Chapitre de l'ouvrage *Parallélisme et applications irrégulières*, **HERMES, Paris 1995, p. 71–84**

Algorithmes irréguliers et ordonnancement

Jean-Louis ROCH - Gilles VILLARD (LMC-IMAG), Catherine ROUCAIROL (PR*i*SM)

Dans le cadre du projet STRATAGÈME, les différentes applications considérées ont toutes ont en commun le fait que le graphe de précédence des tâches qui les composent dépend fortement des données en entrée. La difficulté qui apparaît lors de la parallélisation de telles applications est alors la distribution des calculs aux processeurs, de façon à garantir une efficacité pratique. Il est alors nécessaire de faire appel à des mécanismes adaptés de régulation de charge. La synthèse des travaux réalisés dans STRATAGÈME nous a permis de dégager une définition de l'irrégularité d'un algorithme, et une méthodologie pour le traitement des applications dont l'exécution est fortement corrélée aux données en entrée.

Le but de ce chapitre est d'introduire cette définition [1], et de présenter cette méthodologie en l'illustrant par différents exemples étudiés dans le cadre de STRATAGÈME et présentés plus finement au fil des chapitres de cet ouvrage.

1 Introduction

Les travaux en complexité parallèle visent à classifier les problèmes en fonction de leur parallélisme intrinsèque [6, 2] grâce à la notion de NC-réductibilité. Au delà de cette classification, l'algorithmique parallèle vise à la construction, sur des modèles théoriques généraux tels que la PRAM (*Parallel Random Access Machine* [15]), d'algorithmes qui soient à la fois très parallèles (donc extensibles) et qui n'effectuent pas un nombre d'opérations (ou travail [20, 2]) d'un ordre supérieur à celui du meilleur algorithme séquentiel résolvant le même problème : ce critère peut être caractérisé par la notion de travail optimal et quantifié par celle d'efficacité (ou d'inefficacité [22, 31]).

Une difficulté majeure qui apparaît lors de la programmation d'un tel algorithme parallèle, optimal d'un point de vue théorique, sur une architecture parallèle est la nécessité de limiter les surcoûts liés aux communications et à l'ordonnancement. Le surcôut de communication est lié à l'impossibilité de disposer aujourd'hui d'architectures capables de simuler efficacement une PRAM. Le surcoût d'ordonnancement a pour origine la difficulté comme nous le verrons ci-dessous à mettre en œuvre le principe de Brent.

Communication : localité. De nombreux travaux se sont attachés à caractériser le surcoût lié aux communications sur le modèle PRAM en particulier grâce à la notion de localité (*gross locality*), qui correspond au rapport entre les complexités de calcul et de communication [29]. Dans [29], la localité d'un problème est définie comme le rapport entre le travail du meilleur algorithme PRAM qui le résout et la complexité des communications sur deux processeurs.
Un algorithme pour la résolution d'un problème non local ne peut être implémenté efficacement que sur une architecture offrant une couche de communication très performante. Autrement dit, tant qu'une PRAM ne pourra être efficacement simulée, la maîtrise de la localité est nécessaire pour obtenir des programmes performants.
La localité peut aussi être vue comme un critère de *régularité*. Ainsi, les problèmes

1. Une définition plus générale de l'irrégularité, comparée à la notion de localité [29] est donnée dans [17].

qui induisent des schémas de communication (ou d'accès mémoire) simples ou prévisibles permettent plus facilement la construction de programmes ayant une forte localité. Cependant, une autre clé pour l'efficacité est la régularité des schémas globaux de communication utilisés [7], indépendamment de la localité. Une remarque importante quand nous la transposerons au cadre de l'ordonnancement est que le concept de localité repose sur une distinction entre tâches de calcul et tâches de communication [36].

Ordonnancement: irrégularité en pire cas. L'ordonnancement d'un programme parallèle consiste à affecter à chacune des tâches qui le constituent une date et un site d'exécution. Un "bon" ordonnancement (permettant d'exécuter l'ensemble des tâches dans un temps faible) est donc crucial pour l'obtention de performances. Ainsi, des solutions optimales pour des problèmes clefs, tels que la numérotation des éléments d'une liste – *list ranking* –, peuvent reposer entièrement sur une stratégie d'ordonnancement [5]. Par ailleurs, de nombreux algorithmes parallèles génèrent dynamiquement des tâches de calculs de coûts inconnus pouvant être effectuées en parallèle, et ce de manière non prévisible car fortement dépendant des valeurs en entrée. Ceci montre la difficulté du problème dans les cas qui vont nous intéresser.

Les concepts permettant de caractériser de tels problèmes sont moins bien définis que dans le cadre des communications. Par exemple, le principe de Brent [3] énonce que tout algorithme parallèle synchrone s'exécutant en temps t et qui effectue x opérations peut être implémenté sur p processeurs en temps $\lceil x/p \rceil + t$. Ce principe ne prend pas en compte le surcoût lié au calcul d'un ordonnancement permettant de réaliser la simulation sur p processeurs [5].
De manière générale, la démonstration de ce principe repose sur une numérotation des tâches de l'algorithme parallèle initial selon leur profondeur dans le graphe de précédence. Une allocation cyclique (modulo p) des tâches permet alors de les ordonnancer sans surcoût de manière à obtenir le résultat énoncé.
Mais dans le cas général où des processeurs de l'algorithme initial effectuent des opérations vides (*nop*), qui ne sont pas prises en compte dans le nombre x d'instructions, et ce de manière non structurée et dépendant des données en entrée, la numérotation des tâches ne peut être réalisée sans surcoût.

De manière générale, les algorithmes peuvent être séparés en deux classes : *statique* et *dynamique* [24]. Les premiers sont caractérisés par le fait que la structure de leurs exécutions est connue par avance et reste la même quelles que soient les données en entrée. Au contraire, la séquence des instructions effectuées lors de l'exécution d'un algorithme dynamique varie selon les données en entrée. Mais cette classification n'est pas satisfaisante car elle caractérise plus la manière dont l'ordonnancement peut être calculé (avant ou pendant l'exécution) que les performances qu'il permet d'obtenir.

Aussi, est-il important de pouvoir mesurer la difficulté d'ordonnancer les tâches d'un algorithme parallèle pour obtenir des exécutions efficaces, en prenant en compte le surcoût lié à l'ordonnancement. C'est dans ce cadre que nous définissons l'irrégularité d'algorithmes. Cette mesure est introduite de manière similaire à la localité pour mesurer le surcoût intrinsèque lié aux communications. Intuitivement, plus un algorithme est irrégulier, plus coûteux il est d'ordonnancer les tâches qui le constituent pour obtenir une exécution efficace, et ce indépendamment de l'algorithme utilisé pour calculer l'ordonnancement.

2 Irrégularité

2.1 Modèle parallèle et problème d'ordonnancement

Le modèle parallèle que nous considérons est celui d'une p-PRAM (CREW-PRAM comportant p processeurs indexés). A un premier niveau, chaque processeur calcule séquentiellement avec sa mémoire locale. A un deuxième niveau, les processeurs peuvent communiquer via une mémoire globale partagée ; les lectures à une même adresse en mémoire globale peuvent être concurrentes, mais les écritures sont exclusives.

Le parallélisme est généralement exprimé à l'aide de l'instruction suivante :

`for all` $j \in J$ `in parallel do` instruction(j)

qui affecte à chaque élément j dans J le processeur indexé code(j) qui est déterminé à partir de j en temps constant. Mais ici, en généralisant l'instruction *fork* [15], de manière à faire apparaître d'une part la possibilité de générer dynamiquement du parallélisme, et d'autre part la contribution demandée à l'algorithme d'ordonnancement pour obtenir une exécution efficace, le parallélisme est ici exprimé par l'instruction suivante :

`for all` $t \in \mathcal{T}$ `in parallel do schedule`(t) (1)

où $\mathcal{T}$ est un ensemble de tâches (ou *groupe*). L'exécution d'une telle instruction consiste à ordonnancer les tâches de $\mathcal{T}$ de façon à ce qu'elles soient exécutées en temps optimal.

Le processeur qui effectue un tel appel ne peut passer à l'instruction suivante que lorsque toutes les tâches de $\mathcal{T}$ sont terminées ; il peut cependant être utilisé par l'algorithme d'ordonnancement pour exécuter certaines des tâches de $\mathcal{T}$. Une tâche est un programme séquentiel indivisible ; nous supposons de plus qu'une fois affectée à un processeur, une tâche s'exécute entièrement sur ce processeur (pas de migration). La longueur d'une tâche est son temps séquentiel ; il est *a priori* inconnu tant que la tâche n'est pas terminée. Une tâche ne peut faire aucune hypothèse sur la manière dont sont ordonnancées les tâches de son groupe ; les tâches d'un même groupe ne peuvent donc pas directement communiquer entre elles via la mémoire partagée.

Les processeurs et les dates d'exécution des tâches sont décidés par un algorithme d'ordonnancement (ou *régulateur*) qui résout le problème TSP (*Task Scheduling Problem*[2] [5]) suivant.
En entrée, n tâches indépendantes t_i, $1 \leq i \leq n$, sont données. La tâche t_i est de longueur l_i comprise entre 1 et $c(n)$, et $\sum_{i=1}^{n} l_i = w(n)$. Le problème TSP consiste à ordonnancer les tâches sur la p-PRAM de façon à ce qu'elles soient toutes terminées en temps $O(\max\{w(n)/p, c(n)\})$. Si p est borné par $w(n)/c(n)$, ce temps est – asymptotiquement – optimal.

Le problème TSP est supposé résolu avec un surcoût égal au nombre de tâches soumises $\#\mathcal{T}$ divisé par le nombre de processeurs.

2.2 Complexité d'ordonnancement d'un algorithme

L'exécution d'un algorithme parallèle sur un jeu d'entrée donné peut donc être décrite par un graphe de précédence de type série-parallèle (graphe SP 1 1 [13]). De manière générale, le *problème d'ordonnancement* consiste alors, à partir de la connaissance de ce graphe, à ordonnancer l'ensemble des tâches qui le constituent sur les p processeurs de manière à minimiser le temps d'exécution. Une solution à ce problème doit donc tenir compte des surcoûts liés à la fois au routage des communications [37] et au calcul de l'ordonnancement.

2. Nous reprenons ici la terminologie utilisée dans [5], à ne pas confondre avec *Traveling Salesman Problem* !

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 apparaissant similaires. Ainsi, il peut être possible de caractériser un régulateur par la structure des communications qu'il induit [14]. Réciproquement, le problème général de communications pluralistes peut être ramené à un tri et à un ordonnancement [35].

Lorsque le routage est seul considéré, le problème est de gérer les transferts de données (ou de manière équivalente les accès mémoire) ; les données sont découpées en paquets et communiquées selon une certaine politique [36]. Le surcoût de communication peut alors être assimilé au coût des communications effectuées sur un modèle de machine donné, par exemple XRAM [37] pour un modèle à mémoire distribuée ou LPRAM [1] pour un modèle à mémoire partagée, ces coûts pouvant en effet être quantifiés sur de telles modèles.
Généralement, lorsque le problème de routage est considéré, émetteurs et destinataires son connus. Le problème peut être résolu statiquement (avant l'exécution) ou dynamiquement (pendant l'exécution). Une exécution peut consister en une succession de phases de calcul et de communication [37], ou ces deux phases peuvent être effectuées de manière asynchrone [7].

Lorsque le problème d'ordonnancement est considéré, un régulateur gère l'exécution des tâches générées par l'algorithme. Il consiste en une mesure de la charge de la machine et une politique qui décide de l'affectation des tâches [39, 4]. Le surcoût d'ordonnancement comprend alors non seulement le coût de création des tâches mais aussi le coût de l'estimation de la charge [11]. La mesure de cette charge ne peut être effectuée qu'à partir de la connaissance de l'état courant au sein du graphe de précédence associé à l'exécution (tâches en cours d'exécution, tâches en attente, nouvelles tâches à créer, éventuellement prédiction sur les tâches à venir). Aussi, il apparaît naturel de définir l'irrégularité d'un algorithme en fonction de la difficulté intrinsèque à construire ce graphe (i.e. à mesurer la charge) et à l'ordonnancer ; plus l'ordonnancement est complexe à calculer, plus l'algorithme est irrégulier.

Définition 2.1 *La* complexité d'ordonnancement *de l'instruction (1) est :* $Max(1, \#\mathcal{T}/p)$. *Si l'instruction est appelée plusieurs fois séquentiellement, la complexité s'aditionne. Si l'instruction est appelée simultanément sur k processeurs avec comme ensembles de tâches respectivement* $\mathcal{T}_1, \ldots, \mathcal{T}_k$*, la complexité est* $Max(1, (\#\mathcal{T}_1 + \ldots + \#\mathcal{T}_k)/p)$. *La* complexité d'ordonnancement de l'exécution d'un algorithme *est égale à la somme des complexités d'ordonnancement des instructions (1) effectuées.*

2.3 Irrégularité

Même si pour un algorithme dynamique, le graphe de précédence des tâches (et le coût) lors de l'exécution est directement dépendant des données en entrée, d'un point de vue théorique, seules les exécutions en pire cas sont généralement considérées. Dans ce chapitre, nous nous limitons à l'étude dans ce cadre, en définissant l'irrégularité en pire cas.

Définition 2.2 *L'irrégularité* $\iota_w(n)$ *(en pire cas) d'un algorithme est la plus grande des complexités d'ordonnancement de ses exécutions pour toute entrée de taille n.*

Cette définition est plus centrée sur un algorithme donné que sur le problème qu'il résout.

Un algorithme séquentiel quel qu'il soit est d'irrégularité constante. Pour un algorithme dynamique, qui est efficace exprimé en utilisant l'instruction (1), son irrégularité mesure la contribution qui est demandée au régulateur pour assurer son efficacité.

Dans tous les cas, *l'irrégularité $\iota_w(n)$ d'un algorithme est bornée par son temps d'exécution*, donc en particulier *l'irrégularité $\iota_2(n)$ d'un algorithme écrit pour p processeurs et exécuté sur deux processeurs est bornée par son travail*. De manière analogue à ce qui est fait pour la localité [29], il apparaît maintenant naturel de classifier les algorithmes en fonction de l'importance de la contribution qu'ils demandent au régulateur.

Définition 2.3 *Un algorithme est dit :*

- *régulier si* $\iota_2(n) = O(1)$
- *log-irrégulier si* $\iota_2(n) = (\log w(n))^{O(1)}$
- *irrégulier si* $\iota_2(n) = w(n)^{O(1)}$

Un algorithme statique [24] (i.e. le graphe de précédence reste le même indépendamment des données en entrée), qui peut facilement être écrit en utilisant un ordonnancement pré-calculé, est donc régulier. Ainsi, le calcul de la transformée de Fourier (FFT) à n points peut être implémenté sans difficulté par un algorithme régulier et efficace sur n processeurs.
En résolvant le problème TSP, il a été établi que le problème de la numérotation des éléments d'une liste (*list-ranking*) peut être résolu optimalement en temps logarithmique [5]. Cet algorithme peut être facilement écrit en utilisant un régulateur : il est alors d'irrégularité $O(\log n)$ puisqu'un seul appel est effectué avec n tâches sur $p = n/\log n$ processeurs. Si l'on modifie cet algorithme pour intégrer la résolution du TSP en suivant le schéma proposé dans [5], l'algorithme obtenu est alors régulier. Il est à noter que l'algorithme direct pour résoudre ce problème sur n processeurs [21] est facilement implémentable avec une irrégularité constante, mais est inefficace.

Remarque. La restriction à l'étude de l'irrégularité en pire cas est limitative, notamment pour des algorithmes résolvant en temps séquentiel exponentiel un problème NP-complet. Un tel algorithme peut très bien être évalué, dans le pire cas, comme efficace et d'accélération polynomiale (classe EP [22, 31]) alors qu'il peut être intrinsèquement difficile d'ordonnancer ses exécutions de façon à ce que, pour tout jeu de données en entrée, l'exécution associée soit efficace. Ce problème est illustré par des anomalies d'accélérations expérimentales dans différents domaines [23, 38, 30, 25]. Pour arriver à une caractérisation plus précise, il est alors nécessaire d'étendre la définition de l'irrégularité de façon à prendre en compte l'ensemble des exécutions, soit par des estimations statistiques, soit en considérant l'efficacité obtenue pour chaque instance (efficacité locale) [17].

3 Irrégularité et granularité

Si le nombre de tâches d'un algorithme parallèle reste le même quel que soit le nombre de processeurs utilisés (algorithme "size-dependent" [22]), son irrégularité peut croître lorsque le nombre p de processeurs diminue.

Pour que l'inefficacité reste bornée indépendamment du nombre de processeurs utilisés, il est donc nécessaire de considérer un algorithme pour lequel le nombre de tâches parallèles est lié au nombre p de processeurs (algorithme "size-independent" [22]). En conséquence, le grain de l'algorithme doit être adapté à celui de la machine pour conserver une irrégularité indépendante du nombre de processeurs. Cette remarque peut être illustrée à partir des deux exemples précédents avec $p < n$ processeurs.

L'algorithme de grain le plus fin pour le calcul de la FFT est d'irrégularité $O(n/p)$, puisque constitué de n tâches. Il est cependant facile de regrouper les tâches de grain fin pour construire un algorithme constitué de p tâches et qui s'exécute aussi en temps $\theta(n \log n/p)$: adapter le grain permet alors de garder l'irrégularité constante. Concernant l'algorithme de numérotation d'une liste, le nombre de tâches générées est toujours n, quel que soit le nombre de processeurs utilisés : l'irrégularité est alors $O(n/p)$. Une technique de diminution du nombre de tâches pour ce problème [21] permet ici de construire un algorithme optimal, de temps $O(n/p)$ et d'irrégularité constante.

De manière générale, l'irrégularité est donc intrinsèquement associée à un algorithme. D'un point de vue pratique, la recherche de l'efficacité consiste alors à diminuer le surcoût lié à la régulation, et donc à chercher un algorithme non seulement efficace mais encore d'irrégularité la plus petite possible. Ces deux critères peuvent apparaître antagonistes. Pour une application fortement dynamique, qui génère des calculs parallèles de manière non-prévisible, la recherche de l'efficacité vise à exploiter au maximum le parallélisme de manière à occuper au mieux les processeurs. Mais cela peut entraîner une croissance du nombre de tâches à ordonnancer et, par conséquence, une augmentation de l'irrégularité et du surcoût lié à la régulation.

Dans la section suivante, différentes techniques permettant de maîtriser l'irrégularité sans une perte trop importante d'efficacité sont introduites à partir des applications étudiées dans le cadre de STRATAGÈME.

4 Diminuer l'irrégularité

L'irrégularité d'un algorithme est directement reliée à la complexité d'ordonnancement de ses exécutions. Cette complexité est elle-même définie à partir des instructions (1) d'appel au régulateur effectuées lors d'une exécution. Ces instructions construisent en fait le graphe de précédence de l'exécution. Pour minimiser la complexité d'ordonnancement on peut donc s'intéresser au choix de la granularité et chercher à utiliser le graphe de précédence.

4.1 Irrégularité et granularité

Diminuer l'irrégularité d'un algorithme se ramène à diminuer le nombre d'instructions (1) effectuées lors de son exécution. Sans modifier les instructions de calcul effectuées par l'algorithme, il peut être possible de diminuer son surcoût d'ordonnancement, soit en supprimant les instructions (1) qu'il comporte par un groupement de plusieurs tâches parallèles en une tâche séquentielle, soit en remplaçant plusieurs de ces instructions par d'autres ayant une complexité d'ordonnancement inférieure. Ces instructions décrivant le graphe de précédence de l'exécution, cette diminution est possible dès lors qu'il est possible de décrire ce graphe d'une manière différente, ayant une complexité d'ordonnancement inférieure.

Considérons l'exemple du calcul parallèle au grain le plus fin du produit itéré d'une séquence de n éléments [2], le produit de deux éléments étant supposé de temps constant. Le graphe de précédence de l'algorithme parallèle le plus simple est un arbre binaire équilibré : il comporte $n-1$ tâches et a une profondeur de $\log n$.
Ce graphe peut être facilement décrit par un algorithme récursif qui, à chaque étape, génère deux tâches. L'irrégularité de cet algorithme exécuté sur deux processeurs est n, il est donc *irrégulier*.
Il est cependant possible de décrire le même graphe par un autre algorithme, de manière itérative, par une boucle sur le nombre de pas parallèles. Au pas k, une seule instruction d'appel au régulateur est effectuée qui demande l'ordonnancement de $\frac{n}{2^k}$ tâches de calcul. L'irrégularité de cet algorithme itératif exécuté sur deux processeurs est alors $\log n$, il est donc *log-irrégulier*.

Compte-tenu de la structure du graphe de précédence, il est possible de diminuer encore l'irrégularité. En effet, il est facile de calculer sans surcoût une numérotation des tâches qui respecte l'ordre partiel de leur précédence. A partir de cette numérotation, l'affectation sur $p < \frac{n}{\log n}$ processeurs de la tâche i au processeur $(i \bmod p)$ réalise un ordonnancement optimal. Cet ordonnancement peut être intégré à l'algorithme, et ce indépendamment de la taille n de l'entrée et de sa valeur ; l'algorithme obtenu ne fait alors, après son lancement, plus aucun appel au régulateur. Il est d'irrégularité constante, donc *régulier*.

La diminution de l'irrégularité est donc liée à la possibilité de grouper plusieurs des tâches du graphe de précédence, soit séquentiellement pour les remplacer par une seule, soit en parallèle pour les ordonnancer au sein d'un même groupe.

De par la définition de la classe NC sur le modèle circuit booléen arithmétique [6, 21], tout problème dans NC peut être décrit par une famille log-uniforme de circuits booléens. La log-uniformité impose l'existence d'un algorithme séquentiel s'exécutant en espace mémoire logarithmique qui, étant donnée la taille n de l'entrée, génère en sortie un circuit de profondeur poly-logarithmique et ayant un nombre polynomial de portes qui résout le problème pour toutes les instances de taille n.
Le circuit généré correspond directement au graphe de précédence d'un algorithme parallèle résolvant le problème. La condition d'uniformité précise que ce graphe est valide pour toutes les entrées de taille n.
La taille n étant donnée, il est donc possible de construire le graphe de précédence associé à toutes les exécutions sur une entrée de taille n. L'exécution étant synchrone, ce graphe peut être exécuté par un algorithme PRAM qui effectue au plus la profondeur de ce graphe (i.e. le temps parallèle) opérations d'ordonnancement. Tout problème dans NC peut donc être résolu par un algorithme très parallèle avec une irrégularité poly-logarithmique, indépendamment de l'existence d'un algorithme parallèle efficace pour le résoudre.

4.2 Irrégularité et graphe de précédence

L'irrégularité d'un problème est intrinsèquement liée à la possibilité de construire, totalement ou partiellement, le graphe de précédence des exécutions de l'un des algorithmes le résolvant pour toutes les entrées de taille n.
Dans les paragraphes suivants, trois cas sont distingués pour étudier l'irrégularité d'un algorithme donné.

- Le graphe est *prévisible* : à partir de la seule connaissance de la taille des entrées, il est possible de construire le graphe de précédence.

- Le graphe est *semi-prévisible* : il est possible de structurer le graphe de précédence en une succession de phases, le sous-graphe correspondant à chaque phase pouvant être facilement construit à partir de la connaissance des données et des valeurs calculées dans les phases qui le précèdent.

- Le graphe est *imprévisible* : le graphe est fortement corrélé aux données en entrée et ne peut être découvert que lors de l'exécution des tâches qui le composent.

4.2.1 Graphe prévisible: approche statique

Lorsqu'un algorithme est statique [24], son graphe de précédence peut être déterminé indépendamment des données en entrée. Il est alors possible de calculer statiquement à partir de ce graphe un ordonnancement pour p processeurs [19], et de générer un programme d'irrégularité constante qui permet d'exécuter cet ordonnancement sans faire appel à un mécanisme dynamique de régulation.

Lorsque l'algorithme est dynamique, le graphe de précédence qu'il génère peut dépendre non seulement de la taille des données mais aussi de la valeur de certaines des données. L'utilisation d'un mécanisme d'évaluation partielle (i.e. une exécution séquentielle de l'algorithme parallèle ne prenant en compte que la valeur de ces données caractéristiques en n'effectuant pas les calculs portant sur les autres valeurs) permet alors la construction du graphe de précédence de l'exécution. Un ordonnancement statique de ce graphe permet alors de construire un nouveau programme, spécialisé pour le jeu de données en entrée, et régulier.

Dans un cadre général, cette technique peut s'avérer prohibitive, l'évaluation partielle se ramenant à une exécution du programme sur le jeu de données en entrée. Cependant, dans le cas particulier où la connaissance de quelques données permet une construction facile du graphe de précédence, elle peut être appliquée pour construire un programme parallèle performant.

Cette approche peut être illustrée par l'exemple de la factorisation de Choleski de grandes matrices creuses. De façon à minimiser le coût de calcul d'une telle factorisation, il est intéressant d'appliquer un algorithme de dissection emboîtée qui permet de structurer en blocs les coefficients non nuls de la matrice [10]. Malgré cette restructuration, le nombre d'étapes d'élimination étant liée à la dimension de la matrice, l'irrégularité reste polynomiale. Cependant, une interprétation symbolique, au niveau de ces blocs non-nuls, permet alors d'obtenir le graphe de précédence de l'algorithme de Choleski sur la matrice restructurée. Ce graphe est valué par le coût des tâches et les volumes de communication, déterminés à partir de de la taille des blocs seulement, sans effectuer les calculs d'élimination [10]. A partir de ce graphe il est alors possible d'effectuer un calcul statique d'un ordonnancement qui permet de construire un algorithme d'irrégularité constante et efficace en pratique.

4.2.2 Graphe semi-prévisible: structuration en phases

Certains algorithmes se présentent comme une succession (séquentielle) d'étapes (ou phases), chaque phase étant résolue par un algorithme parallèle. Un exemple typique de ce schéma est l'élimination de Gauss. L'irrégularité est dans ce cas lié au nombre de phases, et au nombre de création de tâches parallèles dans chaque phase. L'algorithme standard au grain le plus fin est d'irrégularité $O(n^3)$ sur n^2 processeurs, donc d'irrégularité polynomiale.

Une première approche pour diminuer l'irrégularité consiste alors à augmenter la granularité pour diminuer le nombre de tâches. Dans l'exemple de l'élimination de Gauss, en découpant cycliquement par colonnes par exemple la matrice en entrée, il est possible de résoudre le problème en ordonnancement seulement p tâches, chacune des tâches étant responsable de la succession des phases d'élimination sur un sous-ensemble de colonnes. La création de nouvelles tâches à chaque nouvelle phase est alors remplacée par la synchronisation des p tâches entre chaque phase. L'irrégularité devient alors constante, le nombre de tâches à ordonnancer étant égale au nombre de processeurs (supposé ici inférieur à la dimension de la matrice).

Néanmoins, il est possible qu'il soit difficile de construire les tâches initiales de façon à ce que la charge soit équilibrée durant le calcul. Pour reprendre le cas de l'élimination de Gauss, si le calcul du pivot est effectué selon une stratégie de "pivot total", il se peut qu'une tâche n'ait plus de colonnes à éliminer, alors que d'autres aient encore toutes leurs colonnes. La volonté de maîtriser l'irrégularité nuit alors à l'efficacité.

La technique générale consiste alors à construire, avant chaque phase et à partir des valeurs précédemment calculées, le graphe de précédence de cette phase pour l'ordonnancer au mieux de façon à garantir l'équilibre de la charge lors de chaque phase.
Cette stratégie se retrouve dans de nombreuses applications, par exemple en lancer

de rayons [16, 27] ou en simulation de particules en mouvement [28, 34].
Dans le cadre de Stratagème, comme présenté dans [2] ou dans ce volume, différentes applications font appel à cette technique, notamment en traitement d'images [26], en Calcul Formel [2] ou encore pour la gestion d'une machine dictionnaire [8, 9].

Deux techniques peuvent alors être envisagées pour ordonnancer les calculs d'une phase.

- Lorsqu'il est possible, après une phase, de déterminer le volume des calculs lors de la phase suivante, il est possible de réordonnancer directement les calculs pour la prochaine phase. Cette restructuration des calculs peut se ramener à une redistribution des données ([12], "ParList"). De manière duale, elle peut être réalisée en utilisant un mécanisme de migration des processus ([18]).
- Lorsqu'aucune connaissance n'est disponible, il est possible de n'ordonnancer qu'une partie des tâches, les autres tâches étant ordonnancées ultérieurement en utilisant une stratégie d'allocation à la demande ([18]). Le principe consiste à ne démarrer effectivement lors d'une nouvelle phase qu'une partie des tâches à effectuer pour garantir une charge suffisante sur les processeurs (stratégie à seuils), les autres tâches étant mises dans une "réserve" et ordonnancées lorsque nécessaire de façon à réguler la charge.

Ainsi, pour reprendre l'exemple de l'élimination de Gauss avec pivot total, la connaissance à la fin d'une phase des colonnes à éliminer lors de la phase suivante permet par une redistribution des données d'équilibrer la charge des processeurs. Il est ainsi possible de construire un nouvel algorithme, intégrant cette redistribution, et qui est à la fois d'irrégularité constante et efficace.

4.2.3 Graphe imprévisible

Les deux techniques précédentes ont consisté à diminuer l'irrégularité par la construction d'un nouvel algorithme intégrant lui-même une régulation des calculs de façon à minimiser le surcoût lié à l'ordonnancement.

Lorsque le problème est fortement non structuré, (par exemple en simulation de particules [19]), l'appel à un ordonnanceur ne peut être ignoré.

Ce type d'applications, dont l'irrégularité ne peut être maîtrisée par l'application des deux stratégies précédentes, est particulièrement étudié dans le cadre de Stratagème à travers les problèmes d'Optimisation Combinatoire par "Branch and bound" présentés dans [2] et par ailleurs dans ce volume.

La résolution par Branch-and-Bound peut être vue comme la construction progressive de l'arbre des sous-problèmes aux solutions potentielles à partir de sa racine, jusqu'à prouver que la dernière solution réalisable obtenue est optimale. Le développement de l'arbre en distribué consiste à construire simultanément des sous-arbres issus de l'exploration des nœuds de l'arbre courant. La taille des sous-arbres étant a priori imprévisible, il peut en résulter un grand déséquilibre sur la charge locale des processeurs. Pour le Branch-and-Bound, l'intégration de la régulation dans l'algorithme consiste donc à le structurer sous forme d'une succession de phases de calcul (se traduisant par une expansion des sous-arbres de recherche déséquilibrant la charge des processeurs) et de phases de régulation permettant d'équilibrer à nouveau la charge.

La méconnaissance a priori de la structure de l'arbre à construire et la place dans cet arbre de la solution optimale rendent l'ordonnancement des nœuds à développer en parallèle difficile. La stratégie à opter pour construire une arborescence [25, 32] de taille minimale pour prouver l'optimalité d'une solution réalisable ne peut être

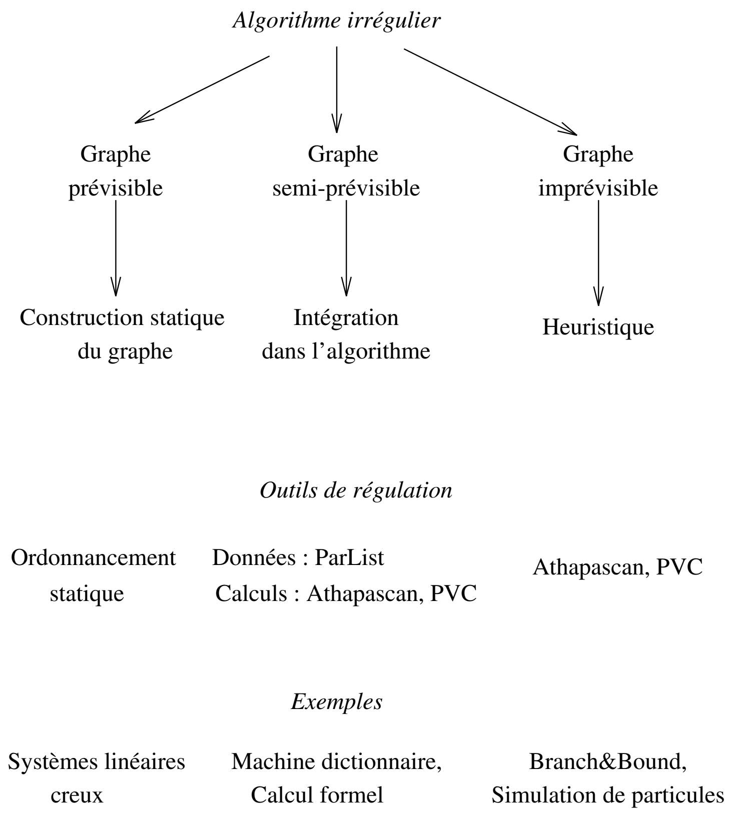

FIG. 1 – *Irrégularité dans les applications de* STRATAGÈME

définie préalablement. Il est alors nécessaire de développer des heuristiques pour décider de l'alternance entre phases de calcul et de régulation.

Une question intéressante est alors la fréquence à laquelle il est nécessaire d'effectuer une phase de régulation. Comme précisé dans le chapitre [33], cette fréquence varie généralement dynamiquement en fonction de l'état de la recherche. Si elle est trop faible, l'irrégularité est faible mais l'exécution est inefficace. Si elle est trop grande, le surcoût de régulation, bien que permettant un meilleur équilibre de la charge, nuit à l'efficacité pratique.

4.3 Irrégularité dans les applications de Stratagème

La figure 1 illustre les différentes techniques mises en œuvre dans les applications étudiées au sein du projet STRATAGÈME pour maîtriser l'irrégularité tout en assurant un équilibre de la charge de façon à garantir une efficacité pratique lors de l'exécution.

5 Conclusion

Le comportement dynamique d'un algorithme est directement corrélé à son irrégularité. L'irrégularité correspond au surcoût lié au calcul d'un ordonnancement permettant d'assurer une efficacité pratique. Différentes techniques permettent de modifier un algorithme irrégulier pour en construire un nouveau ayant une irrégularité plus faible, et donc demandant un surcoût de régulation moindre : adaptation de grain, construction du graphe d'exécution symbolique, structuration en phases.

Ces différentes stratégies sont utilisées dans le cadre de STRATAGÈME sur différents types d'applications.

La recherche d'une efficacité pratique pour une application ayant un fort caractère dynamique est alors liée à trois points clefs au centre des travaux de STRATAGÈME : construction d'un algorithme parallèle théoriquement performant, maîtrise de l'irrégularité et choix d'une stratégie de régulation performante.

Références

[1] Aggarwal (A.), Chandra (A.) et Snir (M.). – Communication complexity of PRAM's. *Theoretical Computer Science*, vol. 71, 1990, pp. 3–28.

[2] Authié (G.) et al. – *Algorithmes parallèles, analyse et conception I.* – Hermès, 1994.

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

[4] Casavant (T.) et Khul (J.). – A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, no14, 1988, pp. 141–154.

[5] Cole (R.) et Vishkin (U.). – Approximate parallel scheduling part I: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, vol. 17, n° 1, 1988.

[6] Cook (S.). – A taxonomy of problems with fast parallel algorithms. *Inf. Control*, vol. 64, 1985, pp. 2–22.

[7] Cosnard (M.). – 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.* pp. 258–267. – IEEE Computer Society Press.

[8] Dehne (F.) et Gastaldo (M.). – A note on the load-balancing problem for coarse grained hypercube dictionary machines. *Parallel Computing*, vol. 16, 1990.

[9] Duboux (T.), Ferreira (A.) et Gastaldo (M.). – MIMD dictionary machine: from theory to practice. *In : CONPAR 92, Lyon, France.*

[10] Facq (L.) et Roman (J.). – SDI et Partitionnement de Systèmes Linéaires Creux. Ce volume.

[11] Ferrari (D.) et Zhou (S.). – 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.

[12] Feschet (F.) et Miguet, S. ans Perroton (L.). – SDI et Equilibrage de Charge en Imagerie. Ce volume.

[13] Finta (L.), Liu (Z.), Milis (I.) et Bampis (E.). – *Scheduling UET-UCT Series-Parallel Graphs on Two Processors.* – Rapport technique n° 2566, INRIA, Mai 1995.

[14] Fonlupt (C.). – *Distribution dynamique de données sur machines SIMD.* – Thèse de PhD, Université de Lille 1, France, 1994.

[15] Fortune (S.) et Wyllie (J.). – Parallelism in random access machines. *In : Proceedings of the 10th ACM Symposium on Theory of Computing*, pp. 114–118.

[16] Fujimoto (A.), Tanaka (T.) et Iwata (K.). – Accelerated ray tracing system. *IEEE Comp. Graph. and App.*, Apr. 1986.

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

[18] Geib (J.) et al. – SDI et Régulation de Charge . Ce volume.

[19] Gerasoulis (A.) et Yang (T.). – A comparison of clustering heuristics for scheduling DAG's on multiprocessors. *J. Par. Distr. Comp.*, Dec. 1992.

[20] Jájá (J.). – *An introduction to parallel Algorithms.* – Addison-Wesley, 1992.

[21] Karp (R.) et Ramachandran (V.). – Parallel algorithms for shared-memory machines. *In : Handbook of Theoretical Computer Science Vol. A*, éd. par van Leuwen (J.), pp. 869–941. – North-Holland, 1990.

[22] Kruskal (C.), Rudolph (L.) et Snir (M.). – A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, no71, 1990, pp. 95–132.

[23] Lai (T.) et Sahni (S.). – Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, vol. 27, n° 6, 1984, pp. 594–602.

[24] Lisper (B.). – Detecting static algorithms by partial evaluation. *In : Proc. ACM Sigplan Symposium on Partial Evaluation and Semi-Based Program Manipulation.*

[25] Mans (B.) et Roucairol (C.). – Performances of parallel branch-and-bound algorithms with best first search. *In : OPOPAC International Workshop on Principles of Parallel Computing*, éd. par Lavallée (I.) et Paker (Y.). pp. 121–139. – Hermès.

[26] Miguet (S.) et Robert (Y.). – Elastic load-balancing for image processing algorithms. *In : 1rst International ACPC Conference, Salzburg, Austria*, éd. par Zima (P. C. H.).

[27] Nemoto (K.) et Omachi (T.). – An adaptative subdivision by sliding boundary surfaces for fast ray tracing. *Graphics Interface*, 1986.

[28] Pierson (J.). – A dynamic parallel implementation of a physically based particles models. *In : Fifth Eurographics Workshop on Animation and Simulation, Oslo Norway*, éd. par Hégron (G.) et Fahlander (O.).

[29] Ranade (A.). – A framework for analyzing locality and portability issues in parallel computing. *In : Parallel Architectures and their Efficient Use*, pp. 185–194.

[30] Rao (V.) et Kumar (V.). – Superlinear speed-up in ordered depth-first search. *In : Proc. 6th Distributed Memory Computing Conference.*

[31] Roch (J.) et Villard (G.). – Algorithmique PRAM II : Algorithmes de travail optimal et probabilistes. Ce volume.

[32] Roucairol (C.). – Parallel processing for difficult combinatorial optimization problems. *In : EURO'XIII*, éd. par of Glasgow (U.). – Tutorial, also available as RR PRiSM, University of Versailles and to appear in EJOR.

[33] Roucairol (C.) et al. – SDI et Optimisation Combinatoire. Ce volume.

[34] Smith (M.) et Renshaw (E.). – Parallel-prefix remapping for efficient data-parallel implementation of unbalanced simulations. *In : Parallel Computing: Trends and Applications, Proceedings of PARCO'93, Grenoble France*. pp. 215–222. – Elsevier Science.

[35] Subramanian (R.) et Scherson (I.). – An analysis of diffusive load-balancing, 1995. Preprint - University of California, Irvine, USA.

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

[37] Valiant (L.). – General purpose parallel architectures. *In : Handbook of Theoretical Computer Science Vol. A*, éd. par van Leuwen (J.), pp. 944–971. – North-Holland, 1990.

[38] Villard (G.). – Parallel general solution of rational linear systems using p-adic expansions. *In : IFIP WG 10.3 Working Conference on Parallel Processing, Pisa Italy*.

[39] Zhou (S.). – A trace-driven simulation study of dynamic load-balancing. *IEEE Trans. on Software Engineering*, vol. 14, n° 9, 1988.