

Métriques de performance pour les algorithmes et programmes parallèles

11–18 nov. 2002

Cette section est basée tout d’abord sur la référence suivante (manuel suggéré mais non obligatoire) :

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

Les références suivantes ont aussi été consultées :

- V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing — Design and Analysis of Algorithms*. Benjamin/Cummings, 1994. [QA76.58I58].
- J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. [QA76.58J34].
- G.E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

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 d’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, où 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?

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 un 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 $O(n)$ processeurs pour exécuter les n processus, il faut alors réaliser que le *travail total* effectué par l'algorithme sera $O(n \lg n)$, donc asymptotiquement supérieur au travail effectué par l'algorithme séquentiel. Lorsque cela est possible, il est évidemment préférable d'améliorer le temps d'exécution, mais sans augmenter le travail total effectué.

Dans les sections qui suivent, nous allons examiner diverses métriques permettant de caractériser les performances d'un algorithme, ou d'un programme, parallèle.

2 Temps d'exécution vs. 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 O 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 s'exécuter sur *son propre processeur (virtuel)*. En d'autres mots, en termes d'analyse *d'algorithmes*, on supposera qu'un nombre illimité de processeurs est disponible. 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.

Une telle approche est semblable à celle décrite par G. Blelloch (Blelloch, 1996), qui parle de la notion de *profondeur (depth)* associée à un algorithme.

Définition 1 *La profondeur du calcul effectué par un algorithme est définie comme 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 de processeurs* permettant d'obtenir ce temps d'exécution. Ajouter des processeurs est toujours coûteux (achat, installation, entretien, etc.) ; pour un même temps d'exécution, un algorithme qui nécessite l'utilisation de moins de processeurs qu'un autre est donc toujours préférable.

Définition 2 Notons par $T_p(n)$ le temps requis par un algorithme sur une machine parallèle à n processeurs. Le coût de cet algorithme est alors défini comme $C(n) = n \times T_p(n)$, qui représente le nombre total de cycles durant lesquels l'algorithme s'exécute sur l'ensemble de la machine.

3.2 Travail

Une définition semblable à celle de coût est celle du *travail total* effectué par un algorithme.

Définition 3 Le travail, $W(n)$, dénote le nombre total d'opérations effectuées par l'algorithme parallèle sur une machine à n processeurs.

3.3 Coût vs. travail

Telles que définies, les notions de travail et de coût sont donc très similaires. Si $C(n)$ et $W(n)$ sont définis en utilisant les mêmes opérations barométriques, on aura nécessairement que $W(n) \leq C(n)$. En d'autres mots, le travail est un estimé un peu 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.

3.4 Optimalité

Soit un problème pour lequel on connaît un algorithme séquentiel *optimal* s'exécutant en temps T_s^* (l'algorithme est optimal au sens où cette borne ne peut pas être améliorée par aucun autre algorithme séquentiel).

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

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.

Soulignons qu'un algorithme optimal, dans le sens décrit dans cette définition, assure que l'accélération résultant (voir prochaine section) de l'algorithme optimal sur une machine à p processeurs sera $\Theta(p)$.

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

Définition 5 *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

L'accélération permet de déterminer de combien un algorithme parallèle est plus rapide qu'un algorithme séquentiel équivalent. On distingue deux façons de définir l'accélération : relative vs. absolue.

Définition 6 *Soit $T_p(n)$ le temps requis par un algorithme sur une machine parallèle à n processeurs. L'accélération relative S_n^r est alors défini comme $T_p(1)/T_p(n)$. En d'autres mots, on compare l'algorithme s'exécutant sur une machine multi-processeurs avec n processeurs par rapport au même algorithme s'exécutant sur la même machine mais avec un seul processeur.*

Définition 7 *Soit $T_p(n)$ le temps requis par un algorithme sur une machine parallèle à n processeurs. Soit T_s^* le temps requis pour le meilleur algorithme séquentiel possible. L'accélération absolue S_n^a est alors définie comme $T_s^*/T_p(n)$.*

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

4.1.1 Accélération linéaire vs. superlinéaire

On parle d'accélération (relative ou absolue) *linéaire* lorsque $S_n = 1$. En d'autres mots, l'utilisation de n processeurs permet d'obtenir un algorithme n fois plus rapide. En général, de telles accélérations linéaires sont assez rares.

Bizarrement toutefois, on rencontre parfois des accélérations *superlinéaires*, c'est-à-dire, telles que $S_n > 1$. 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 jeux) : 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 plutôt à *l'exécution du programme* plutôt qu'à l'algorithme lui-même, est la présence d'*effets de cache*. Toutes les machines modernes utilisent une hiérarchie mémoire à plusieurs niveaux :

- Registres ;
- Cache(s) ;¹
- 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'ensemble des mémoires cache), conduisant à une accélération supérieure à une accélération linéaire.

4.2 Efficacité

Alors que l'accélération nous indique dans quelle mesure l'utilisation de plusieurs processeurs permet d'obtenir une solution plus rapidement, l'efficacité nous indique dans quelle mesure les divers processeurs sont utilisés.

Définition 8 *L'efficacité d'un algorithme est le rapport entre le temps d'exécution séquentiel et le coût d'exécution sur une machine à n processeurs. En d'autres mots, l'efficacité est obtenue en divisant l'accélération obtenue avec n processeurs par le nombre de processeurs :*

$$E_n = \frac{T_s}{C(n)} = \frac{S_n^a}{n}$$

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

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.

¹En 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.

6 Coûts des communications

Un algorithme est *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 surcoûts 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., Pthreads de 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. Minimale, 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*, les éléments additionnels présents dans le contexte d'un processus sont plutôt *partagés* entre les *threads* (des *threads* étant 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 introduisent donc des surcoûts qui peuvent devenir important si, par exemple, le nombre de *threads* est nettement supérieur au nombre de processeurs, conduisant ainsi à de nombreux changements 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 surcoûts 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 surcoûts non négligeables (principalement au niveau des communications requises pour monitorer le système et la charge, puis 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.