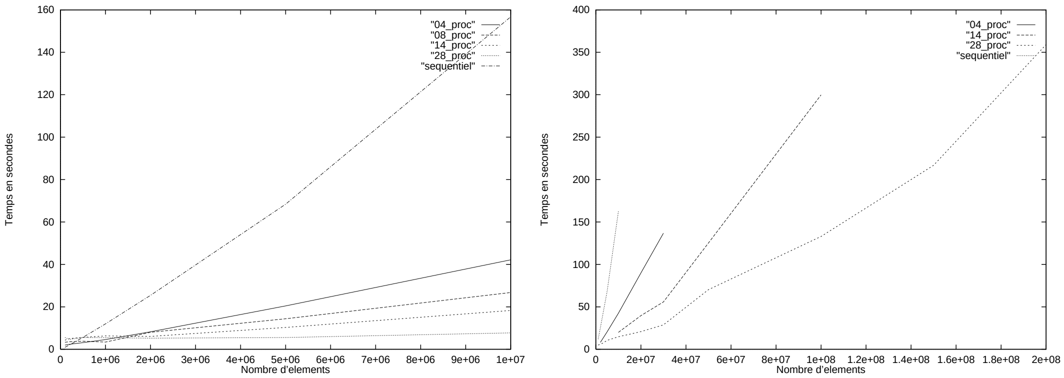

FIG. 2 – *Accélérations obtenues pour le tri de tableaux de diverses tailles*

Le premier est englobé par la définition de la complexité d'ordonnancement qui nous a servi à définir la régularité d'un algorithme. Intuitivement, plus un algorithme est irrégulier, plus il est difficile de l'ordonnancer afin d'obtenir une exécution efficace.

Le second surcoût est mesuré par le facteur de compétitivité de l'algorithme d'ordonnancement. Même pour un graphe de précédence inconnu, l'algorithme d'ordonnacement glouton, sur le modèle p-PRAM (p processeurs identiques), est $(2 - \frac{1}{p})$-compétitif et atteint la borne inférieure de compétitivité d'un algorithme en-ligne déterministe.

Afin d'améliorer ce facteur de compétitivité, il convient de restreindre le problème de l'exécution parallèle à des familles de graphes particuliers, comme les ensembles de tâches indépendantes.

Remerciements

Les auteurs tiennent à remercier Gilles Villard pour sa contribution dans la définition du problème de l'ordonnancement d'un graphe. L'introduction aux algorithmes en-ligne et à gros grain pour l'ordonnancement de tâches indépendantes a été inspirée par François Galilée. Mathias Doreille a contribué à cet article grâce à son travail sur les algorithmes parallèles, en particulier le tri-rapide et l'élimination de Gauss avec durée inconnue. Gerson Gavalheiro nous a aidé à définir et à implémenter la hiérarchie de classes d'ATHAPASCAN-1.

Références

[1] A. AGGARWAL, A. CHANDRA, AND M. SNIR, *Communication complexity of PRAM's*, Theoretical Computer Science, 71 (1990), pp. 3–28.

[2] J. BALCÁZAR, J. DIAZ, AND J. GABARRÓ, *Structural Complexity II*, Springer-Verlag, 1989.

[3] S. BEN-DAVID, A. BORODIN, R. KARP, G. TARDOS, AND A. WIGDERSON, *On the power of randomization in online algorithms*, in 22nd Annual ACM sym-

posium on Theory of Computing, N. Y. ACM, ed., May 1990, pp. 379–386. Baltimore, Maryland.

[4] G. BERNARD, D. STEVE, AND M. SIMATIC, *Placement et migration de processus dans les systèmes reépartis faiblement couplés*, TSI, 10 (5) (1991), pp. 375–392.

[5] J. BLAZEWICZ, K. EXKER, G. SCHMIDT, AND J. WĘGLARZ, *Scheduling in Computer and Manufacturing Systems*, Springer-Verlag, 1993.

[6] R. BRENT, *The parallel evaluation of general arithmetic expressions*, J. ACM, 21 (1974), pp. 201–206.

[7] T. CASAVANT AND J. KHUL, *A taxonomy of scheduling in general-purpose distributed computing sy stems*, IEEE Transactions on Software Engineering, 14 (1988), pp. 141–154.

[8] E. COFFMAN AND S. R., *A Generalized Bound on LPT Sequencing*, RAIRO Informatique, 10 (1976), pp. 17–25.

[9] R. COLE AND U. VISHKIN, *Approximate Parallel Scheduling. Part I : The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time*, SIAM J. Computing, 17 (1988).

[10] S. COOK, *A taxonomy of problems with fast parallel algorithms*, Information and Control, 64 (1985), pp. 2–22.

[11] M. COSNARD, *A comparison of parallel machine models from the point of view of scalability*, in proceedings of the 1rst Int. Conf. on Massively Parallel Computing Systems, Ischia, Italy, IEEE Computer Society Press, May 1993, pp. 258–267.

[12] M. COSNARD AND D. TRYSTRAM, *Parallel Algorithms and Architectures*, Thomson Computer Press, 1995.

[13] P. EMDE BOAS, *Machine models and simulations*, in Algorithms and Complexity, J. van Leuwen, ed., Elsevier, 1990, pp. 869–932.

[14] A. FELDMANN, M.-Y. KAO, AND J. SGALL, *Optimal Online Scheduling of Parallel Jobs with Dependencies*, in Proceedings of the 25th ACM Symposium on Theory of Computing, ACM Press, 1993, pp. 642–651.

[15] D. FERRARI AND S. ZHOU, *An empirical investigation of load indices for load-balancing applications*, in Proc. Performance'87, 12th IFIP WG7.3 International Symposium on Computer Performance, Brussels Belgium, Elsevier Science Publishers, 1987.

[16] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proceedings of the 10th ACM Symposium on Theory of Computing, ACM Press, 1978, pp. 114–118.

[17] M. GAREY AND D. JOHNSON, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.

[18] T. GAUTIER, J. ROCH, AND G. VILLARD, *Regular versus irregular problems and algorithms*, in Proc. of IRREGULAR'95, Lyon, France, Springer-Verlag – L.N.C.S. 980, Sep. 1995, pp. 1–26.

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

[20] J. JÁJÁ, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[21] R. KARP AND V. RAMACHANDRAN, *Parallel algorithms for shared-memory machines*, in Algorithms and Complexity, J. van Leuwen, ed., Elsevier, 1990, pp. 869–932.

[22] C. KRUSKAL, L. RUDOLPH, AND M. SNIR, *A complexity theory of efficient parallel algorithms*, Theoretical Computer Science, 71 (1990), pp. 95–132.

[23] D. LIFKA, M. HENDERSON, AND K. RAYL, *User's Guide to the Argonne SP Scheduling System*, Tech. Rep. ANL/MCS-TM-201, Argonne National Lab, May 1995.

[24] M. PALIS, J.-C. LIOU, S. RAJASEKARAN, S. SHENDE, AND D. LEI, *Online scheduling of dynamic trees*, Parallel Processing Letters, 5 (1995), pp. 635–646.

[25] A. RANADE, *A framework for analyzing locality and portability issues in parallel computing*, in Parallel Architectures and their Efficient Use, Springer-Verlag L.N.C.S. 678, 1993, pp. 185–194.

[26] R. REISCHUK, *Probabilistic parallel algorithms for sorting and selection*, SIAM J. Computing, 14 (1985), pp. 396–409.

[27] D. B. SHMOYS, J. WEIN, AND P. WILLIAMSON, *Scheduling parallel machines on-line*, SIAM, J. Comput., 24 (1995), pp. 1313–1331.

[28] D. D. SLEATOR AND R. E. TARJAN, *Amortized efficiency of list update and paging rule*, Communication of the ACM, 28 (1985), pp. 202–208.

[29] L. VALIANT, *A bridging model for parallel computation*, Communication ACM, 33 (1990), pp. 103–111.

[30] ———, *General purpose parallel architectures*, in Algorithms and Complexity, J. van Leuwen, ed., Elsevier, 1990, pp. 944–971.

[31] M. WILLEBEEK-LE-MAIR AND P. REEVES, *Strategies for dynamic load-balancing on higly parallel computers*, IEEE Parallel and Distributed Systems, 4 (1993), pp. 979–993.

[32] S. ZHOU, *A trace-driven simulation study of dynamic load-balancing*, IEEE Trans. on Software Engineering, 14 (1988).