Table des matières

Chapitre 6

Métriques de performance pour algorithmes et programmes parallèles

6.1 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 ce qui suit :

1. Métriques *asymptotiques*, pour des algorithmes
2. Métriques *réelles*, pour des programmes

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 asymptotiques.

6.2 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.

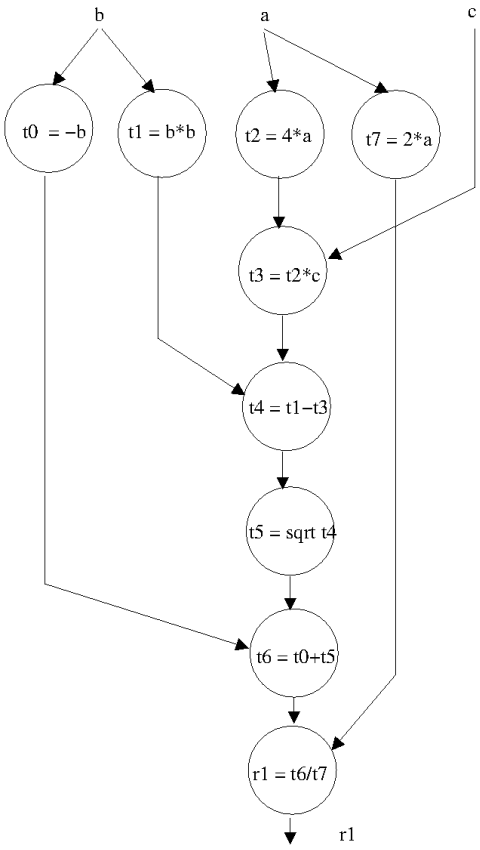
Définition 1 *Le* temps d'exécution $T_P(n)$ *d'un* **algorithme parallèle** *A exécuté pour un problème de taille n est le temps requis pour exécuter l'algorithme* **en supposant qu'un nombre illimité de processeurs sont disponibles** *pour exécuter les diverses opérations, c'est-à-dire en supposant 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 *d'un calcul effectué par un algorithme est définie comme* **la longueur de la plus longue chaîne de dépendances** *séquentielles présentes dans ce calcul.*

La profondeur (le temps) 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.



Nb. d'UE	**Temps min.**
1	9
2	6
3	6
4	6
...	...

Figure 6.1: Graphe de dépendances pour calcul d'une racine d'un polynome de deuxième degré pour illustrer la notion du meilleur temps parallèle comme longueur du plus long chemin.

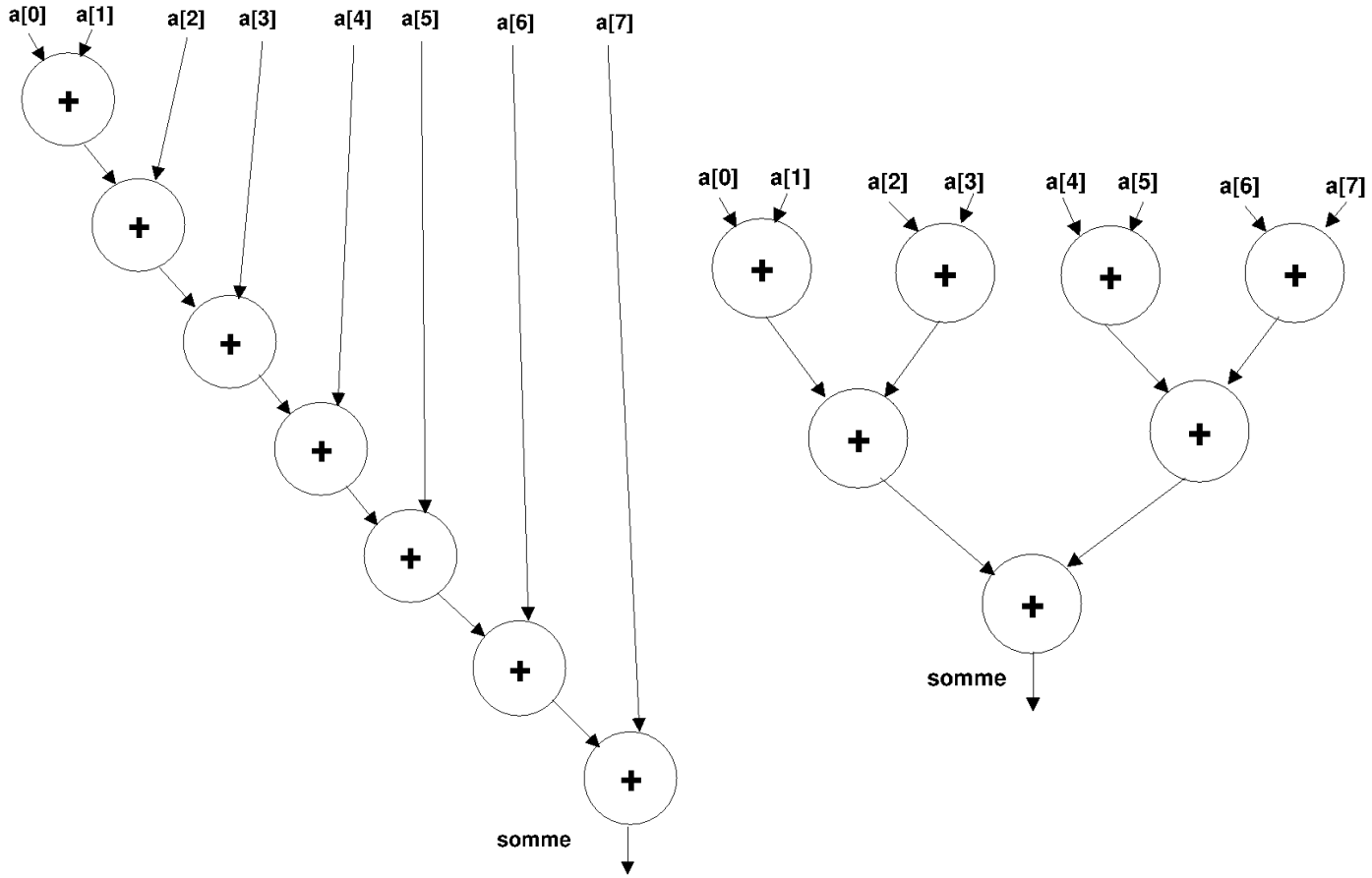


Figure 6.2: Graphe de dépendances pour calcul de la somme d'un tableau de huit éléments, pour illustrer la notion du meilleur temps parallèle comme longueur du plus long chemin.

6.3 Coût, travail et optimalité

6.3.1 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 decet algorithme A 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.

6.3.2 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*.

6.3.3 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 du **nombre total exact d'opérations** exécutées qui tient compte du fait que ce ne sont pas nécessairement tous les processeurs qui travaillent durant toute la durée de l'algorithme.

Relation entre temps, travail et coût

- Séquentiel :

$$T_S(n) = W(n) = C(n)$$

- Parallèle :

$$T_P(n) \leq W(n) \leq C(n)$$

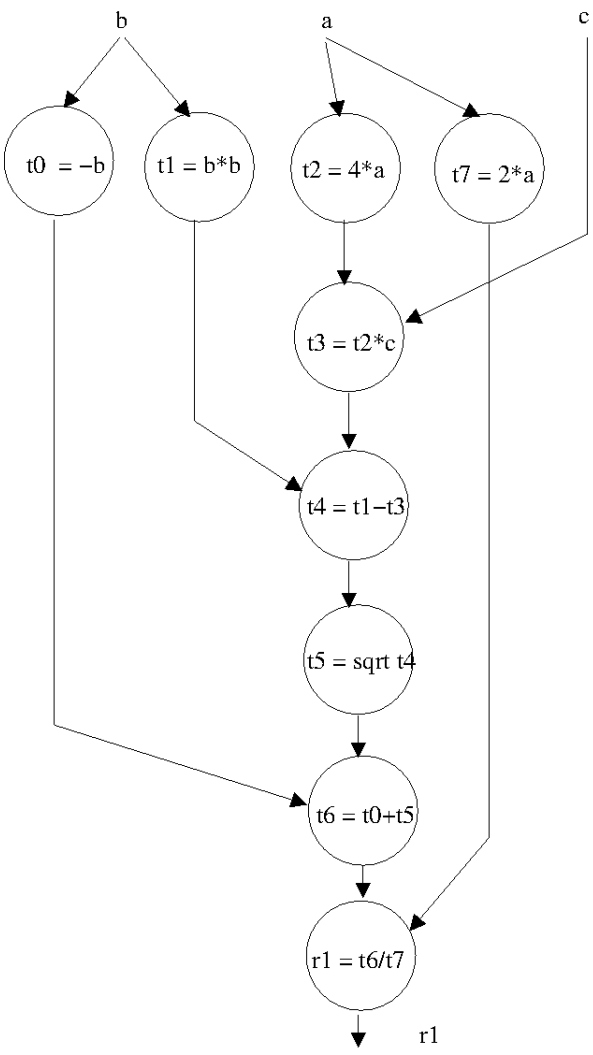


Figure 6.3: Le travail.

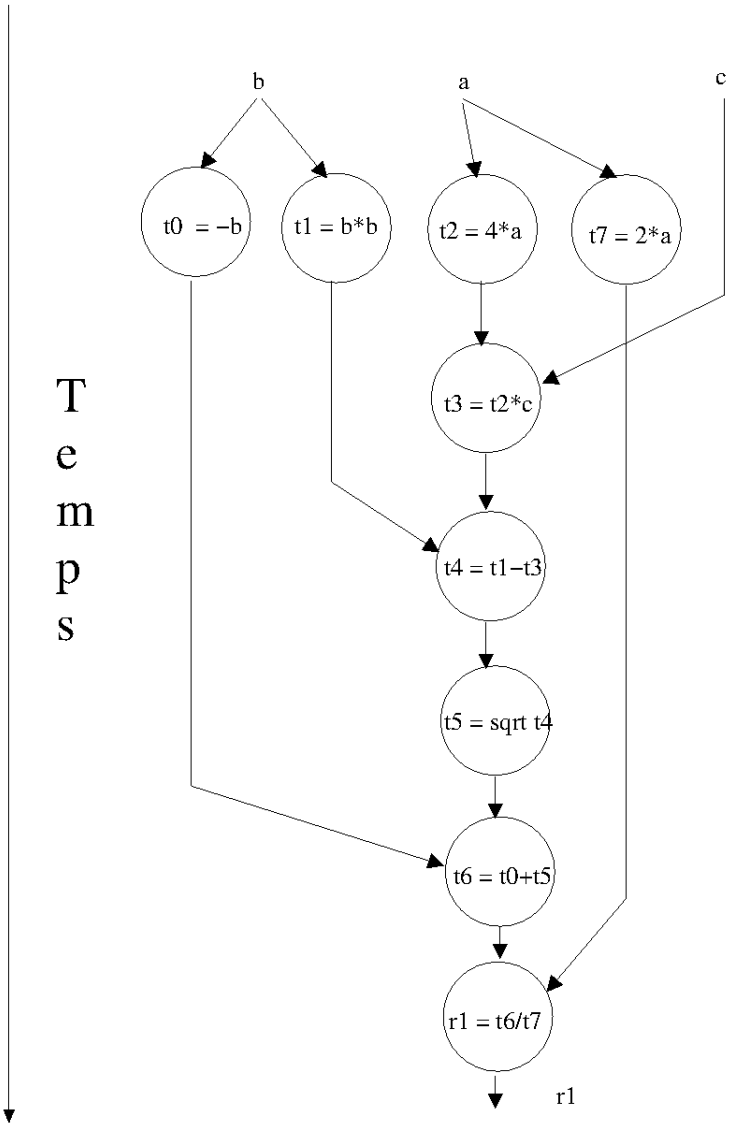


Figure 6.4: Le temps.

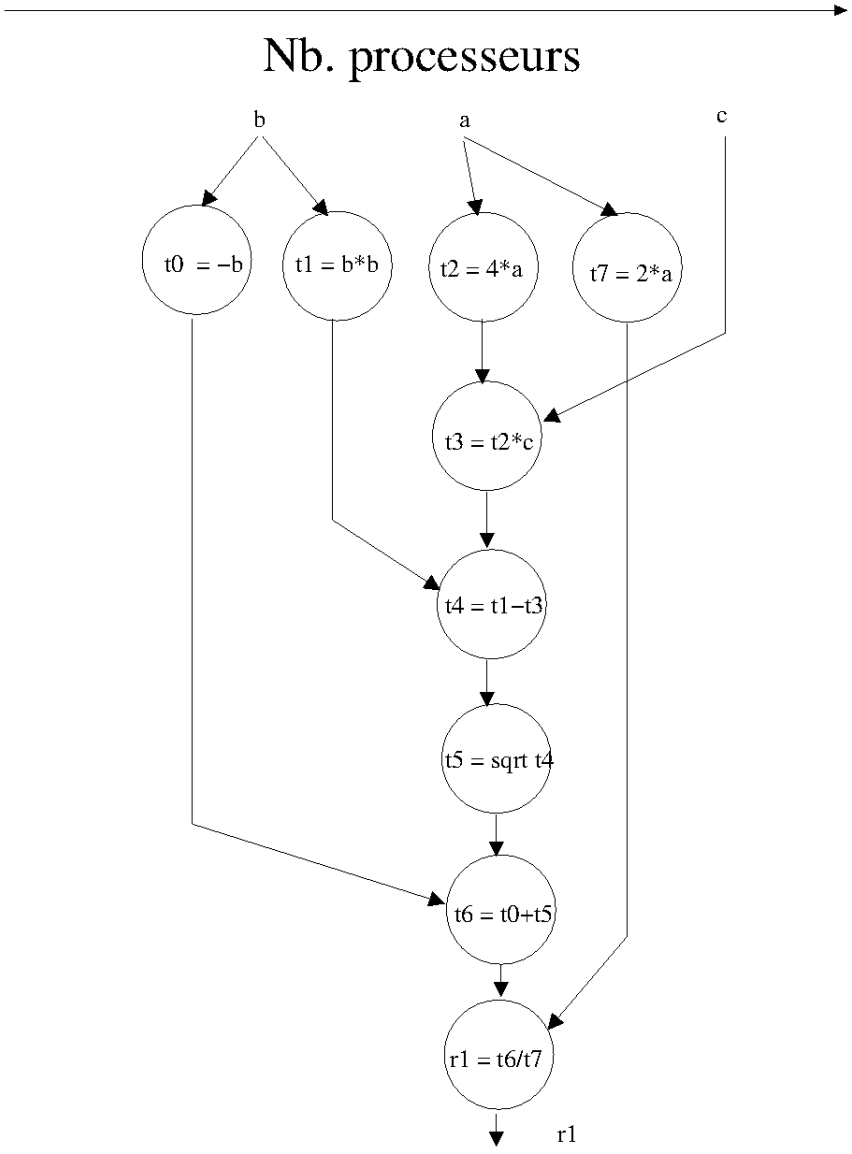


Figure 6.5: Le nombre de processeurs.

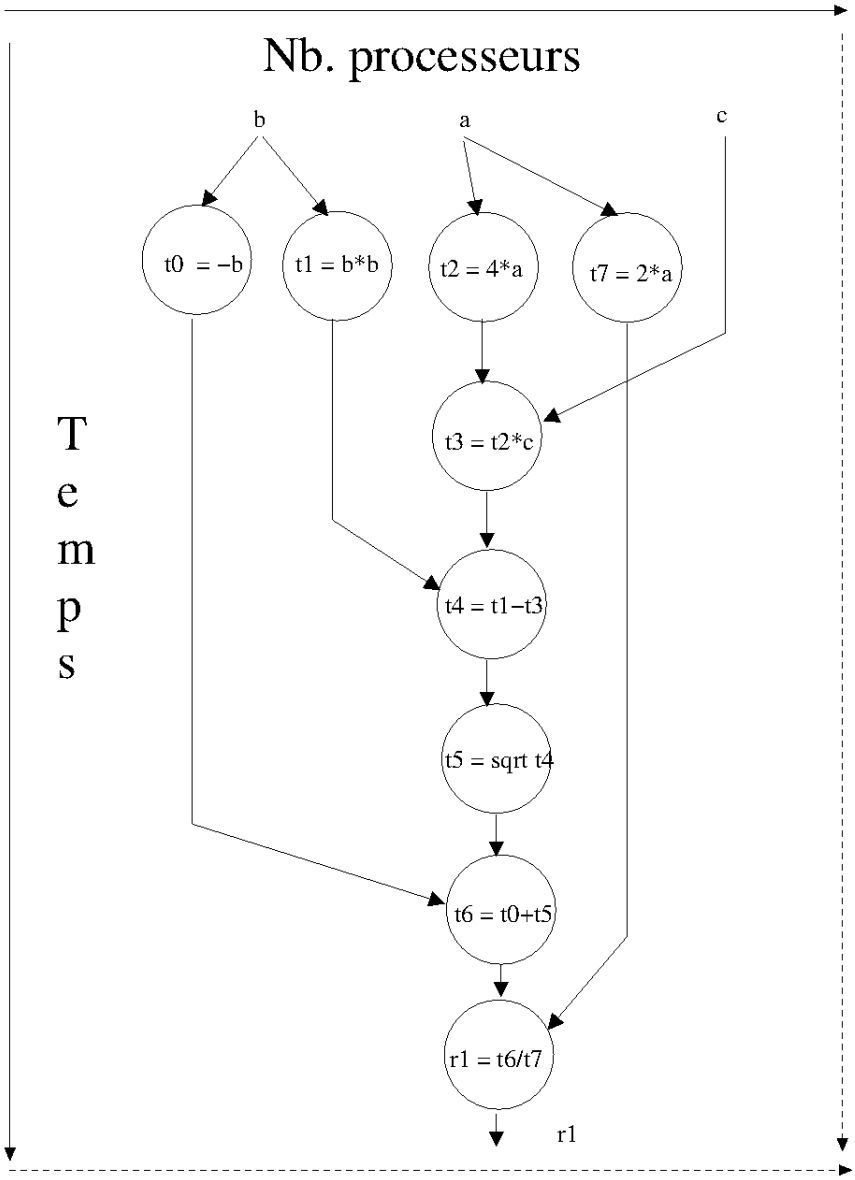
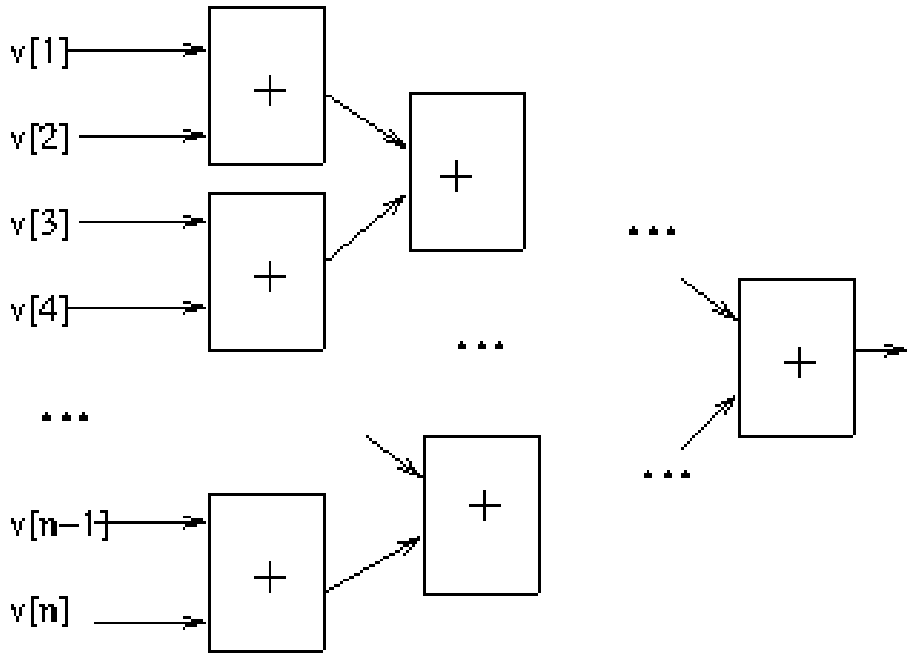


Figure 6.6: Le coût.

Soit le graphe suivant, qui représente les opérations et leurs dépendances pour faire la somme d'un tableau `v` comptant `n` éléments :



1. Quel est le temps d'exécution?
2. Quel est le travail?
3. Quel est le nombre maximal de processeurs requis?
4. Quel est le coût?

Exercice 6.1: Temps vs. coût vs. travail.

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 processeurset 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] = ... # Expression O(1)
  oc

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

procedure proc( ref int a[*], int n )
{
  for [i = 1 to n] {
    a[i] = ...   # Expression O(1)
  }
}
```

Algorithme 6.1: Petit algorithme pour illustrer que le coût n'est pas une métrique compositionnelle. Quel est le temps de cet algorithme? Quel est son travail? Quel est son coût?

Par exemple, soit la procédure `foo` présentée dans l'algorithme 6.1.

Le temps total de cette procédure sera le suivant, puisque tous les appels à «`a[i] = ...`» dans le premier `co` peuvent se faire en parallèle, de même qu'ensuite tous les appels à `proc` dans le deuxième `co` : $\Theta(1) + \Theta(n) = \Theta(n)$. À cause du premier `co`, le nombre maximum de processeurs requis par l'algorithme est n. Le coût total est donc $n \times \Theta(n) = \Theta(n^2)$ ... et non pas $\Theta(n\ lg\ n)$ 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.

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

6.3.4 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.

Question : L'algorithme avec le graphe de l'exercice précédent est-il optimal en travail? En coût?

6.4 Accélération et efficacité

6.4.1 Accélération relative vs. absolue

Lorsqu'on utilise un algorithme ou programme parallèle, on le fait dans le but d'obtenir un résultat plus rapidement.[1] Une première exigence pour qu'un algorithme/programmme parallèle soit intéressant est donc que son temps d'exécution soit inférieur au temps d'exécution d'un bon algorithme/programme séquentiel équivalent.

La notion *d'accélération* permet de déterminer de façon plus précise **combien fois** un algorithme/programme parallèle est plus rapide qu'un algorithme/programme 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 $T_P(n) = T_P(+\infty, n)$.

Notons $T_S^*(n)$ le temps requis pour *le meilleur algorithme/programme séquentiel possible* (pour un problème de taille n).

Définition 7 *L'***accélération relative** A_p^r *est définie 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/le programme s'exécutant sur une machine multi-processeurs avec p processeurs par rapport au même algorithme/programme s'exécutant sur la même machine mais avec un (1) seul processeur.

Cette mesure est souvent utilisée pour caractériser des algorithmes/programmes parallèles... car elle est plutôt *indulgente* et 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/programme séquentiel permet d'obtenir. Pour une comparaison plus juste, il est préférable d'utiliser la notion *d'accélération absolue*.

Définition 8 *L'***accélération absolue** A_p^a *est définie par le rapport suivant :*

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

[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.

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

Soit une machine avec p processeurs.

Quelle est **en théorie** la meilleure accélération absolue qu'il est possible d'obtenir?

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

Exercice 6.2: Meilleure accélération pour une machine avec p processeurs.

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/programme 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 utilise une *décomposition exploratoire* [GGKK03, Chap. 5], 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 dans l'une des branches de l'arbre.

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.

6.4.2 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 bien utilisés... ou pas**.

Définition 9 *L'*efficacité *d'un algorithme (programme) est le rapport entre le temps d'exécution séquentiel et le coût d'exécution sur une machine à p processeurs Exprimé autrement — et d'une façon plus facile à comprendre —* **l'efficacité est obtenue en divisant par p l'accélération (absolue) pour p processeurs** *:*

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

Typiquement, on exprime l'efficacité **en pourcentage**. Donc, alors que pour p processeurs, l'accélération idéale est de p, l'efficacité idéale est de $1 = 100$ %! En

[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.

d'autres mots, le cas idéal est lorsque les p processeurs sont utilisés à 100 % — ce qui est **très rare!**.

Signalons aussi que, **en pratique**, on mesurera l'efficacité comme on le fait pour l'accélération, c'est-à-dire, en variant le nombre de *threads* et en définissant p comme le nombre de *threads* utilisés.

6.5 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.[3]

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.

6.6 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 6.7, 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».

6.7 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 :

[3] «*Scalability is a parallel system's ability to gain proportionate increase in parallel speedup with the addition of more processors.*» [Gra07].

- Création des *threads* 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 surcoûts 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 les *threads* : 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* la charge et répartir le travail entre les processeurs).

- Calculs supplémentaires : l'accélération *absolue* se mesure relativement au meilleur algorithme/programme séquentiel permettant de résoudre le prob-

lème. Or, il est possible que cet algorithme/programme 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/le programme parallèle pourra être basé sur une version séquentielle moins intéressante, mais plus facilement parallélisable.

6.8 Lois d'Amdhal, de Gustafson–Barsis et fraction séquentielle expérimentale

Remarques :

- La section qui suit est (en partie) une traduction et (en partie) une adaptation du chapitre intitulé «*Performance analysis*» du livre de M.J. Quinn «*Parallel Programming in C with MPI and OpenMP*» [Qui03, Chap. 7].
- Parmi les modifications, des détails supplémentaires ont été ajoutés pour les manipulations algébriques, et certains éléments de notation ont été simplifiés ou modifiés pour refléter la notation utilisée dans la première partie des notes de cours :

	Notation de Quinn	**Notation utilisée**
Temps partie séquentielle	$\sigma(n)$	σ
Temps partie parallèle	$\varphi(n)$	φ
Accélération	$\psi(n,p)$	ψ
Temps pour taille n et p processeurs	$T(n,p)$	$T(p,n)$

6.8.1 Temps d'exécution, partie séquentielle, partie parallèle et accélération

Le temps d'exécution d'un programme paralléle pour un problème de taille n avec p processeurs — en ignorant les (sur)coûts des synchronisations et communications — peut être décomposé en deux parties :[4]

- σ = partie intrinsèquement séquentielle
- φ = partie pouvant s'exécuter en parallèle

Le temps pour un processeur est le suivant :

$$T(1,n) \quad = \quad \sigma + \varphi \tag{6.1}$$

Le temps pour p processeursest alors le suivant :

$$T(p,n) \quad = \quad \sigma + \frac{\varphi}{p}$$

L'accélération est alors bornée comme suit :

$$\psi \quad \leq \quad \frac{\sigma + \varphi}{\sigma + \varphi/p} \tag{6.2}$$

6.8.2 Loi d'Amdahl

Note : Gene Amdahl était un architecte (matériel) pour IBM, qui a ensuite a fondé sa propre compagnie de conception et construction de machines (gros *mainframes*).

Notons par f la *fraction* des opérations d'un calcul qui doivent être exécutées de façon séquentielle, avec$0 \leq f \leq 1$, donc définie comme suit :

$$f \quad = \quad \frac{\sigma}{\sigma + \varphi}$$

On a alors, par simples manipuations algébriques :

$$\sigma + \varphi \quad = \quad \frac{\sigma}{f}$$

Et :

$$\varphi \quad = \quad \frac{\sigma}{f} - \sigma = \sigma(\frac{1}{f} - 1)$$

[4] Petit truc mnémonique : σ = sigma = séquentielle. φ = phi = paralléle.

En remplaçant ces deux identités pour σ et φ dans la formule (6.2) de l'accélération, on obtient ce qui suit :

$$\begin{aligned}\psi &\leq \frac{\sigma+\varphi}{\sigma+\varphi/p} \\ &= \frac{\sigma/f}{\sigma+\sigma(1/f-1)/p} \\ &= \frac{1/f}{1+(1/f-1)/p} \\ &= \frac{1}{f+(1-f)/p}\end{aligned}$$

C'est cette relation qu'on appelle *la loi d'Amdahl* [Amd67].

Définition 10 (Loi d'Amdahl) *Soit f la fraction des opérations qui doivent être exécutées de façon séquentielle, avec $0 \leq f \leq 1$. Alors, l'accélération maximale ψ pouvant être obtenue avec p processeurs est la suivante :*

$$\psi \leq \frac{1}{f+(1-f)/p}$$

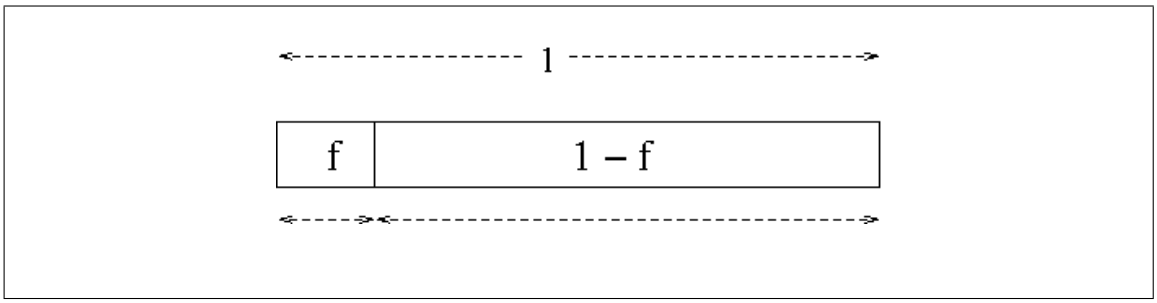


Figure 6.1: Illustration graphique de la loi d'Amdahl ≈ on **normalise** le temps d'exécution séquentielle (à 1).

Exemples :

- Soit un programme pour lequel on détermine que 90 % du temps d'exécution est pour du code qui pourrait s'exécuter en parallèle. Alors, l'accélération maximale pouvant être obtenue sur une machine à huit (8) processeurs sera la suivante :
$$\psi \leq \frac{1}{0.1+(1-0.1)/8} \approx 4.7$$

- Soit un programme pour lequel on détermine que 75 %du temps d'exécution est pour du code qui pourrait s'exécuter en parallèle. Alors, l'accélération maximale pouvant être obtenue sur une machine parallèle sans limite au nombre de processeurs sera la suivante :

$$\lim_{p\to\infty} \frac{1}{0.25 + (1 - 0.25)/p} \approx 4$$

Limites de la loi d'Amdahl

La formulation de la loi d'Amdahl ignore les surcoûts associés à l'exécution parallèle, c'est-à-dire, les coûts des synchronisations et communications entre processus. De façon plus détaillée, donc, le temps d'exécution d'un programme parallèle devrait être définie comme suit, où $\kappa(n, p)$ représente les surcoûts (*overhead*) de l'exécution parallèle :

$$T(p, n) \quad = \quad \sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)$$

Si on tient compte de ces surcoûts, *qui augmentent* lorsque le nombre de processeurs augmente, alors l'accélération *serait encore plus faible que celle prédite par la loi d'Amdahl* ☹

L'effet Amdahl

Règle générale, la complexité de $\kappa(n, p)$ est plus faible que celle de $\varphi(n)$. C'est-à-dire que le temps d'exécution augmente plus rapidement (en fonction de n) que le temps de synchronisation et communication (en fonction de n et p).

Donc, pour un nombre fixe de processeurs, l'accélération va généralement augmenter lorsqu'on augmente la taille du problème. En d'autres mots, **plus la taille du problème est grande, plus l'accélération est grande.**

6.8.3 Loi de Gustafson–Barsis

La loi d'Amdahl suppose que l'objectif essentiel d'une exécution parallèle est de **produire le résultat le plus rapidement possible**. On considère alors un problème de taille fixe, puis on examine l'accélération obtenue en augmentant le nombre de processeurs.

Toutefois, dans plusieurs domaines, l'objectif n'est pas nécessairement d'obtenir la réponse plus rapidement. L'objectif peut aussi être, pour une période de temps (durée) donné, de traiter un problème de plus grande taille — exemples : prévisions

météos, où on augmente la précision des résultats en travaillant avec des grilles plus fines, donc comportant un plus grand nombre de points, une plus grande quantité de données. En d'autres mots, il est faux de penser que la taille du problème est indépendante du nombre de processeurs.

Pour en arriver à la loi de Gustafson–Barsis, plutôt que supposer que la taille du problème est fixe, on va supposer que *le temps d'exécution parallèle est fixe*. De plus, on prend aussi comme hypothèse que la taille de la fraction intrinsèquement séquentielle n'augmente pas lorsque la taille du problème augmente.

Supposons qu'on fixe la durée d'exécution. Supposons ensuite qu'on augmente la taille du problème au fur et à mesure où on augmente le nombre de processeurs. Or, l'effet Amdahl nous indique que pour un nombre fixe de processeurs, si on augmente la taille du problème, alors l'accélération va augmenter. Or, l'accélération augmente aussi lorsqu'on augmente le nombre de processeurs. Donc, si on augmente à la fois la taille du problème et le nombre de processeurs, alors *l'accélération en sera d'autant plus augmentée!*

Intuitivement, l'*effet Amdahl* s'explique par le fait que lorsqu'on augmente la taille du problème, la fraction du temps d'exécution de la partie intrinsèquement séquentielle *diminue*. Donc, lorsqu'on ajoute des processeurs, alors on peut traiter des problèmes de plus grande taille. Dans ce cas, la fraction parfaitement parallélisable augmente, donc la fraction intrinsèquement séquentielle diminue, donc l'accélération augmente :

On ajoute des processeurs $\Rightarrow$ On peut traiter des problèmes plus gros

$\Rightarrow$ La fraction parfaitement parallélisable augmente

$\Rightarrow$ La fraction intrinsèquement séquentielle diminue

$\Rightarrow$ L'accélération augmente

Notons par s la fraction du *temps d'exécution parallèle* consacrée aux opérations

séquentielles, i.e., s est définie comme suit : [5]

$$s = \frac{\sigma}{\sigma + \varphi/p}$$

On a alors, par simples manipulations algébriques :

$$\sigma = (\sigma + \varphi/p)s$$

Et : [6]

$$\varphi = (\sigma + \varphi/p)(1 - s)p$$

Donc, en substituant ces deux dernières identités dans le numéreateur de la formule (6.2) de l'accélération et en factorisant $\sigma + \varphi/p$:

$$\begin{aligned}
\psi &\leq \frac{\sigma + \varphi}{\sigma + \varphi/p} \\
&= \frac{(\sigma + \varphi/p)(s + (1 - s)p)}{\sigma + \varphi/p} \\
&= s + (1 - s)p \\
&= s + p - sp \\
&= p + (1 - p)s
\end{aligned}$$

C'est cette relation qu'on appelle *la loi de Gustafson–Barsis* [Gus88].

[5] On remarque la différence par rapport à la définition de la fraction f, où f était définie par rapport au temps séquentiel (pour un processeur) et non par rapport au temps parallèle (pour p processeurs) :

$$f = \frac{\sigma}{\sigma + \varphi}$$

Donc, alors que f était la fraction du temps pour l'exécution de la partie intrinsèquement séquentielle sur une machine uniprocesseur, s est la fraction du temps pour l'exécution de la partie intrinsèquement séquentielle sur machine parallèle à p processeurs.

[6] On a que :

$$1 = \frac{\sigma + \varphi/p}{\sigma + \varphi/p} = \frac{\sigma}{\sigma + \varphi/p} + \frac{\varphi/p}{\sigma + \varphi/p} = s + (1 - s)$$

Définition 11 (Loi de Gustafson–Barsis) *Pour un problème de taille n traité avec p processeurs, soit s la fraction du temps d'exécution (parallèle) consacrée à la partie intrinsèquement séquentielle (avec $0 \leq s \leq 1$). Alors, l'accélération maximale ψ pouvant être obtenue est la suivante :*

$$\begin{aligned} \psi &\leq \frac{s + (1-s)p}{1} \\ &= s + (1-s)p \\ &= p + (1-p)s \end{aligned}$$

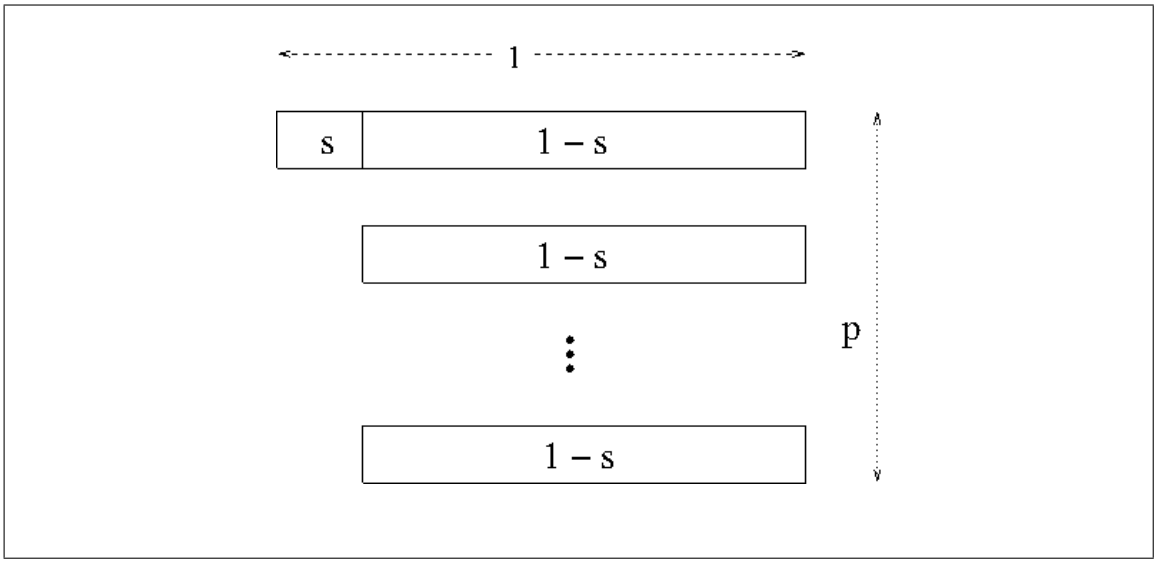


Figure 6.2: Illustration graphique de la loi de Gustafson–Barsis $\approx$ on **normalise** le temps d'exécution parallèle.

Cette accélération peut donc être inteprétée comme étant définie comme suit (voir Figure au tableau) :[7]

$$\frac{\text{Temps que l'exécution du programme } \textit{aurait pris} \text{ avec un seul processeur}}{\text{Temps que l'exécution } \textit{a vraiment pris} \text{ avec } p \text{ processeurs}}$$

L'accélération prédite par la loi de Gustafson–Barsis est aussi appelée ***scaled speedup***, parce qu'en utilisant le temps parallèle comme point de comparaison (pour déterminer la fraction séquentielle) plutôt que le temps séquentiel, on fait en sorte que la taille du problème soit une fonction croissante du nombre de processeurs.

[7] En supposant que le processeur a suffisamment d'espace mémoire.

Signalons que Gustafson [Gus88] a formulé cette loi après avoir constaté, expérimentalement, que certains problèmes s'exécutaient de façon très efficace, i.e., avec une forte accélération, sur des machines avec un grand nombre de processeurs (accélération $\approx$ 1016–1020 pour 1024 processeurs).

Exemple :

- Soit un programme s'exécutant en 220 secondes sur 64 processeurs. Des mesures et expérimentations (*benchmarking*, profilage) permettent de déterminer que 5 % du temps d'exécution est consacré à des parties intrinsèquement séquentielles. Quelle sera alors l'accélération (*scaled speedup*) obtenue par ce programme?

$$\psi \quad = \quad 64 + (1 - 64)(0.05) = 64 - 3.15 = 60.85$$

Analogie pour comparer la loi d'Amdahl et la loi de Gustafson

L'analogie suivante permet d'illustrer la différence entre la loi d'Amdahl et la loi de Gustafson[8]. Soit une automobile qui se déplace à 30 km/h d'une ville A vers une ville B, distantes l'une de l'autre de 60 km. Supposons que la voiture roule déjà depuis une heure, donc a franchi 30 km.

- La loi d'Amdahl suggère que peu importe la vitesse à laquelle on se déplace durant la deuxième partie du trajet, il sera impossible d'arriver à une moyenne de 90 km/h — même si l'automobile se déplaçait à une vitesse infinie, la vitesse moyenne serait alors de 60 km/h.
- La loi de Gustafson suggère qu'avec suffisamment de temps et une distance assez longue, alors il sera toujours possible d'arriver à une vitesse moyenne de 60 km/h. Par exemple, si l'automobile se déplace à 150 km/h pendant une heure, alors on atteindra la vitesse moyenne de 60 km/h.

[8] `http://en.wikipedia.org/wiki/Gustafson's_law`

6.8.4 Accélération selon la loi d'Amdhal vs. selon la loi de Gustafson-Barsis

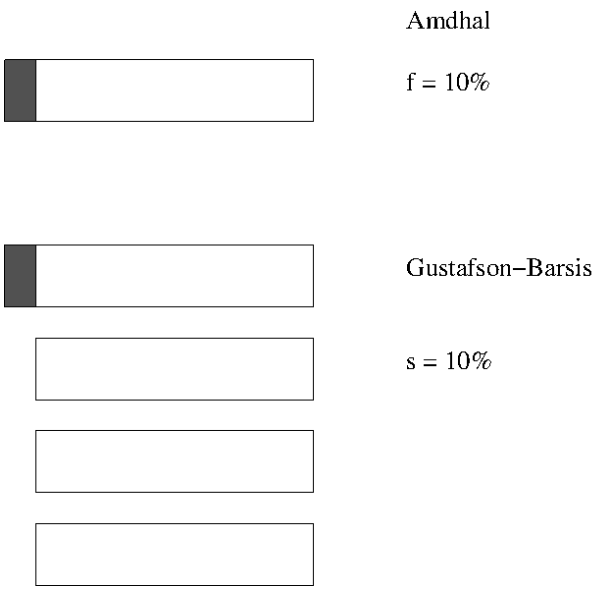
Le tableau 6.1 présente l'accélération obtenue pour diverses valeurs de f (loi d'Amdhal) ou s (loi de Gustafson–Barsis) et divers nombres de processeurs.

	Nb. procs.	Amdhal	Gustafson	Amdhal	Gustafson	Amdhal	Gustafson
f **ou** s		**0,05**	**0,05**	**0,10**	**0,10**	**0,20**	**0,20**
	2	1,90	1,95	1,82	1,90	1,67	1,80
	4	3,48	3,85	3,08	3,70	2,50	3,40
	8	5,93	7,65	4,71	7,30	3,33	6,60
	16	9,14	15,25	6,40	14,50	4,00	13,00
	32	12,55	30,45	7,80	28,90	4,44	25,80
	64	15,42	60,85	8,77	57,70	4,71	51,40
	128	17,41	121,65	9,34	115,30	4,85	102,60

Tableau 6.1: Accélération maximale possible pour diverses valeurs de f ou s et divers nombres de processeurs.

Une autre façon de voir ces deux lois :

- Loi d'Amdhal
  - Borne inférieure de l'accélération
  - Hypothèse *pessimiste* = la fraction séquentielle reste constante, peu importe la taille du problème = Le temps intrinséquement séquentiel augmente au même rythme que la taille du problème ☹
- Gustafson-Barsis
  - Borne supérieure de l'accélération
  - Hypothèse *optimiste* = le temps intrinsèquement séquentiel reste constant, peu importe la taille du problème
- La réalité... est quelque part entre les deux
  - Le temps intrinsèquement séquentiel augmente, mais *beaucoup plus lentement* que la taille du problème
  - $\Rightarrow$ si on augmente la taille du problème, alors on pourra augmenter le nombre de processeurs pour obtenir une meilleure accélération ☺



- Accélération Amdhal avec 4 processeurs
  - 10 / (1 + (9/4)) = 10 / (1 + 2.25) = 3.08
- Accélération Gustafson–Barsis avec 4 processeurs
  - (1 + 4*9) / 10 = 37 / 10 = 3.7

6.8.5 Fraction séquentielle déterminée expérimentalement

Karp & Flatt [KF90] ont proposé une «*métrique*» pour tenter de comprendre et évaluer le comportement de programmes parallèles — donc en plus des métriques d'accélération et d'efficacité.

Appelons e la *fraction* du temps d'exécution associée à la partie intrinsèquement séquentielle, donc définie comme suit :

$$e = \frac{\sigma}{T(1,n)} \tag{6.3}$$

Alors :

$$\sigma = eT(1,n)$$

$$\begin{aligned} \varphi &= T(1,n) - \sigma && \text{[Déduit de l'équation (6.1)]} \\ &= T(1,n) - eT(1,n) && \text{[Déduit de l'équation (6.3)]} \\ &= (1-e)T(1,n) \end{aligned}$$

Donc :

$$T(p,n) = \sigma + \frac{\varphi}{p} \quad (6.4)$$

$$= eT(1,n) + \frac{(1-e)T(1,n)}{p} \quad (6.5)$$

Soit l'accélération[9], notée ψ, définie comme suit :

$$\psi = \frac{T(1,n)}{T(p,n)} \quad (6.6)$$

Donc :

$$\begin{array}{lll} T(p,n) = eT(1,n) + (1-e)T(1,n)/p & & \text{[De (6.5)]} \\ = e\psi T(p,n) + (1-e)\psi T(p,n)/p & & \text{[Déduit de (6.6)]} \end{array}$$

En divisant par $T(p,n)$ de chaque coté, on obtient :

$$1 = e\psi + \frac{(1-e)\psi}{p}$$

En divisant par ψ de chaque coté, on a alors :

$$\begin{aligned} \frac{1}{\psi} &= e + \frac{(1-e)}{p} \\ &= e + \frac{1}{p} - \frac{e}{p} \\ &= e(1 - \frac{1}{p}) + \frac{1}{p} \end{aligned}$$

Finalement, en isolant e à gauche :

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Cette métrique, déterminée de façon *expérimentale*, à partir de l'accélération obtenue pour un programme donné s'exécutant sur une machine avec p processeurs, est appelée la *fraction séquentielle déterminée expérimentalement.*

Règle générale, pour un problème de taille donnée, lorsqu'on augmente le nombre de processeurs, l'efficacité décroit. On peut alors utiliser cette métrique pour tenter d'expliquer les causes de cette baisse d'efficacité :

[9]Donc, accélération *relative.*

- Parce que le programme ne contient pas suffisamment de parallélisme.
- Parce que les surcoûts liés à l'exécution parallèle augmentent trop rapidement.

Exemples [KF90] :

- Pour qu'une machine soit utilisée efficacement, il faut que la charge soit bien balancée entre les processeurs — temps de travail équivalent pour chacun des processeurs.

  Supposons qu'on a 12 tâches à distribuer. Pour $p = 2, 3, 4, 6$ et 12, la charge sera bien balancée entre les processeurs. Par contre, pour d'autres valeurs de p, la charge ne sera pas bien balancée, même si l'accélération s'améliore. Or, un débalancement de la charge va conduire à des valeurs anormalement croissantes de la fraction e.

- Les surcoûts de synchronisation et communication augmentent lorsqu'on accroit le nombre de processeurs. Cette augmentation des surcoûts diminue l'accélération, et conduit à un accroissement de e. Une telle augmentation de e est alors probablement un signe que la granularité des tâches est trop fine, i.e., qu'il y a trop de surcoûts de synchronisation et communication.

6.A Mesures de performances : Un exemple concret

Dans cette section, nous allons examiner un exemple concret de mesures de performances pour un programme sur Java s'exécutant une machine à coeurs/processeurs multiples.

Le Programme Java 6.1 présente un programme Java permettant d'approximer la valeur de π à l'aide d'une méthode Monte Carlo avec plusieurs *threads* — un programme avec parallélisme semi-embarrassant. Il s'agit d'une version Java d'une fonction Ruby vue précédemment : Programme Ruby **??** (style impératif). Voir la Section **??** pour les explications concernant cette mise en oeuvre Java.

```
public static int nbDansCercleSeq( int nbLancers ) {
   Random rnd = new Random();
   int nb = 0;
   for( int k = 0; k < nbLancers; k++ ) {
      double x = rnd.nextDouble();
      double y = rnd.nextDouble();
      if( x * x + y * y <= 1.0 ) {
         nb += 1;
      }
   }
   return nb;
}
```

Programme Java 6.1 Une fonction parallèle `evaluerPi` (et sa fonction auxiliaire `nbDansCercleSeq`) pour approximer la valeur de π à l'aide de la méthode de Monte Carlo à plusieurs *threads*.

```
@SuppressWarnings("unchecked")
public static double evaluerPi( final int nbLancers,
                                int nbThreads ) {
   ExecutorService pool =
     Executors.newFixedThreadPool( nbThreads );

   Future<Integer> lesNbs[] = new Future[nbThreads]; // unchecked cast
   for( int k = 0; k < nbThreads; k++ ) {
      lesNbs[k] = pool.submit( () -> {
        return nbDansCercleSeq( nbLancers/nbThreads );
      } );
   }

   int nbTotalDansCercle = 0;
   for( int k = 0; k < nbThreads; k++ ) {
      try { nbTotalDansCercle += lesNbs[k].get(); }
        catch(Exception ie) {}
   }
   pool.shutdown();

   return 4.0 * nbTotalDansCercle / nbLancers;
}
```

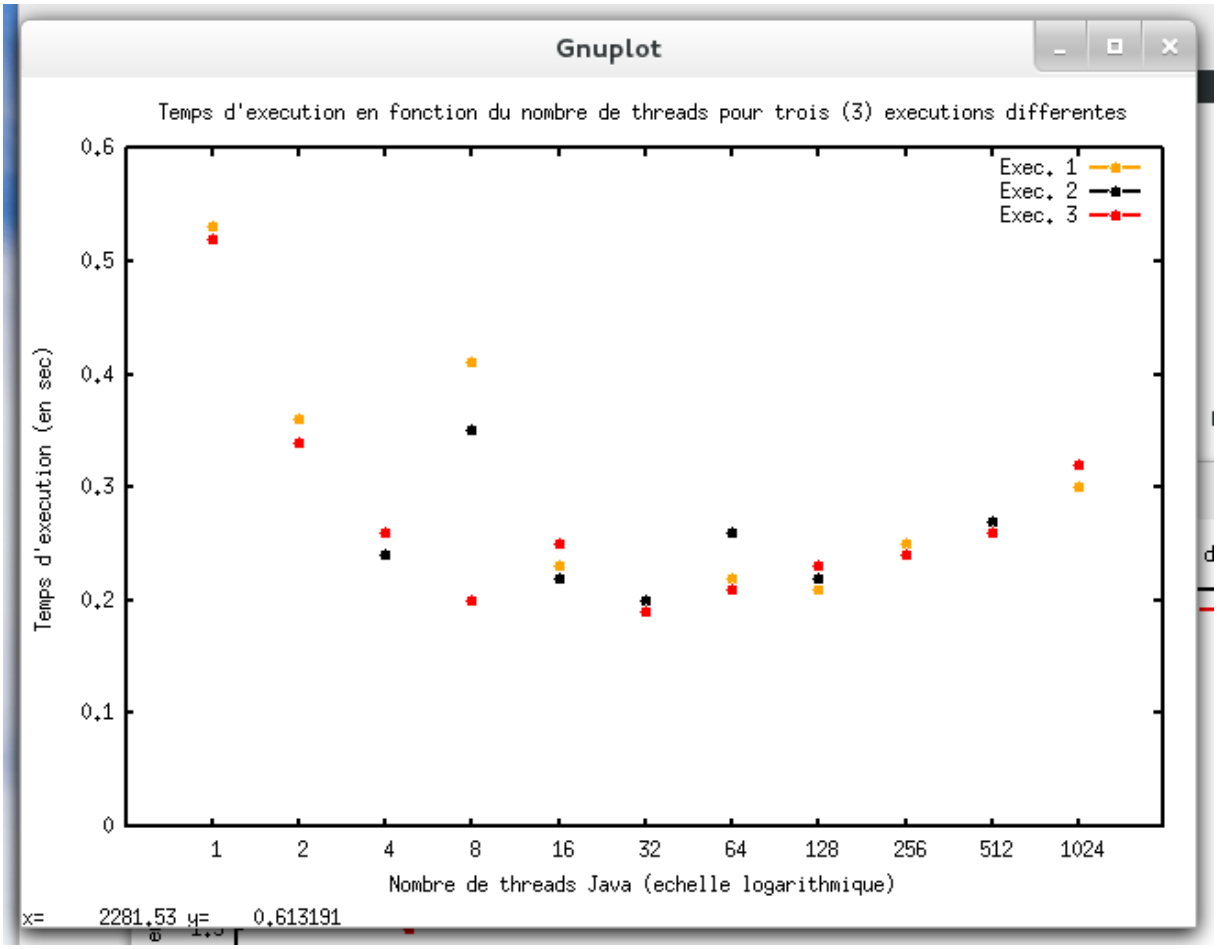


Figure 6.7: Graphe donnant le temps d'exécution du programme `Pi.java` pour différents nombres de *threads*. Les temps sont indiqués pour trois (3) séries différentes d'exécution.

La Figure 6.7 présente les temps d'exécution pour trois séries distinctes d'exécution du programme Java **??**, et ce en faisant varier le nombre de *threads* — 1, 2, 4, 8, 16, . . . , 1024 *threads*. Dans tous les cas, on effectue un total de **10 000 000 lancers** — donc 10 000 000 lancers *partagés* entre les divers *threads*.

Quelques remarques sur ces expérimentations et résultats :

- Le programme a été exécuté sur une machine Linux avec 8 coeurs.
- L'échelle des x, qui indique le nombre de *threads* Java, est *logarithmique*. C'est ce qui explique que l'écart entre les valeurs indiquées sur cet axe est constant, même si à chaque fois on double le nombre de *threads*.

- Le temps pour une exécution avec un certain nombre de *threads* n'est pas toujours le même — il varie d'une fois à une autre. Cela est tout à fait normal et usuel, car plusieurs facteurs peuvent influencer le temps d'exécution — ordre différent de préemption des *threads*, temps variable d'accès au cache ou à la mémoire, etc.

- On constate que, au début, lorsqu'on augmente le nombre de *threads*, en gros le temps d'exécution diminue. C'est le cas même lorsqu'on utilise *plus de* threads *que le nombre de processeurs/coeurs!*

  Par contre, lorsque le nombre de *threads* devient trop grand, alors le temps d'exécution recommence à augmenter! C'est le phénomène de la courbe en U décrit à la section **??**.

- On remarque qu'il y a une certaine *flexibilité* quant au nombre de *threads* : le temps d'exécution semble le plus bas lorsqu'on utilise entre 8 et 64 *threads*. Donc, la valeur choisie pour le nombre de *threads* n'a pas à être exacte et précise — on a un certain jeu. Par contre, si ce nombre est trop petit ou trop grand, alors les résultats sont moins bons ☹

  Soulignons que ce n'est que de façon expérimentale que l'on peut déterminer les meilleures valeurs pour ce genre de paramètres : il n'y a pas de formule magique pour trouver la ou les bonne valeurs, car trop de facteurs entrent en jeu.

figures 6.8 et 6.9, quant à elles, présentent les deux graphes suivants :

- Le *graphe d'accélération relative* : ce graphe indique, pour les divers nombres de *threads nbt*, de combien de fois cette version avec *nbt threads* est plus rapide que la version avec un (1) *thread*.

- Le *graphe d'efficacité* : ce graphe indique, pour les divers nombres de *threads*, l'efficacité — le pourcentage d'utilisation des ressources de la machine. Comme il s'agit d'une machine à 8 processeurs, on aimerait que le programme, dans le meilleur des cas, puisse s'exécuter 8 fois plus rapidement. Malheureusement, ce n'est pas le cas — et c'est rarement le cas : ici, dans le meilleur des cas, soit avec 32 *threads*, l'accélération est de 2.6, donc une efficacité de 32.5 % ($\frac{2.6}{8}$).

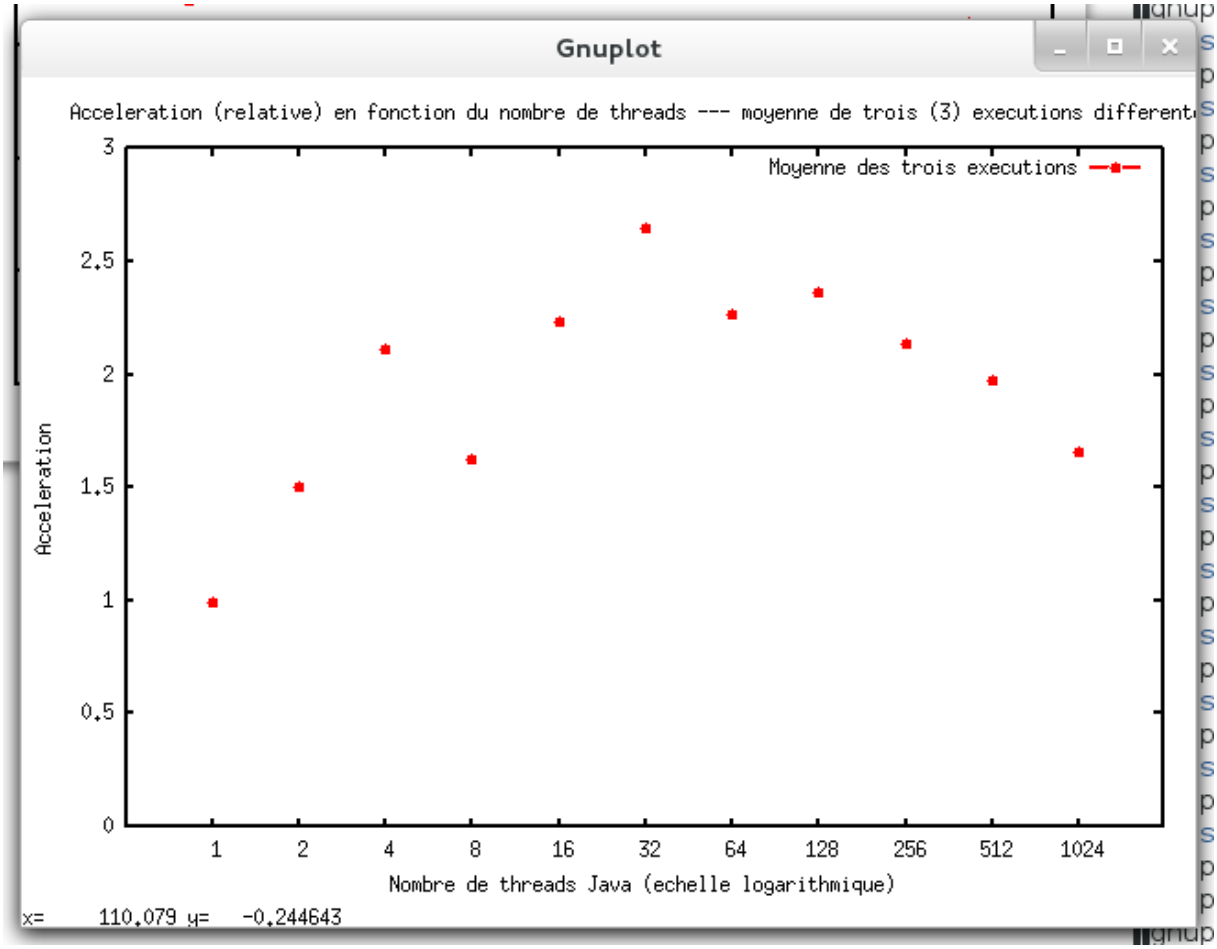


Figure 6.8: Graphe d'accélération du programme `Pi.java` pour différents nombres de *threads*— moyenne de trois (3) éxécutions.

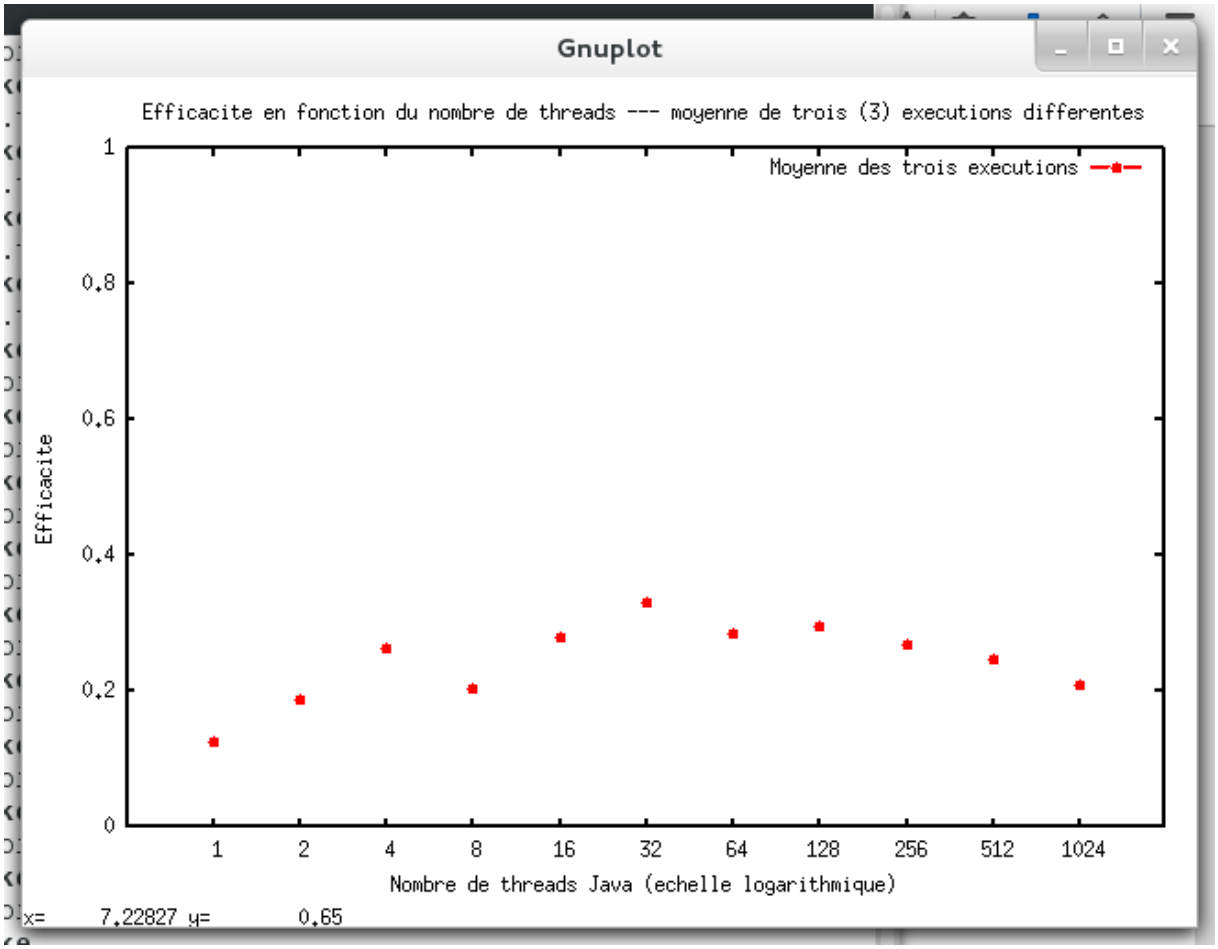


Figure 6.9: Graphe d'efficacité du programme `Pi.java` pour différents nombres de *threads*— moyenne de trois (3) éxécutions, pour huit (8) processeurs.

Références

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. 1967 AFIPS Conf.*, volume 30, page 483. AFIPS Press, 1967.

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

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing (Second Edition)*. Addison-Wesley, 2003.

[Gra07] J.R. Graham. Integrating parallel programming techniques into traditional computer science curricula. *Inroads — SIGCSE Bulletin*, 39(4):75–78, Dec, 2007.

[Gus88] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

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

[KF90] A.H. Karp and H.P Flatt. Measuring parallel processor performance. *Communications of The ACM*, 33(5):539–543, May 1990.

[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].

[Qui03] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.