Métriques de performance pour les algorithmes et programmes parallèles

Table des matières

1 Introduction : le temps d'exécution suffit-il?

Lorsqu'on désire analyser des algorithmes, on le fait de façon relativement abstraite, en ignorant de nombreux détails de la machine (temps d'accès à la mémoire, vitesse d'horloge de la machine, etc.).

Une approche abstraite semblable peut être utilisée pour les algorithmes parallèles, mais on doit malgré tout tenir compte de nombreux autres facteurs. Par exemple, il est parfois nécessaire de tenir compte des principales caractéristiques de la machine sous-jacente (modèle architectural) : s'agit-il d'une machine multi-processeurs à mémoire partagée? Dans ce cas, on peut simplifier l'analyse en supposant, comme dans une machine uni-processeur, que le temps d'accès à la mémoire est négligeable (bien qu'on doive tenir compte des interactions possibles entre processeurs par l'intermédiaire de synchronisations). On peut aussi supposer, puisqu'on travaille dans le monde idéal des algorithmes, qu'aucune limite n'est imposée au nombre de processeurs (ressources illimitées, comme on le fait en ignorant, par exemple, qu'un algorithme, une fois traduit dans un programme, ne sera pas nécessairement le seul à être exécuté sur la machine). S'agit-il plutôt d'une machine multi-ordinateurs à mémoire distribuée, donc où le temps d'exécution est très souvent dominé par les coûts de communication entre processeurs? Dans ce cas, ce sont souvent les coûts des communications qui vont dominer le temps d'exécution, ce dont il faut tenir compte lorsqu'on analyse l'algorithme.

Un autre facteur important dont on doit tenir compte : le travail total effectué. Ainsi, l'amélioration du temps d'exécution d'un algorithme parallèle par rapport à un algorithme séquentiel équivalent se fait évidemment en introduisant des processus additionnels qui effectueront les diverses tâches de l'algorithme de façon concurrente et parallèle. Supposons donc qu'on ait algorithme séquentiel dont le temps d'exécution est $O(n)$. Il peut être possible, à l'aide d'un algorithme parallèle, de réduire le temps d'exécution pour obtenir un temps $O(\lg\, n)$. Toutefois, si l'algorithme demande d'utiliser n processeurs pour exécuter les n processus, le coût pour exécuter cet algorithme sera alors $O(n\, \lg\, n)$, donc asymptotiquement supérieur au coût associé à l'algorithme séquentiel. Lorsque cela est possible, il est évidemment préférable d'améliorer le temps d'exécution, mais sans augmenter le coût ou le travail total effectué (deux notions que nous définirons plus en détail plus loin).

Dans les sections qui suivent, nous allons examiner diverses métriques permettant de caractériser les performances d'un algorithme, ou d'un programme, parallèle. La plupart de ces métriques seront, comme on l'a fait pour analyser les algorithmes séquentiels, des métriques *asymptotiques*. Toutefois, nous présenterons aussi certaines métriques qui peuvent être utilisées pour des analyses de programmes concrets, avec un nombre limité (constant) de processeurs, donc des analyses non asymptotique.

2 Temps d'exécution et profondeur

Le temps d'exécution d'un *algorithme* parallèle (indépendant de sa mise en oeuvre par un programme et de son exécution sur une machine donnée) peut être défini comme dans le cas des algorithmes séquentiels, c'est-à-dire, en utilisant la notation asymptotique pour approximer le nombre total d'opérations effectuées (soit en sélectionnant une opération barométrique, soit en utilisant diverses opérations sur les approximations Θ pour estimer le nombre total d'opérations).

Une différence importante, toutefois, est que dans le cas d'une machine parallèle, on doit tenir compte du fait que *plusieurs* opérations de base peuvent s'exécuter en même temps. De plus, même en supposant des ressources infinies (nombre illimité de processeurs), ce ne sont évidemment pas toutes les opérations qui peuvent s'exécuter en même temps, puisqu'il faut évidemment respecter les dépendances de contrôle et de données (les instructions d'une séquence d'instructions doivent s'exécuter les unes après les autres ; un résultat ne peut être

utilisé avant d'être produit ; etc.).

Lorsqu'on voudra analyser le temps d'exécution d'un algorithme écrit dans la notation MPD, qui permet de spécifier facilement un grand nombre de processus, on va supposer que chaque processus pourra, si nécessaire, s'exécuter sur *son propre processeur (processeur virtuel)*. En d'autres mots, en termes d'analyse *d'algorithmes*, on supposera qu'on dispose d'autant de processeurs qu'on en a besoin pour exécuter efficacement l'algorithme. Comme dans le cas d'un algorithme séquentiel, bien qu'une telle analyse ne permette pas nécessairement de prédire de façon exacte le temps d'exécution d'un programme réalisant cet algorithme sur une machine donnée, elle est malgré tout utile pour déterminer comment le temps d'exécution croît en fonction de la taille des données.

Définition 1 *Le* temps d'exécution $T_P(n)$ *d'un algorithme parallèle A exécuté pour un problème de taille n est le temps requis pour exécuter l'algorithme en supposant qu'un nombre illimité de processeurs sont disponibles pour exécuter les diverses opérations, c'est-à-dire en supposant qu'il y a suffisamment de processeurs pour que toutes les opérations qui* peuvent *s'exécuter en parallèle* le soient *effectivement.*

Une telle approche est semblable à celle décrite par G. Blelloch [Ble96], qui parle plutôt de la notion de *profondeur* (*depth*) du calcul associée à un algorithme.

Définition 2 *La* profondeur *du calcul effectué par un algorithme est définie comme la longueur de la plus longue chaîne de dépendances séquentielles présentes dans ce calcul.*

La profondeur représente donc le meilleur temps d'exécution possible en supposant une machine idéale avec un nombre illimité de processeurs. Le terme de *longueur du chemin critique* (*critical path length*) est aussi parfois utilisé au lieu du terme profondeur.

3 Coût, travail et optimalité

3.1 Coût

La notion de *coût* d'un algorithme a pour but de tenir compte à la fois du temps d'exécution mais aussi *du nombre (maximum) de processeurs* utilisés pour obtenir ce temps d'exécution. Ajouter des processeurs est toujours coûteux (achat, installation, entretien, etc.) ; pour un même temps d'exécution, un algorithme qui utilise (asymptotiquement) moins de processeurs qu'un autre est donc préférable.

Définition 3 *Soit un algorithme A utilisé pour résoudre, de façon parallèle, un problème de taille n en temps $T_P(n)$. Soit $p(n)$ le nombre maximum de processeurs effectivement requis par l'algorithme. Le coût de cet algorithme est alors défini comme $C(n) = p(n) \times T_P(n)$*

Ici, on parle de coût d'un algorithme puisque si une machine qui exécute cet algorithme doit, à un certain point de son fonctionnement, avoir jusqu'à $p(n)$ processeurs, alors la machine dans son ensemble avec les $p(n)$ processeurs doit fonctionner durant $T_P(n)$ cycles, et ce même si certains des processeurs ne sont pas utilisés durant tout l'algorithme.

3.2 Travail

Une autre caractéristique intéressante d'un algorithme parallèle est celle du *travail total* effectué par l'algorithme.

Définition 4 *Le* travail, *$W(n)$, dénote le nombre total d'opérations effectuées par l'algorithme parallèle pour un problème de taille n sur une machine à $p(n)$ processeurs.*

Le travail effectué par un algorithme parallèle nous donne donc, d'une certaine façon, le temps requis (plus précisément, un ordre de grandeur) pour faire *simuler* l'exécution de l'algorithme parallèle par un programme à un seul processus, programme qui simulerait l'exécution parallèle de plusieurs opérations en exécutant, une après l'autre, les diverses opérations. On reviendra ultérieurement sur cette notion de *simulation*.

3.3 Coût vs. travail

Telles que définies, les notions de travail et de coût sont très semblables. Si $C(n)$ et $W(n)$ sont définis en utilisant les mêmes opérations barométriques, on aura nécessairement que $T_P(n) \leq W(n) \leq C(n)$. En d'autres mots, le travail est un estimé plus précis, qui tient compte du fait que ce ne sont pas nécessairement tous les processeurs qui travaillent durant toute la durée de l'algorithme.

Une autre différence importante entre le coût et le travail est la suivante : alors que le travail est une métrique *compositionnelle*, ce n'est pas le cas pour le coût. Plus précisément, soit S_1 et S_2 deux segments de code. Le travail effectué pour la composition séquentielle de ces deux segments de code sera simplement la somme du travail de chaque segment. En d'autres mots, on aura l'équivalence suivante :

$$\text{Travail}(S_1; S_2) \quad = \quad \text{Travail}(S_1) + \text{Travail}(S_2)$$

Par contre, cette équivalence ne s'applique pas nécessairement pour le coût, puisque les différentes étapes peuvent ne pas utiliser le même nombre de processeurs et que le coût ne s'intéresse qu'au nombre *maximum* de processeurs requis. Pour le travail, on a donc la relation suivante :

$$\text{Coût}(S_1; S_2) \quad \geq \quad \text{Coût}(S_1) + \text{Coût}(S_2)$$

```
procedure foo( ref int a[*], int b[*], int n )
{
  co [i = 1 to n]
    a[i] = func(a[i], b[i]);
  oc

  co [j = 1 to n/lg(n)]
    proc( a, j, lg(n) );
  oc
}
```

Exemple de code 1: Petit algorithme pour illustrer que le coût n'est pas une métrique compositionnelle

Par exemple, soit la procédure `foo` présentée dans l'exemple de code MPD 1. Supposons que `func` et `proc` soient des fonction et procédure séquentielles et qu'un appel à `func` se fasse en temps $\Theta(1)$ alors qu'un appel à `proc` se fasse en temps $\Theta(\lg\, n)$ (par exemple, une boucle `for` dont le nombre d'itérations est spécifié par le troisième argument).

Le temps total de cette procédure sera le suivant, puisque tous les appels à `func` ainsi que tous ceux à `proc` peuvent se faire en parallèle : $\Theta(1) + \Theta(\lg\, n) = \Theta(\lg\, n)$. À cause du premier `co`, le nombre maximum de processeurs requis par l'algorithme sera n. Le coût total sera donc $n \times \Theta(\lg\, n) = \Theta(n\, \lg\, n)$ ... et non pas $\Theta(n)$ comme on pourrait être tenté de le croire! En d'autres mots, pour le coût, on doit utiliser le nombre *maximum* de processeurs requis par l'algorithme.

Soulignons par contre que si on calculait le travail, on aurait effectivement un résultat qui serait $\Theta(n)$: $n \times \Theta(1) + n/\lg\, n \times \Theta(\lg\, n) = \Theta(n) + \Theta(n) = \Theta(n)$.

3.4 Optimalité

Soit un problème pour lequel on connaît un algorithme séquentiel *optimal* s'exécutant en temps $T_S^*(n)$ (pour un problème de taille n).

Note : Un algorithme séquentiel est optimal au sens où son temps d'exécution ne peut pas être amélioré (asymptotiquement) par aucun autre algorithme séquentiel. En d'autres mots, pour n'importe quel autre algorithme séquentiel s'exécutant en temps $T_S(n)$, on aurait nécessairement que $T_S(n) \in \Omega(T_S^*(n))$.

On peut alors définir deux formes d'optimalité dans le cas d'un algorithme parallèle : une première forme *faible* et une autre plus *forte*.

Définition 5 *Un algorithme parallèle est dit* optimal en travail *si le travail $W(n)$ effectué par l'algorithme est tel que $W(n) \in \Theta(T_S^*(n))$.*

En d'autres mots, le nombre total d'opérations effectuées par l'algorithme parallèle est asymptotiquement le même que celui de l'algorithme séquentiel, et ce indépendamment du temps d'exécution $T_P(n)$ de l'algorithme parallèle.

Une définition semblable peut aussi être introduite en utilisant le *coût* plutôt que le travail dans la définition précédente.

Définition 6 *Un algorithme parallèle est dit* optimal en coût *si le coût $C(n)$ de l'algorithme est tel que $C(n) \in \Theta(T_S^*(n))$.*

Soulignons qu'un algorithme optimal, dans le sens décrit dans cette définition, assure que l'accélération résultante (voir prochaine section) de l'algorithme optimal sur une machine à p processeurs sera $\Theta(p)$, c'est-à-dire résultera en une accélération linéaire.

Une autre définition plus forte d'optimalité de la notion d'optimalité est la suivante :

Définition 7 *Un algorithme parallèle est dit* fortement optimal *(on dit aussi optimal en travail–temps =* work-time optimal*) si on peut prouver que le temps $T_P(n)$ de cet algorithme, lui-même optimal, ne peut pas être amélioré* par aucun autre algorithme parallèle *optimal.*

4 Accélération et efficacité

4.1 Accélération relative vs. absolue

Lorsqu'on utilise un algorithme parallèle, on le fait dans le but d'obtenir un résultat plus rapidement.[1] Une première exigence pour qu'un algorithme parallèle soit intéressant est donc que son temps d'exécution soit asymptotiquement inférieur au temps d'exécution de l'algorithme séquentiel. Ceci peut être formalisé par la condition suivante :

$$T_P(n) \in o(T_S^*(n))$$

La notion *d'accélération* permet de déterminer de façon plus précise de combien un algorithme parallèle est plus rapide qu'un algorithme séquentiel équivalent. On distingue deux façons de définir l'accélération : relative ou absolue.

Notons par $T_P(p, n)$ le temps d'exécution pour un problème de taille n sur une machine à p processeurs — on a donc $T_P(n) = T_P(+\infty, n)$.

[1] Notons qu'on peut aussi vouloir utiliser des algorithmes et programmes parallèles dans le but surtout de résoudre des problèmes *de plus grande taille.* Ceci est particulièrement le cas lorsque qu'on utilise des machines parallèles à mémoire distribuée, dont l'espace mémoire est plus grand, puisque composé de la mémoire des différentes machines.

Rappelons que $g(n) \in o(f(n))$ — qui signifie que $g(n)$ est *petit o*micron de $f(n)$, c'est-à-dire $g(n)$ est dominée asymptotiquement de façon *stricte* par $f(n)$ — implique la relation suivante :

$$g(n) \in o(f(n)) \Rightarrow g(n) \in O(f(n)) - \Omega(f(n))$$

De façon générale, donc, on a la relation suivante :

$$o(f(n)) \supset O(f(n)) - \Omega(f(n))$$

Notons toutefois que pour les fonctions typiques représentant des temps d'exécution, on a habituellement que $o(f(n)) = O(f(n)) - \Omega(f(n))$.

Parenthèse 1: Définition de $o(f(n))$

Définition 8 *L'*accélération relative A_p^r *est défini par le rapport suivant :*

$$A_p^r = \frac{T_P(1,n)}{T_P(p,n)}$$

Pour l'accélération relative, on compare donc l'algorithme s'exécutant sur une machine multi-processeurs avec p processeurs par rapport au même algorithme s'exécutant sur la même machine mais avec un (1) seul processeur.

Cette mesure est souvent utilisée pour caractériser des algorithmes et programmes parallèles ... car elle est plutôt *indulgente*, optimiste. En d'autres mots, cette mesure permet d'obtenir de bonnes accélérations, mais qui ne reflètent pas toujours l'accélération réelle par rapport à ce qu'un bon algorithme séquentiel permet d'obtenir. Pour une comparaison plus juste, il est préférable d'utiliser la notion *d'accélération absolue.*

Définition 9 *Soit* $T_S^*(n)$ *le temps requis pour* le meilleur algorithme séquentiel possible *(pour un problème de taille* n*). L'*accélération absolue A_p^a *est alors définie par le rapport suivant :*

$$A_p^a = \frac{T_S^*(n)}{T_P(p,n)}$$

En d'autres mots, pour l'accélération absolue, on compare l'algorithme s'exécutant sur une machine multi-processeurs avec p processeurs avec le meilleur algorithme séquentiel permettant de résoudre le même problème.

Il est possible de montrer, mais nous ne le ferons pas ici (exercice?), qu'un algorithme parallèle qui est optimal en coût conduit nécessairement à un algorithme possédant une accélération linéaire :

$$C(n) \in \Theta(T_S^*(n)) \Rightarrow A_p^a = p$$

4.1.1 Accélération linéaire vs. superlinéaire

On parle d'accélération (relative ou absolue) *linéaire* lorsque $A_p = p$. En d'autres mots, l'utilisation de p processeurs permet d'obtenir un algorithme p fois plus rapide. En général et, surtout, en pratique (avec des vraies machines et de vrais programmes), de telles accélérations linéaires sont assez rares.

Bizarrement, toutefois, lorsqu'on travaille avec de vraies machines et de vrais programmes (plutôt que simplement avec des algorithmes abstraits), on rencontre parfois des accélérations *superlinéaires*, c'est-à-dire, telles que $A_p > p$. Deux situations expliquent généralement ce genre d'accélérations superlinéaires.

Un premier cas est lorsque l'algorithme est non déterministe (par exemple, une fouille dans un arbre de jeu) : dans ce cas, le fait d'utiliser plusieurs processus peut faire en sorte que l'un des processus "*est chanceux*" et trouve plus rapidement la solution désirée.

Une autre situation, mais qui s'applique clairement à *l'exécution du programme* plutôt qu'à l'algorithme lui-même, est la présence d'*effets de cache*. Ainsi, toutes les machines modernes utilisent une hiérarchie mémoire à plusieurs niveaux :

- Registres ;
- Cache(s) ; [2]
- Mémoire (DRAM).

Les niveaux les plus près du processeur ont un temps d'accès plus rapide, mais sont plus coûteux à mettre en oeuvre, donc sont de plus petite taille.

Il arrive parfois que l'exécution d'un programme sur une machine uni-processeur nécessite un espace mémoire qui conduit à de nombreuses fautes de caches, par ex., à cause de la grande taille des données à traiter. Or, lorsqu'on exécute le même programme mais sur une machine multi-processeurs avec une mémoire cache indépendante pour chaque processeur, le nombre total de fautes de caches est alors réduit de façon importante (parce que l'ensemble des données peut maintenant entrer dans l'ensembles des mémoires cache), conduisant à une accélération supérieure à une accélération linéaire.

4.2 Efficacité

Efficacité de l'utilisation des processeurs

Alors que l'accélération nous indique dans quelle mesure l'utilisation de plusieurs processeurs permet d'obtenir une solution plus rapidement, l'efficacité (en anglais : *efficiency*) nous indique dans quelle mesure les divers processeurs sont utilisés ou non.

Définition 10 *L'*efficacité *d'un algorithme est le rapport entre le temps d'exécution séquentiel et le coût d'exécution sur une machine à p processeurs. En d'autres mots, l'efficacité est obtenue en divisant l'accélération obtenue avec p processeurs par le nombre de processeurs p :*

$$E_p = \frac{T_S^*(n)}{C(n)} = \frac{A_p^a(n)}{p} = \frac{T_S^*(n)}{p \times T_P(p,n)}$$

Alors que pour p processeurs, l'accélération idéale est de p, l'efficacité idéale est de 1, c'est-à-dire, le cas idéal est lorsque les p processeurs sont utilisés à 100 % (très rare).

Efficacité en travail

Une autre notion d'efficacité est celle *d'efficacité en travail* (en anglais : *work efficiency*) par rapport à un algorithme séquentiel optimal. Pour qu'un algorithme parallèle soit considéré efficace en travail, il est acceptable que cet algorithme fasse "*plus*" de travail qu'un bon algorithme séquentiel. Toutefois, il ne faut pas non plus que le travail effectué soit énormément plus grand. Plusieurs auteurs considèrent qu'il est acceptable que le travail soit plus grand *par un facteur polylogarithmique.*[3]

Définition 11 *On dit d'un algorithme parallèle (pour un problème de taille n) est* efficace en travail *s'il existe un entier $k \geq 1$ tel que la condition suivante est satisfaite :*

$$W(n) \in T_S^*(n) \times O(\lg^k(T_S^*(n)))$$

[2] En fait, la plupart des machines modernes ont maintenant deux ou plusieurs niveaux de cache, par exemple : (i) mémoire (SRAM) de premier niveau, sur la même puce que le processeur ; (ii) mémoire de deuxième niveau, souvent sur une autre puce, mais avec un temps d'accès inférieur au temps d'accès à la mémoire.

[3] Une fonction est polylogarithmique si elle est définie par une puissance de fonction logarithmique, c'est-à-dire définie par une fonction $\lg^k n = (\lg n)^k$, pour $k \geq 0$. Notons que cette notation n'est pas la même que celle utilisée dans le chapitres sur les algorithmes voraces pour l'analyse des forêts d'arbres disjoints.

Évidemment, on acceptera que le travail effectué soit plus grand si l'accélération résultante est intéressante, donc si l'algorithme parallèle est nettement plus rapide que l'algorithme séquentiel.

D'autres auteurs définissent plutôt les algorithmes efficaces (*efficient*) comme étant ceux qui satisfont les deux conditions suivantes :

1. L'algorithme requiert un nombre *polynômial* de processeurs.
2. Le temps d'exécution de l'algorithme est *polylogarithmique*

En fait, un peu comme pour la classe des problèmes NP-complets qui identifie une classe de problèmes de difficultés équivalentes pour le modèle séquentiel d'exécution, on peut définir une certaine classe NC de problèmes pour le modèle parallèle d'exécution (plus spécifiquement le modèle PRAM, que nous verrons au prochain chapitre) [KKT01] :

Définition (La classe NC) *La classe NC est formée des problèmes qui peuvent être solutionnés à l'aide du modèle PRAM en temps polylogarithmique $O(log^k\ n)$ pour un $k \geq 0$ en utilisant seulement $n^{O(1)}$ processeurs (i.e., un nombre polynomial de processeurs), où n représente la taille de l'instance du problème.*

Cette classe de problèmes est souvent considérée comme la classe des problèmes qui sont *bien parallélisables*. Notons toutefois que certains auteurs critiquent cette façon de caractériser les algorithmes parallèles efficaces, dans la mesure où la classe NC contient des problèmes qui sont connus pour être difficilement parallélisables de façon efficace, alors qu'elle exclut des problèmes pour lesquels des algorithmes parallèles raisonnables existent. Une autre critique est que cette définition exclut les aspects liés au coût, ce qui fait que certains algorithmes de cette classe peuvent donc être nettement plus coûteux qu'un algorithme séquentiel équivalent.

Algorithme fortement parallélisable

Finalement, une autre définition de l'efficacité en travail est celle basée sur la notion d'algorithme *fortement parallélisable*, notion plus forte que celle d'algorithme bien parallélisable [GUD96] :

Définition 12 *On dit d'un algorithme parallèle (pour un problème de taille n) qu'il est* fortement parallélisable *si les conditions suivantes sont satisfaites :*

- *Le temps d'exécution avec p processeurs est polylogarithmique, c'est-à-dire de la forme $O(\lg^{O(1)}\ n)$.*
- *Le nombre de processeurs utilisés, p, n'excède pas la complexité séquentielle $T_S^*(n)$.*
- *L'accélération résultante est d'au moins $\Omega(p^\epsilon)$ avec $\epsilon > 0$.*

De façon plus détaillée, cette définition implique les conditions suivantes [GUD96, p. 177]:

> Le temps d'exécution doit être logarithmique par rapport à la taille du problème. Le nombre de processeurs est limité à la complexité séquentielle du problème. Utiliser plus de processeurs paraît en effet exagéré. Finalement, l'accélération doit être un polynôme dans le nombre de processeurs. Ceci garantit que l'accélération est importante pour des instances du problème de toutes tailles et pour tout nombre de processeurs.

5 Dimensionnement (*scalability*)

On dit d'un algorithme qu'il est *dimensionnable* (*scalable*) lorsque le niveau de parallélisme augmente au moins de façon linéaire avec la taille du problème. On dit d'une architecture qu'elle est dimensionnable si la machine continue à fournir les mêmes performances par processeur lorsque l'on accroît le nombre de processeurs.

L'importance d'avoir un algorithme et une machine dimensionnables provient du fait que cela permet de résoudre des problèmes de plus grande taille sans augmenter le temps d'exécution simplement en augmentant le nombre de processeurs.

6 Coûts des communications

Un algorithme est dit *distribué* lorsqu'il est conçu pour être exécuté sur une architecture composée d'un certain nombre de processeurs — reliés par un réseau, où chaque processeur exécute un ou plusieurs processus —, et que les processus coopèrent strictement en s'échangeant des messages.

Dans un tel algorithme, il arrive fréquemment que le temps d'exécution soit largement dominé par le temps requis pour effectuer les communications entre processeurs — tel que décrit à la Section 7, le coût d'une communication peut être de plusieurs ordres de grandeur supérieur au coût d'exécution d'une instruction normale. Dans ce cas, il peut alors être suffisant d'estimer la complexité du *nombre de communications* requis par l'algorithme. En d'autres mots, les communications deviennent les opérations barométriques.

7 Sources des sur-coûts d'exécution parallèle

Idéalement, on aimerait, pour une machine parallèle à n processeurs, obtenir une accélération qui soit $\approx n$ et une efficacité qui soit ≈ 1. En pratique, de telles performances sont très difficiles à atteindre. Plusieurs facteurs entrent en jeu et ont pour effet de réduire les performances :

- Création de processus et changements de contexte : créer un "processus" est une opération relativement coûteuse.

  Les langages supportant les *threads* (soit de façon directe, par ex., Java, soit par l'intermédiaire de bibliothèques, par ex., les *threads* Posix en C) permettent d'utiliser un plus grand nombre de tâches/*threads* à des coûts inférieurs, puisqu'un *thread* est considéré comme une forme *légère* de processus (*lightweight process*).

  Qu'est-ce que le "poids" d'un processus? Un processus/*thread*, au sens général du terme, est simplement une tâche indépendante (donc pas nécessairement un `process` au sens Unix du terme). Un processus/*thread* est toujours associé à *contexte*, qui définit l'environnement d'exécution de cette tâche. Minimalement, le contexte d'un *thread* contient les éléments suivants :

  - Registres (y compris le pointeur d'instruction) ;
  - Variables locales (contenues dans le *bloc d'activation* sur la pile).

  Par contre, un processus, au sens Unix du terme, contient en gros les éléments suivants :

  - Registres ;
  - Variables locales ;
  - Tas (*heap*) ;
  - Descripteurs de fichiers et *pipes* ouverts/actifs ;
  - Gestionnaires d'interruption ;

– Code du programme.

Dans le cas des *threads*, certains des éléments qui sont présents dans le contexte d'un processus et qui sont requis pour exécuter les divers *threads* — par ex., le code du programme — sont plutôt *partagés* entre les *threads*, des *threads* étant toujours définis dans le contexte d'un processus.

Or, la création et l'amorce d'un nouveau *thread* ou processus demande toujours d'allouer et d'initialiser un contexte approprié. De plus, un *changement de contexte* survient lorsqu'on doit suspendre l'exécution d'un *thread* (parce qu'il a terminé son exécution, parce qu'il devient bloqué, ou parce que la tranche de temps (*time slice*) qui lui était allouée est écoulée) et sélectionner puis réactiver un nouveau *thread*.

Effectuer ces opérations introduit donc des sur-coûts qui peuvent devenir important si, par exemple, le nombre de *threads* est nettement supérieur au nombre de processeurs, conduisant ainsi à de nombreux changement de contexte.

- Synchronisation et communication entre processus : un programme concurrent, par définition, est formé de plusieurs processus qui interagissent et coopèrent pour résoudre un problème global. Cette coopération, et les interactions qui en résultent, entraînent l'utilisation de divers mécanismes de synchronisation (par exemple, sémaphores pour le modèle de programmation par variables partagées) ou de communication (par ex., envois et réception de messages sur les canaux de communication dans le modèle par échanges de messages). L'utilisation de ces mécanismes entraîne alors des coûts supplémentaires par rapport à un programme séquentiel équivalent (en termes des opérations additionnelles à exécuter, ainsi qu'en termes des délais et changements de contexte qu'ils peuvent impliquer).

- Communications inter-processeurs : les coûts de communications peuvent devenir particulièrement marqués lorsque l'architecture sur laquelle s'exécute le programme est de type "multi-ordinateurs", c'est-à-dire, lorsque les communications et échanges entre les processus/processeurs se font par l'intermédiaire d'un réseau. Le temps nécessaire pour effectuer une communication sur un réseau est supérieur, en gros, au temps d'exécution d'une instruction normale par un facteur de 2–4 *ordres de grandeur* — donc pas 2–4 fois plus long, mais bien 10^2–10^4 fois plus. Les communications introduisent donc des délais dans le travail effectué par le programme. De plus, comme il n'est pas raisonnable d'attendre durant plusieurs milliers de cycles sans rien faire, une communication génère habituellement aussi un changement de contexte, donc des sur-coûts additionnels.

- Répartition de la charge de travail (*load balancing*) : la répartition du travail (des tâches, des *threads*) entre les divers processeurs, tant sur une machine multi-processeurs que sur une machine multi-ordinateurs, entraîne l'exécution de travail supplémentaire (généralement effectué par un *load balancer*, qui fait partie du système d'exécution (*run-time system*) de la machine).

  Si les tâches sont mal réparties, il est alors possible qu'un processeur soit surchargé de travail, alors qu'un autre ait peu de travail à effectuer, donc soit mal utilisé. L'effet global d'un déséquilibre de la charge est alors d'augmenter le temps d'exécution et de réduire l'efficacité globale. Toutefois, assurer une répartition véritablement équilibrée de la charge de travail, particulièrement sur une machine multi-ordinateurs, peut entraîner des sur-coûts non négligeables (principalement au niveau des communications requises pour *monitorer* la charge et répartir le travail entre les processeurs).

- Calculs supplémentaires : l'accélération *absolue* se mesure relativement au meilleur algorithme séquentiel permettant de résoudre le problème. Or, il est possible que cet algorithme séquentiel ne puisse pas être parallélisé (intrinsèquement séquentiel, à cause

des dépendances de contrôle et de données qui le définissent). Dans ce cas, l'algorithme parallèle pourra être basé sur une version séquentielle moins intéressante, mais plus facilement parallélisable.

8 Simulation, réduction du nombre de processeurs et théorème de Brent

Lorsqu'on conçoit un algorithme parallèle, il est souvent préférable, dans un premier temps, de supposer que l'on dispose d'un nombre *arbitraire* de processeurs. On peut alors parler, à cette étape, de l'utilisation de *processeurs virtuels*. Dans ce cas, on formule donc l'algorithme en fonction d'un nombre arbitraire de tâches (opérations) pouvant s'exécuter, à chacune des étapes, de façon indépendante, en supposant qu'il y a autant de processeurs (*virtuels*) disponibles que de tâches à exécuter.

Évidemment, les machines réelles ne disposant que d'un nombre limité de processeurs, l'algorithme initial doit alors être transformé pour tenir compte des limites réelles de la machine. Dans le cas d'une machine concrète, un travail important de *programmation* doit alors être effectué, travail demandant de recourir à des stratégies et techniques de *programmation parallèle* bien au-delà de ce que l'on peut aborder dans le présent cours.

Par contre, il est intéressant de réaliser, en restant toujours au niveau des algorithmes, qu'un algorithme parallèle A nécessitant p processeurs peut généralement être *simulé* par un algorithme parallèle A' utilisant un nombre réduit q de processeurs ($q < p$). Intuitivement, il s'agit d'utiliser une approche de simulation par A' des étapes parallèles de A. Pour simplifier, supposons donc que p soit un multiple de q, c'est-à-dire $p = kq$ pour un certain $k > 1$. Pour effectuer la simulation, il suffit d'assigner à chacun des q processeurs un sous-ensemble de k ($= \frac{p}{q}$) processeurs de l'algorithme initial A. L'exécution des opérations effectuées, à une étape donnée, par un groupe de k processeurs de A pourra alors être simulée par l'exécution de $\Theta(k)$ opérations par A' — le facteur multiplicatif représenté par $\Theta(k)$ provient évidemment des sur-coûts associés à la simulation : gestion de la simulation (par ex., stratégie *round-robin*) des processeurs, coûts des changements de contexte, etc.

Un théorème dû à Brent (1974) — formulé dans le contexte des machines PRAM, que nous examinerons dans un prochain chapitre — formalise cette propriété et peut être exprimé comme suit :

Théorème 1 (Théorème de Brent) *Soit un algorithme parallèle A fonctionnant en temps $t(n)$ avec $p(n)$ processeurs.*

Pour $q(n) < p(n)$, il existe un algorithme parallèle modifié qui résout le même problème avec $q(n)$ processeurs dans le temps suivant :

$$\Theta(t(n) \times \frac{p(n)}{q(n)})$$

A Application des métriques à des algorithmes parallèles exprimés en MPD

Caractéristiques des différentes constructions

- `x = ops. élémentaires`

T (temps)	P (nb. max. procs)	W (travail)	C (coût)
$\Theta(1)$	1	$\Theta(1)$	$\Theta(1)$

- `f(a1, ..., an)` *(Appel de procédure)*
  ou
  `x = f(a1, ..., an)` *(Appel de fonction)*

T (temps)	P (nb. max. procs)	W (travail)	C (coût)
$T(\texttt{f(a1,...,an)})$	$P(\texttt{f(a1,...,an)})$	$W(\texttt{f(a1,...,an)})$	$C(\texttt{f(a1,...,an)})$

- ```
  if ( c ) {
    B1
  } else {
    B2
  }
  ```

T (temps)	P (nb. max. procs)	W (travail)	C (coût)
$max\{T(\texttt{B1}), T(\texttt{B2})\}$	$max\{P(\texttt{B1}), P(\texttt{B2})\}$	$max\{W(\texttt{B1}), W(\texttt{B2})\}$	$max\{C(\texttt{B1}), C(\texttt{B2})\}$

- ```
  for [i = 1 to n] {
    B(i)
  }
  ```

T (temps)	P (nb. max. procs)	W (travail)	C (coût)
$\sum_{i=1}^{n} T(B(i))$	$max_{i=1}^{n} P(B(i))$	$\sum_{i=1}^{n} W(B(i))$	$\sum_{i=1}^{n} C(B(i))$

- ```
  while ( c ) {
    B(i)
  }
  ```

T (temps)	P (nb. max. procs)	W (travail)	C (coût)
$\sum_{i=1}^{n} T(B(i))$	$max_{i=1}^{n} P(B(i))$	$\sum_{i=1}^{n} W(B(i))$	$\sum_{i=1}^{n} C(B(i))$

Note : On suppose que n itérations sont requises.

- ```
  co
     B(1)
  // B(2)
     ...
  // B(n)
  oc
  ```

T (temps)	P (nb. max. procs)	W (travail)	C (coût)
$max_{i=1}^{n} T(B(i))$	$\sum_{i=1}^{n} P(B(i))$	$\sum_{i=1}^{n} W(B(i))$	$\sum_{i=1}^{n} P(B(i)) \times \max_{i=1}^{n} T(B(i))$

- ```
  co [i = 1 to n]
    B(i)
  oc
  ```

T (temps)	P (nb. max. procs)	W (travail)	C (coût)
$max_{i=1}^{n} T(B(i))$	$\sum_{i=1}^{n} P(B(i))$	$\sum_{i=1}^{n} W(B(i))$	$\sum_{i=1}^{n} P(B(i)) \times \max_{i=1}^{n} T(B(i))$
  ```

Analyse des algorithmes récursifs parallèles

Le temps requis par un algorithme récursif parallèle doit être caractérisé par des équations de récurrence, comme c'est le cas pour un algorithme récursif séquentiel. Toutefois, les équations de récurrence seront différentes car elles tiendront compte du fait que certains appels récursifs pourront s'exécuter en parallèle.

Par exemple, soit l'algorithme suivant, qui multiplie un vecteur par un scalaire pour produire un vecteur en résultat :

```
procedure multScalaire( int a[*], int i, int j, int x, res int r[*] )
{
  if (i == j) {
    r[i] = x * a[i];
  } else {
    int mid = (i+j)/2;
    co
        multScalaire(a, i,     mid, r);
    //  multScalaire(a, mid+1, j,   r);
    oc
  }
}
```

Les équations de récurrence décrivant la complexité asymptotique du temps d'exécution seront les suivantes, puisque les deux appels récursifs à `multScalaire` peuvent se faire de façon concurrente :

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(n/2) + \Theta(1) \text{ si } n > 1 \end{aligned}$$

Dans ce cas, on peut conclure que le temps d'exécution sera $\Theta(\lg\, n)$.

Pour déterminer le nombre de processeurs requis pour l'exécution d'un algorithme récursif, on peut considérer que lorsqu'une instance de la procédure effectue un ou plusieurs appels récursifs, le processeur associé à cette procédure devient alors *libre*, c'est-à-dire *disponible* pour être utilisé pour effectuer d'autres opérations, donc traiter d'autres appels récursifs. Ceci s'explique par le comportement (la sémantique) de l'instruction `co` : tant que les procédures appelées (les processus créés dans le `co`) n'ont pas terminé leur exécution, la procédure appelante est *suspendue* (puisque l'instruction `co` est bloquée jusqu'à ce que les processus créés dans le `co` se terminent) et donc n'a pas besoin qu'un processeur lui soit associé.

Le nombre maximum de processeurs requis par l'algorithme est donc déterminé par *le nombre maximum d'instances de la procédure récursive qui peuvent être actives en même temps.* De façon intuitive, on peut déterminer cette caractéristique en analysant la structure de l'arbre des appels récursifs : le nombre maximum d'instances actives est déterminé *par le nombre maximum de feuilles* de cet arbre, puisque pour tout arbre k-aire ($k \geq 2$, k indiquant le nombre d'enfants de chaque noeud interne), le nombre de feuilles de l'arbre complet est nécessairement supérieur au nombre de feuilles d'un arbre tronqué :

- Un noeud interne (donc qui n'est pas une feuille) représente une instance de la procédure suspendue, en attente que les procédures qu'elle a appelées se terminent.
- L'arbre complet, avec toutes les feuilles présentes, contient nécessairement plus de feuilles qu'un arbre partiellement développé, puisque chaque noeud interne est associé à au moins k feuilles.

Dans notre exemple, le nombre maximum de processeurs requis sera donc de n — le nombre de feuilles qui correspond au cas où toutes les instances des cas de base sont actives en même temps, pour chacun des n éléments de `a`.

Références

[Ble96] G.E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[GUD96] M. Gengler, S. Ubéda, and F. Desprez. *Initiation au parallélisme—Concepts, architectures et algorithmes.* Masson, 1996. [QA76.58G45].

[JaJ92] J. JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992. [QA76.58J34].

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing—Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, 1994. [QA76.58I58].

[KGGK03] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing (Second Edition).* Addison-Wesley, 2003.

[KKT01] J. Keller, C. Kessler, and J. Traff. *Practical PRAM Programming.* John Wiley & Sons, Inc., 2001.

[MB00] R. Miller and L. Boxer. *Algorithms Sequential & Parallel.* Prentice-Hall, 2000. [QA76.9A43M55].