# 7. Métriques de performance pour algorithmes et programmes parallèles

## 7.1 Introduction : le temps d'exécution suffit-il?

Algorithme/programme parallèle

$\Rightarrow$ Utilisation de plusieurs processeurs

$\Rightarrow$ Réduction du temps d'exécution ☺

$\Rightarrow$ Augmentation du *coût* de la machine ☹

L'algorithme parallèle peut parfois utiliser une mise en oeuvre qui augmente la quantité totale de travail effectuée ☹

Donc, **non, le temps d'exécution ne suffit pas!**

Dans ce qui suit :

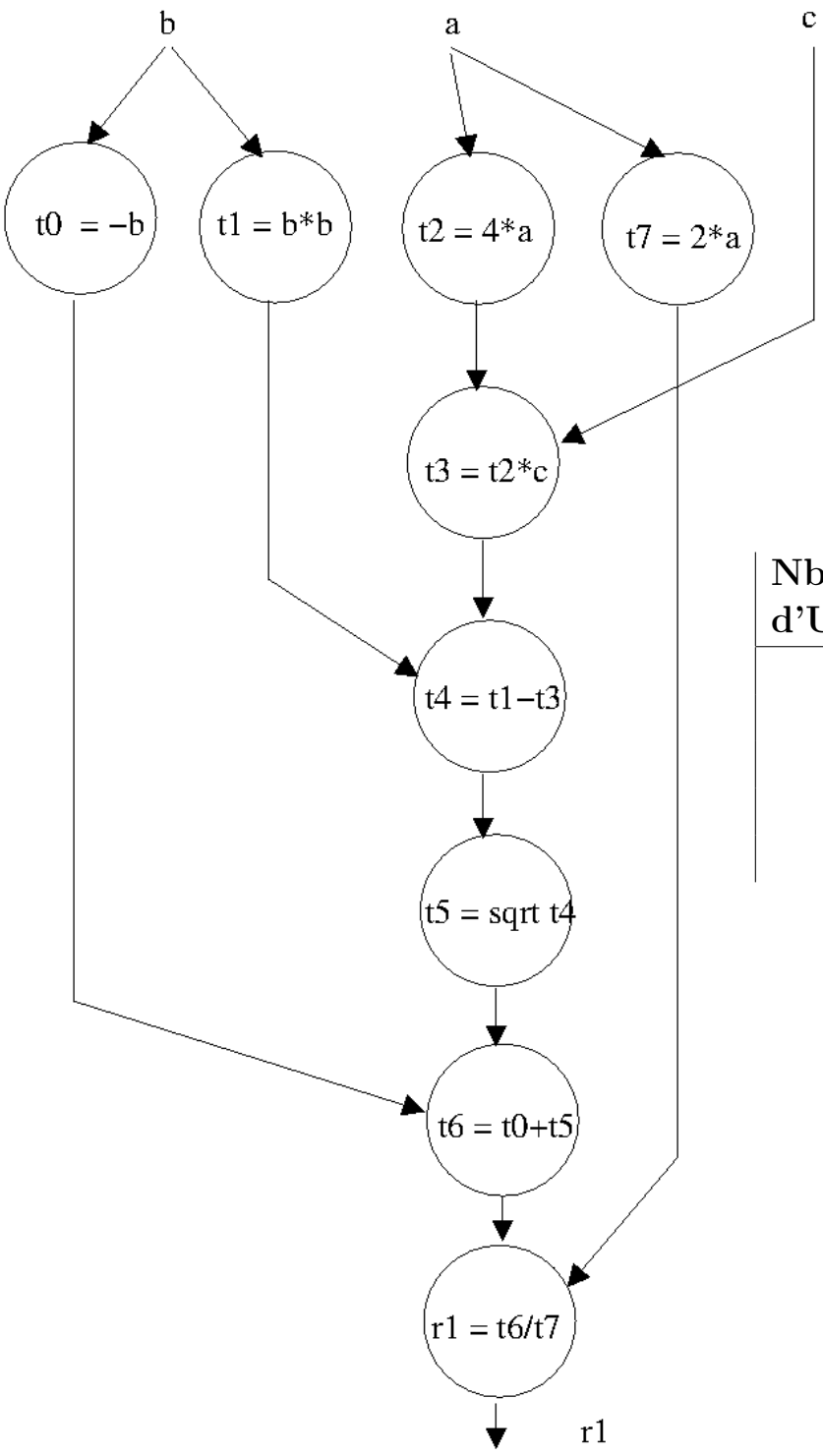
1. Métriques *asymptotiques*, pour des algorithmes
2. Métriques *réelles*, pour des programmes

## 7.2 Temps d'exécution et profondeur

**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.*

**Définition 2** *La* profondeur *d'un calcul est définie comme* **la longueur de la plus longue chaîne de dépendances** *présentes dans ce calcul.*

La profondeur représente le meilleur temps d'exécution possible en supposant une machine idéale.




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

**Figure 7.1:** 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.

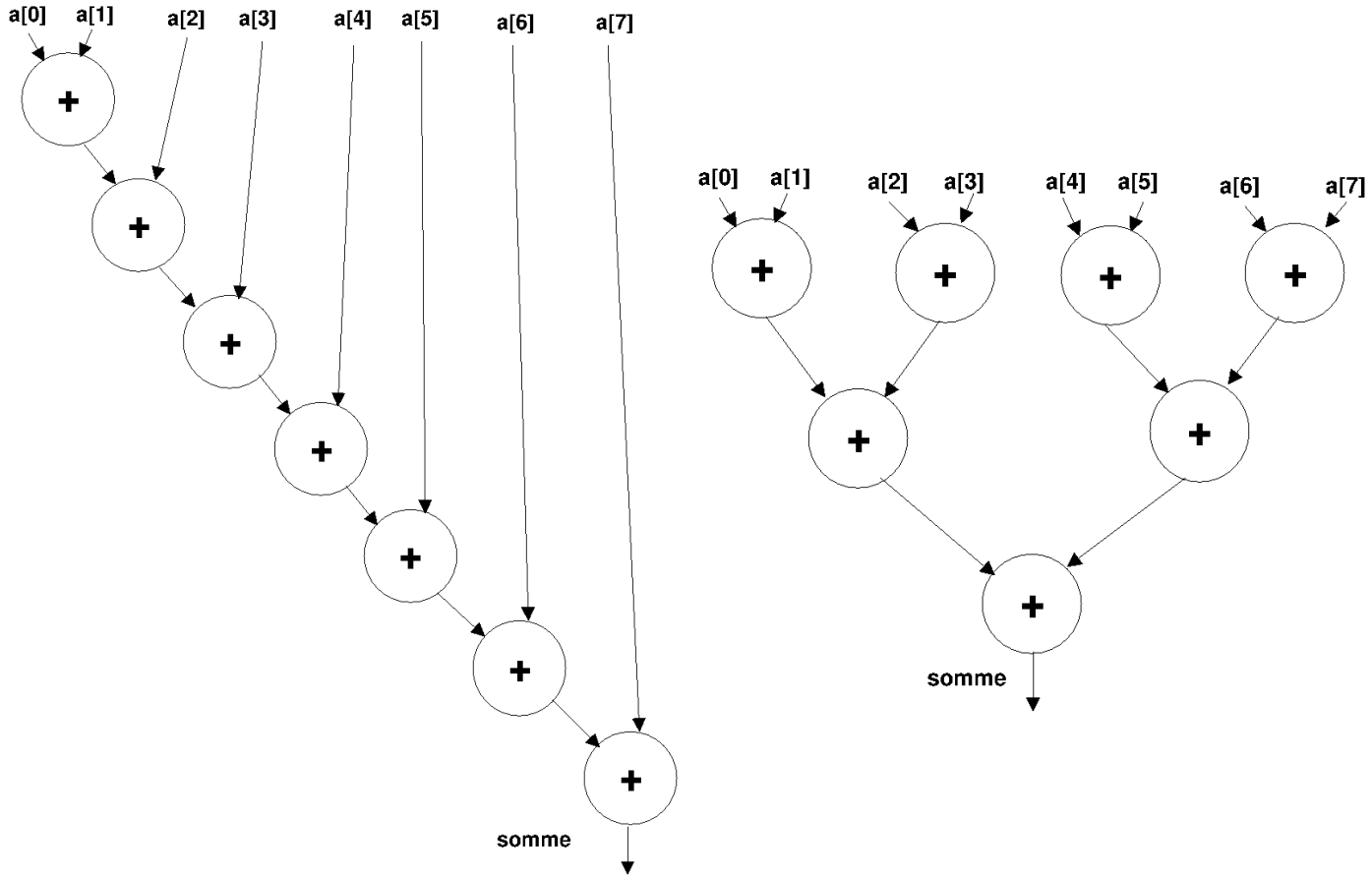



**Figure 7.2:** Graphe de dépendances pour calcul de la somme d'un tableau de huit éléments, pour illustrer la notion du meilleur temps parallèle comme longueur du plus long chemin.

Il s'agit de métriques *asymptotiques* (idéales).

**En pratique**, on va mesurer le temps d'exécution réel, *en utilisant avec un nombre* **limité** *de processeurs*

## 7.3 Coût, travail et optimalité

### 7.3.1 Coût

**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 $A$ est alors* $C(n) = p(n) \times T_P(n)$

### 7.3.2 Travail

**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.*

### 7.3.3 Coût vs. travail

Le travail est un estimé plus précis du **nombre total exact d'opérations** exécutées

## Relation entre temps, travail et coût

- Séquentiel :

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

- Parallèle :

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

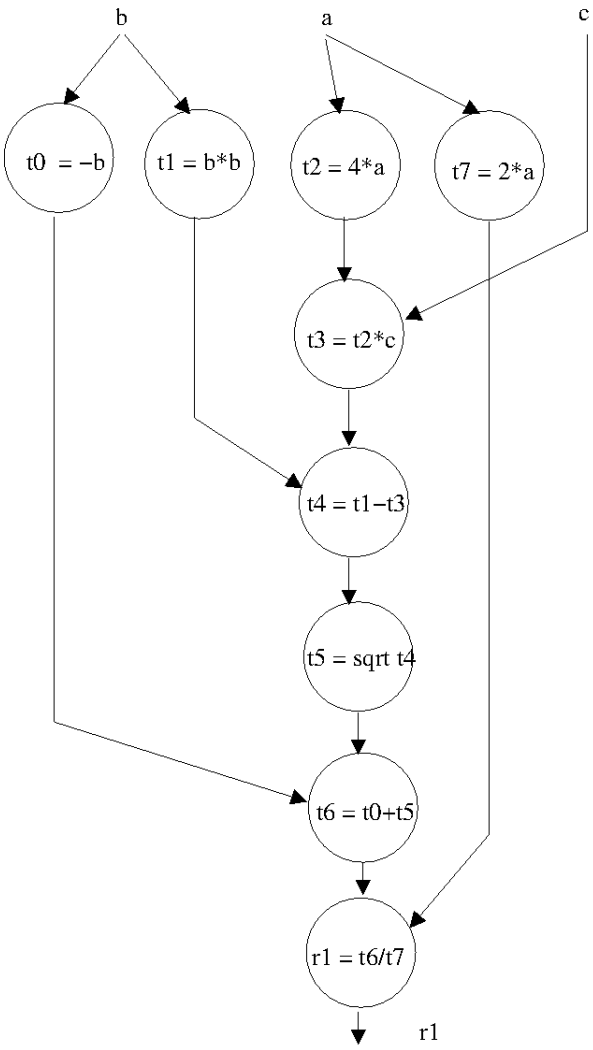



**Figure 7.3:** Le travail.

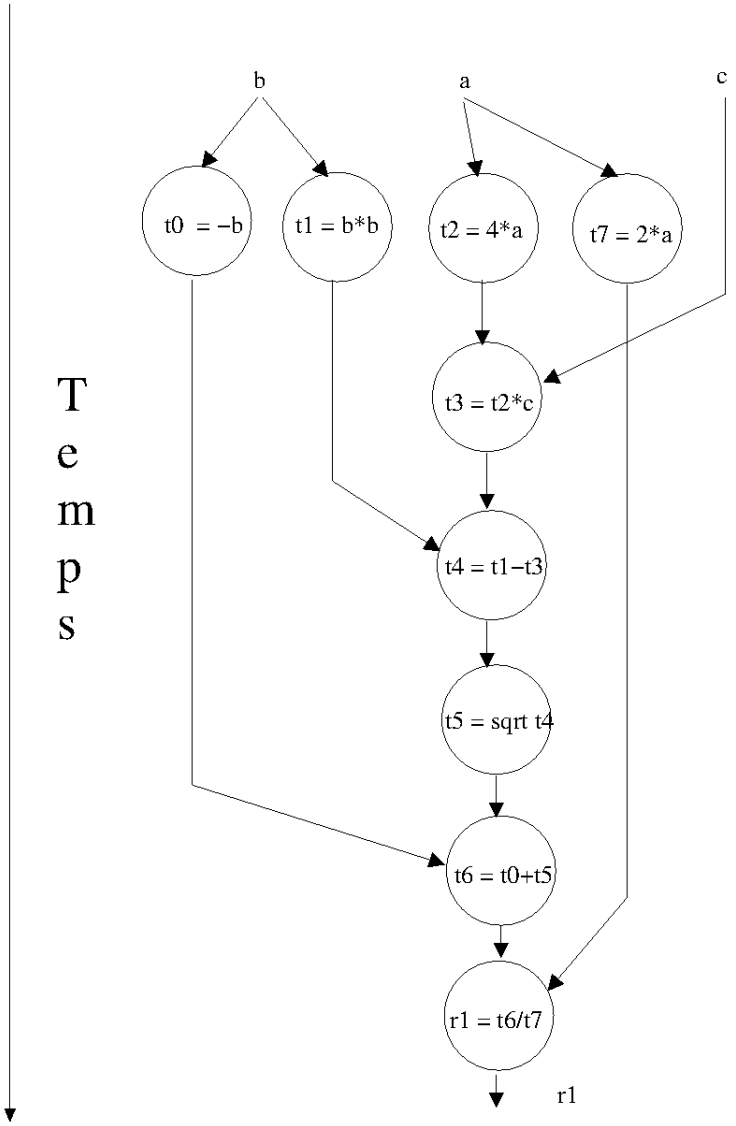



**Figure 7.4:** Le temps.

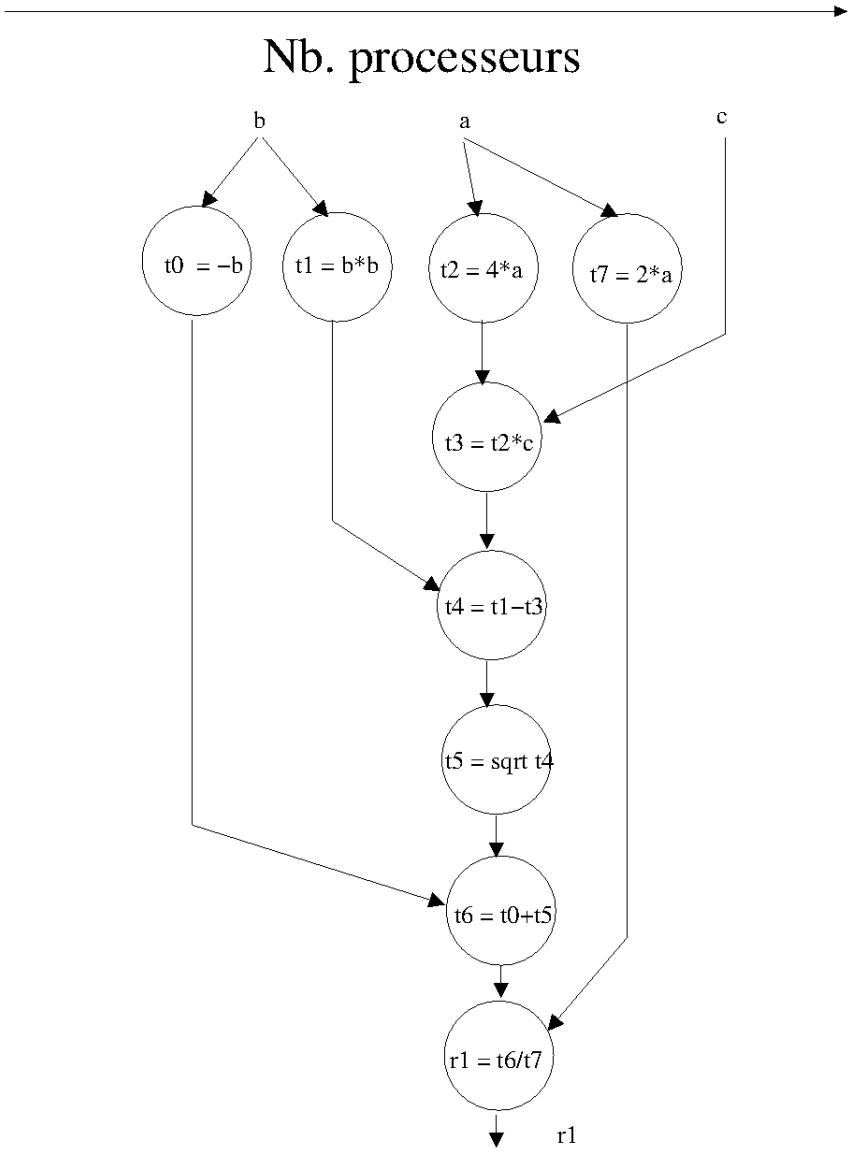



**Figure 7.5:** Le nombre de processeurs.

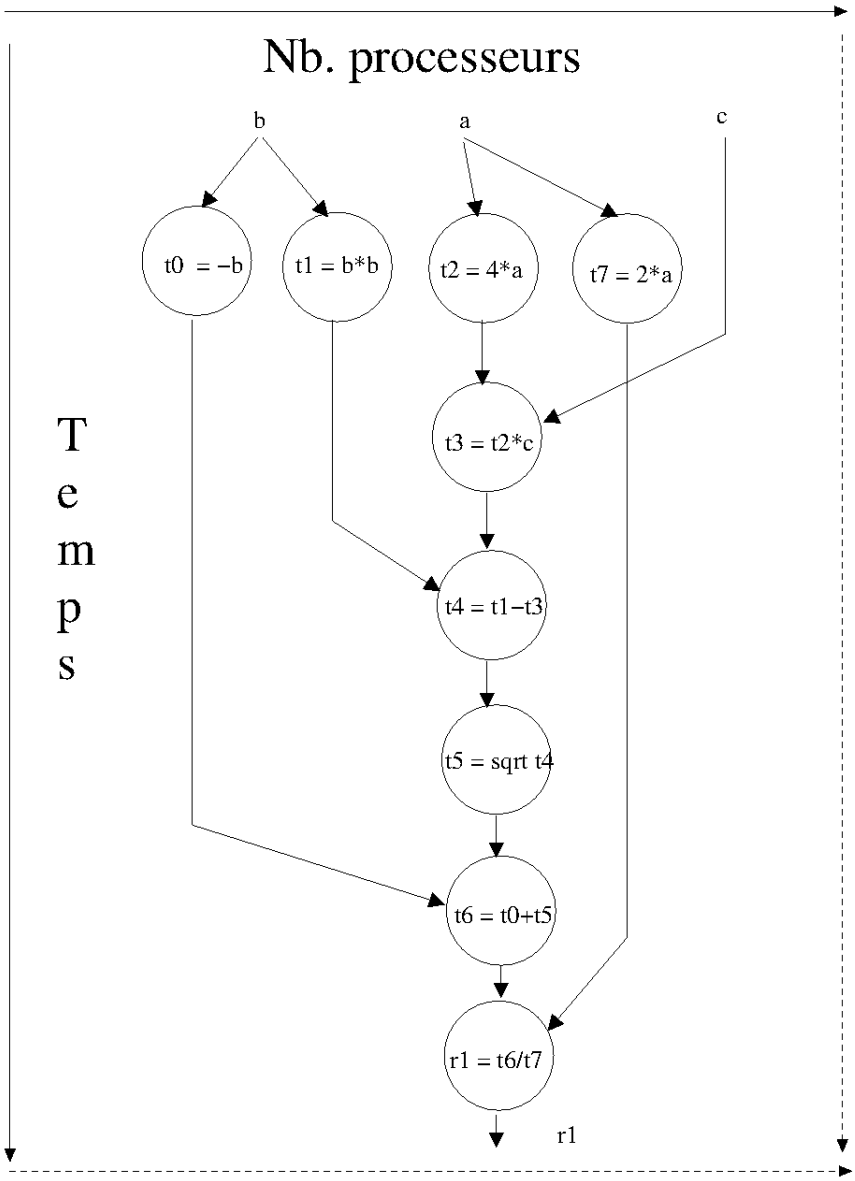
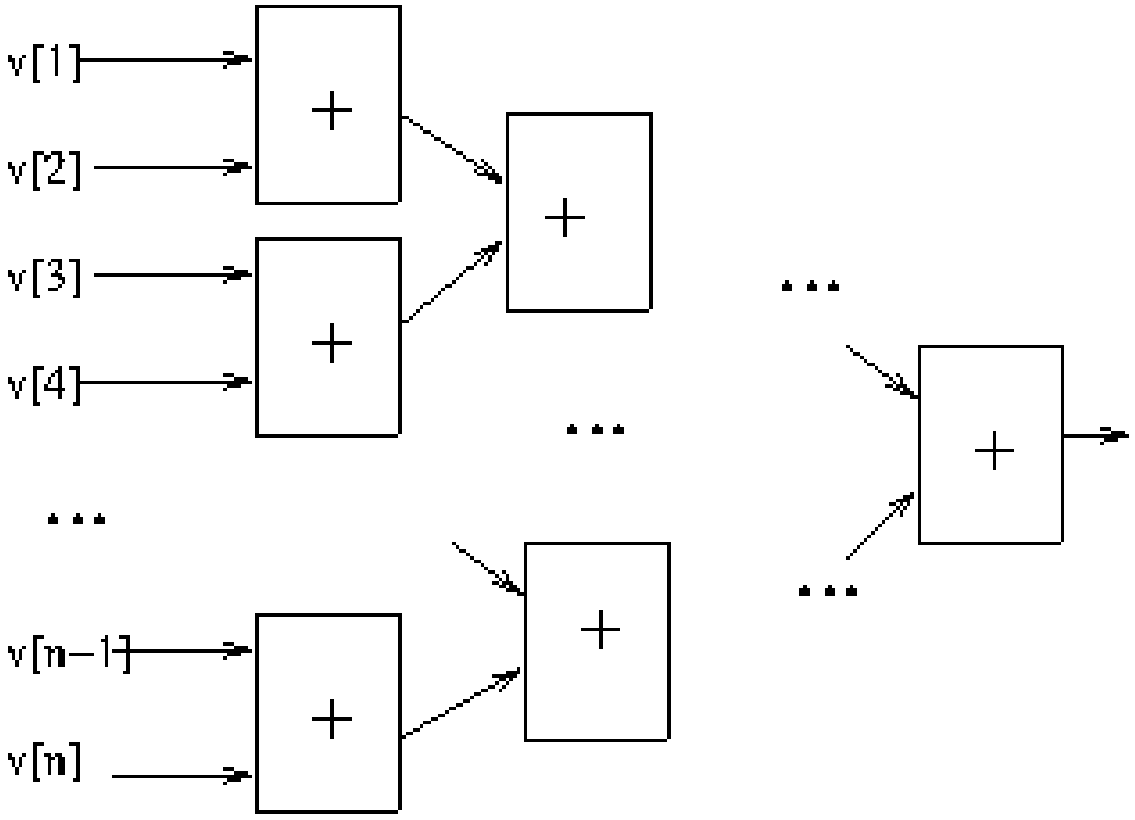



**Figure 7.6:** Le coût.

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 7.1:** Temps vs. coût vs. travail.

Autre différence entre coût et travail :

le travail est une métrique *compositionnelle*,

pas le coût.

Soit $S_1$ et $S_2$ deux segments de code.

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

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

# Exemple illustrant la non-compositionalité

```
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)
  }
}
```

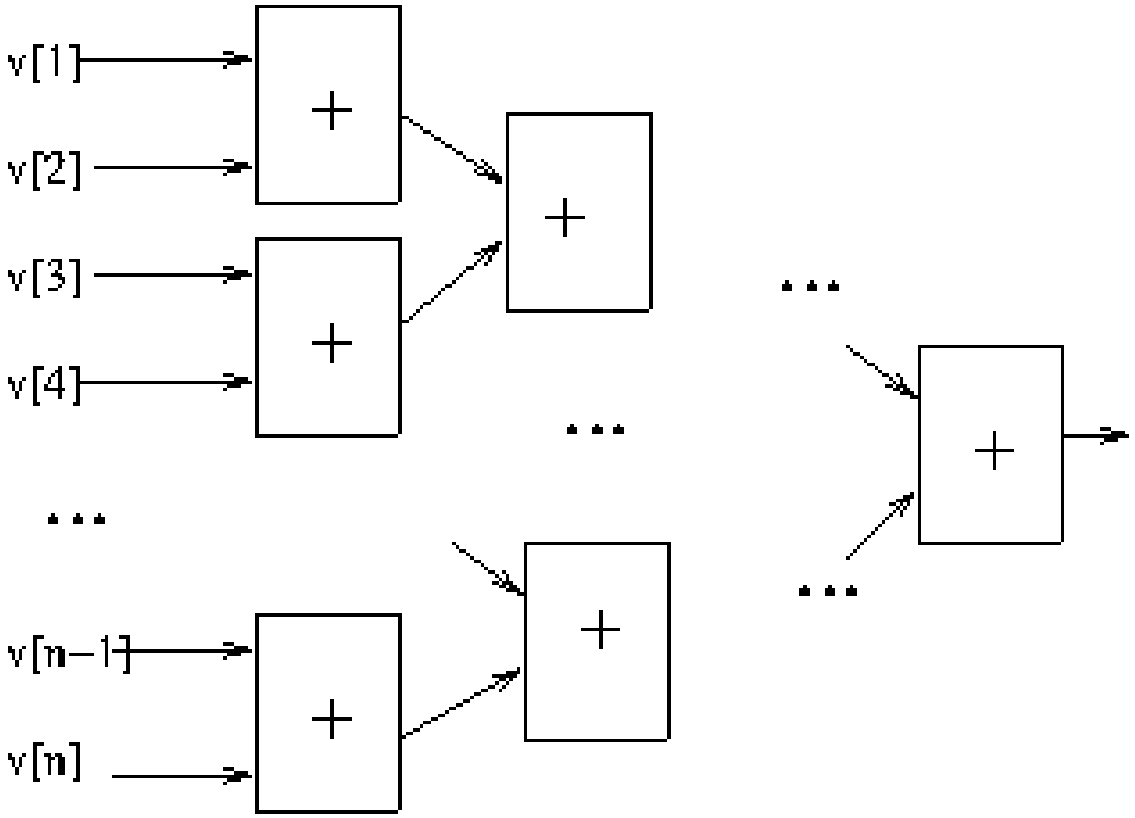
**Algorithme 7.1:** Petit algorithme pour illustrer que le coût n'est pas une métrique compositionnelle. Quel est le temps de cet algorithme? Quel est son travail? Quel est son coût?

### 7.3.4 Optimalité

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

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

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



## 7.4 Accélération et efficacité

### 7.4.1 Accélération relative vs. absolue

Détermine **combien fois** un programme parallèle est plus rapide qu'un programme séquentiel.

Notons $T_P(p, n)$ le temps pour un problème de taille $n$ avec $p$ processeurs — $T_P(n) = T_P(+\infty, n)$.

Notons $T_S^*(n)$ le temps requis pour *le meilleur programme séquentiel* (pour taille $n$).

**Définition 7** *L'*accélération relative $A_p^r$ *:*

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

**Définition 8** *L'*accélération absolue $A_p^a$ *:*

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

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 7.2:** Meilleure accélération pour une machine avec $p$ processeurs.

**Accélération linéaire vs. superlinéaire**

On parle d'accélération **linéaire** lorsque $A_p = p$.

En pratique, des accélérations linéaires sont **rares** ☹

Bizarrement, on rencontre parfois des accélérations **superlinéaires**, où $A_p > p$.

### 7.4.2 Efficacité de l'utilisation des processeurs

L'efficacité nous indique **dans quelle mesure les divers processeurs sont bien utilisés... ou pas**.

**Définition 9** *L'*efficacité *d'un programme est le rapport entre le temps d'exécution séquentiel et le coût d'exécution sur une machine à $p$ processeurs :*

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

**Donc** :

Efficacité = Accélération divisé par $p$!

- Généralement exprimé **en pourcentage**
  Que signifie une efficacité de 50 %?

## 7.5 Dimensionnement (*scalability*)

Un **algorithme** est *dimensionnable* lorsque le niveau de parallélisme augmente au moins de façon linéaire avec la taille du problème.

Une **architecture** est dimensionnable si la machine continue à fournir les mêmes performances par processeur lorsque l'on accroît le nombre de processeurs.

## 7.6 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

En d'autres mots, les communications deviennent les opérations «barométriques».

## 7.7 Sources des surcoûts d'exécution parallèle

- Création des *threads* et changements de contexte
- Synchronisation et communication entre les *threads*

- Communications inter-processeurs
- Répartition de la charge de travail (*load balancing*)
- Calculs supplémentaires

## 7.8 Lois d'Amdhal et de Gustafson–Barsis

**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*»
- Parmi les modifications, certains éléments de notation ont été simplifiés ou modifiés :

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

### 7.8.1 Temps d'exécution, partie séquentielle, partie parallèle et accélération

Le temps pour un problème de taille $n$ avec $p$ processeurs peut être décomposé en deux parties :

- $\sigma$ = partie intrinsèquement séquentielle
- $\varphi$ = partie pouvant s'exécuter en parallèle

Temps pour un processeur :

$$T(1,n) = \sigma + \varphi \tag{7.1}$$

Temps pour $p$ processeurs :

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

Accélération :

$$\psi \leq \frac{\sigma + \varphi}{\sigma + \varphi/p} \tag{7.2}$$

### 7.8.2 Loi d'Amdahl

$f$ = *fraction* des opérations qui doivent être exécutées de façon séquentielle ($0 \leq f \leq 1$) :

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

On a alors :

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

Et :

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

Donc :

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

**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 $\psi$ pouvant être obtenue avec $p$ processeurs est :*

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

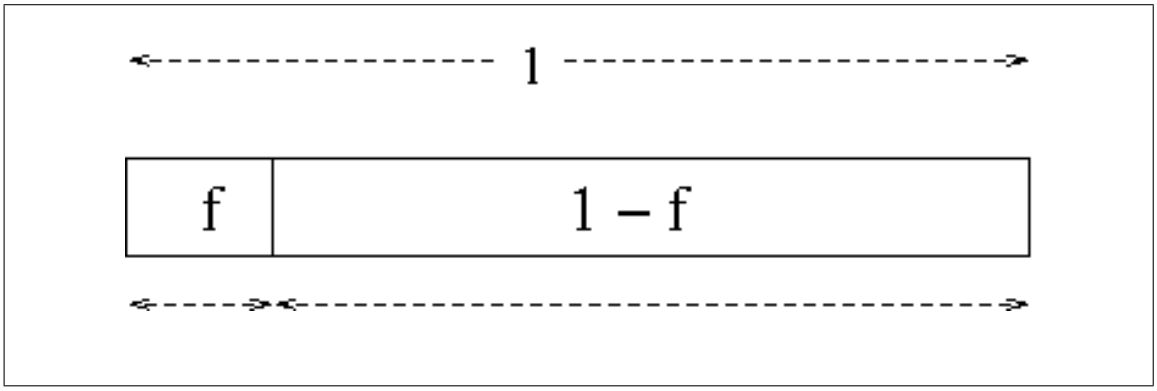



**Figure 7.1:** Illustration graphique de la loi d'Amdahl $\approx$ on **normalise** le temps d'exécution séquentielle (à 1).

Exemples :

- Un programme pour lequel 90 % pourrait s'exécuter en parallèle.

  Accélération maximale avec huit (8) processeurs :

$$\psi \leq \frac{1}{0.1 + (1 - 0.1)/8} \approx 4.7$$

- Un programme pour lequel 75 % pourrait s'exécuter en parallèle.

  Accélération maximale sur une machine parallèle sans limite de processeurs :

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

**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.

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

**L'effet Amdahl**

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

Donc, pour un nombre fixe de processeurs, l'accélération va augmenter lorsqu'on augmente la taille du problème.

En d'autres mots, **plus la taille du problème est grande, plus l'accélération est grande.**

### 7.8.3 Loi de Gustafson–Barsis

La loi d'Amdahl suppose que l'objectif est de **produire le résultat le plus rapidement possible**.

Mais, souvent, l'objectif est plutôt de traiter un problème **de plus grande taille**

Pour la loi de Gustafson–Barsis, on va supposer que **le temps d'exécution parallèle est fixe**.

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*.

| | | |
|---|---|---|
| On ajoute des processeurs | $\Rightarrow$ | On peut traiter des problèmes plus gros |
| | $\Rightarrow$ | La fraction parallélisable augmente |
| | $\Rightarrow$ | La fraction séquentielle diminue |
| | $\Rightarrow$ | L'accélération augmente |

$s$ = fraction du *temps d'exécution parallèle* consacrée aux opérations séquentielles :

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

On a alors :

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

Et :

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

Donc :

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

**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 $\psi$ pouvant être obtenue est la suivante :*

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

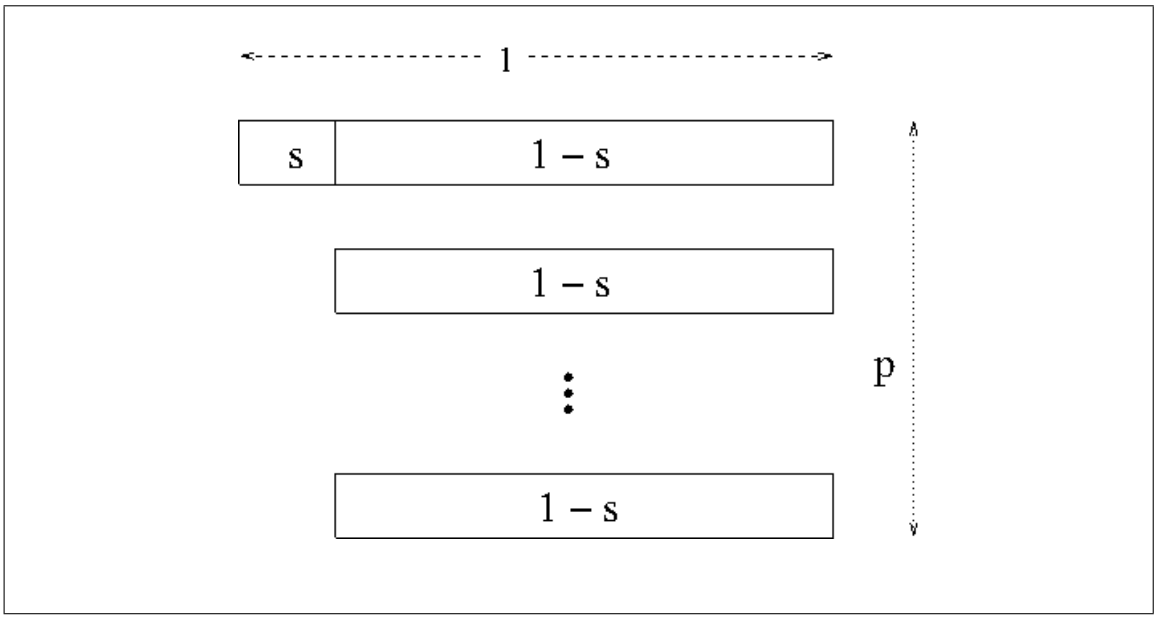



**Figure 7.2:** Illustration graphique de la loi de Gustafson–Barsis $\approx$ on **normalise** le temps d'exécution parallèle.

L'accélération prédite par la loi de Gustafson–Barsis est aussi appelée ***scaled speedup***

Exemple :

- Un programme s'exécute en 220 secondes sur 64 processeurs.

  Des mesures montrent que 5 % du temps d'exécution est pour des parties séquentielles.

  Quelle sera l'accélération (*scaled speedup*) obtenue par ce programme?

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

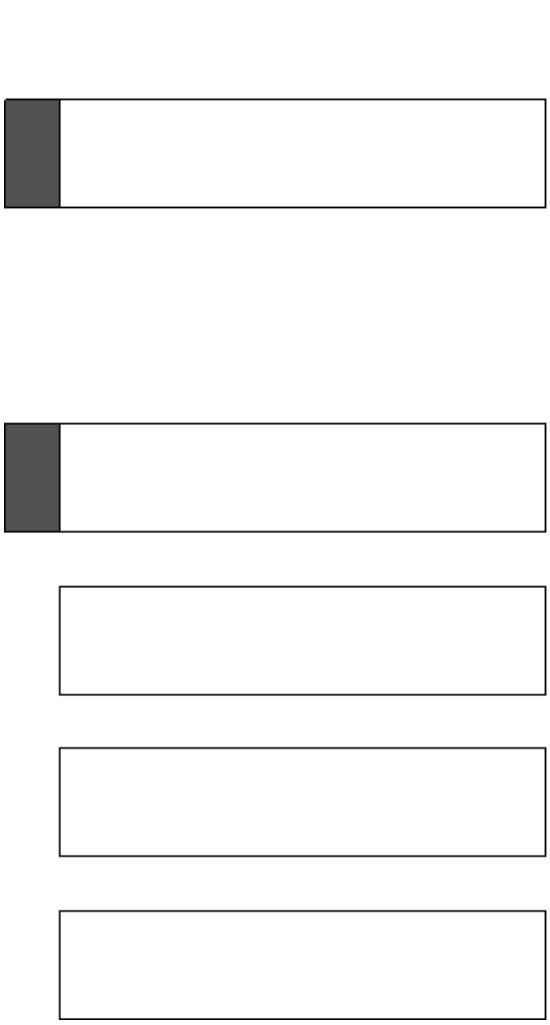
### 7.8.4 Accélération selon la loi d'Amdhal vs. selon la loi de Gustafson-Barsis

| | **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 7.1:** 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

### 7.8.5 Fraction séquentielle déterminée expérimentalement

Karp & Flatt 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 :

$$e = \frac{\sigma}{T(1,n)} \tag{7.3}$$

Alors :

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

$$\begin{aligned} \varphi &= T(1,n) - \sigma && \text{[Déduit de l'équation (7.1)]} \\ &= T(1,n) - eT(1,n) && \text{[Déduit de l'équation (7.3)]} \\ &= (1-e)T(1,n) \end{aligned}$$

Donc :

$$T(p,n) = \sigma + \frac{\varphi}{p} \tag{7.4}$$

$$= eT(1,n) + \frac{(1-e)T(1,n)}{p} \tag{7.5}$$

Soit l'accélération, notée $\psi$, définie comme suit :

$$\psi = \frac{T(1,n)}{T(p,n)} \tag{7.6}$$

Donc :

$$\begin{aligned} T(p,n) &= eT(1,n) + (1-e)T(1,n)/p && \text{[De (7.5)]} \\ &= e\psi T(p,n) + (1-e)\psi T(p,n)/p && \text{[Déduit de (7} \end{aligned}$$

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

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

En divisant par $\psi$ 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 :

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

Exemples :

- 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 déballancement 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.

# 7.A Mesures de performances : Un exemple concret

```
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 7.1** Une fonction parallèle `evaluerPi` (et sa fonction auxiliaire `nbDansCercleSeq`) pour approximer la valeur de $\pi$ 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]; // uncl
   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;
}
```

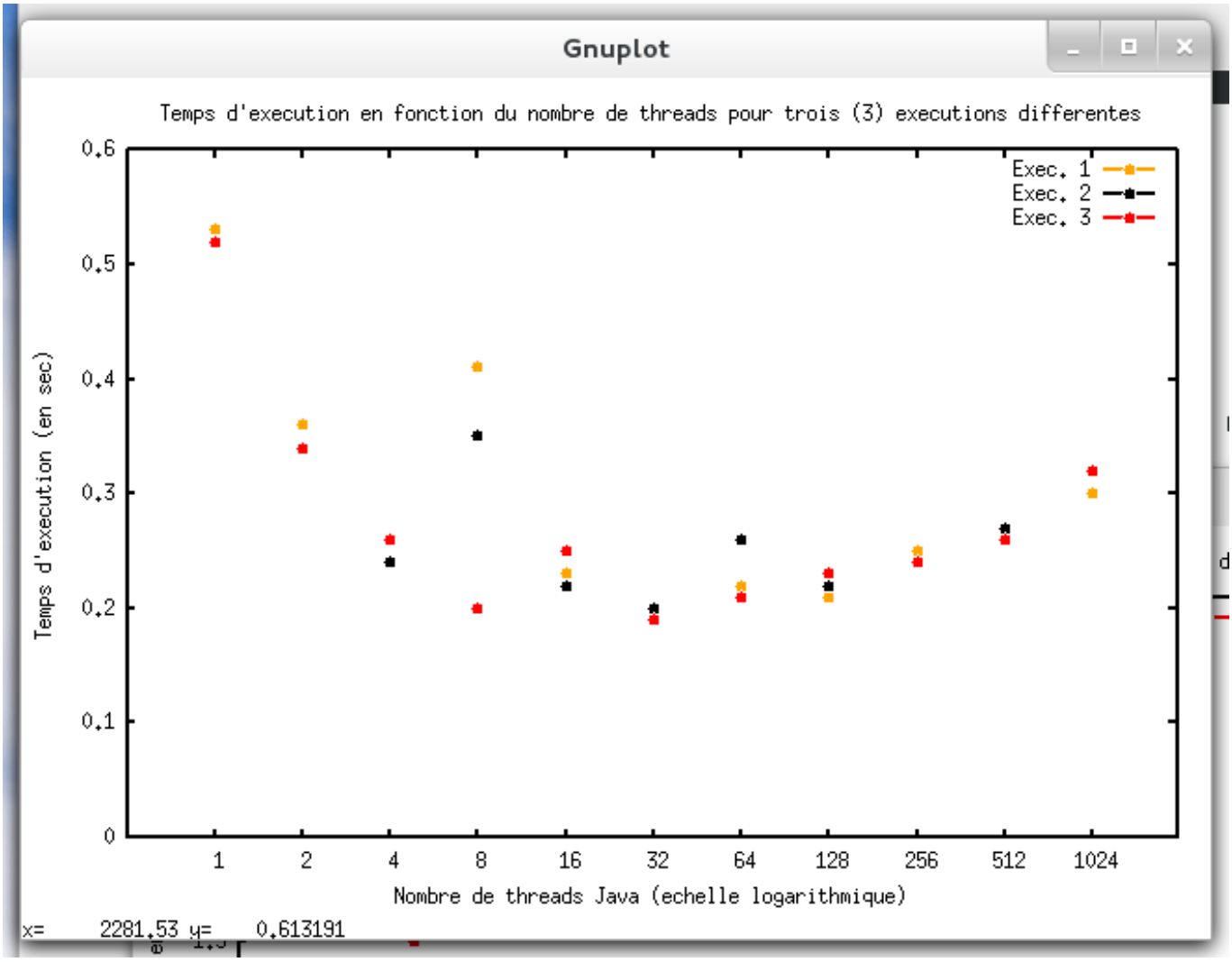



**Figure 7.7:** Graphe donnant le temps d'exécution du programme `Pi.java` pour différents nombres de *threads*. Les temps sont indiqués pour trois (3) séries différentes d'exécution.

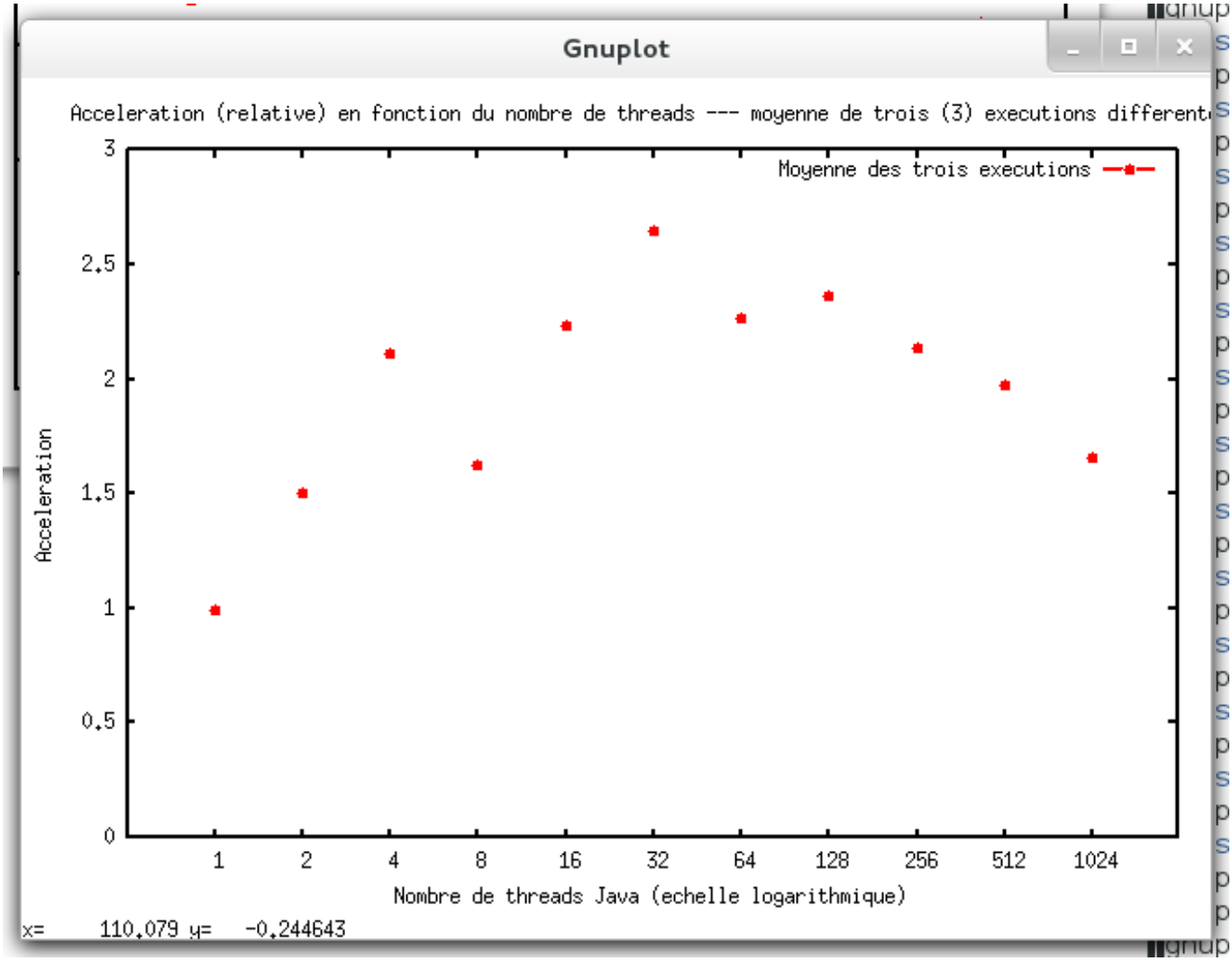



**Figure 7.8:** Graphe d'accélération du programme `Pi.java` pour différents nombres de *threads*— moyenne de trois (3) éxécutions.

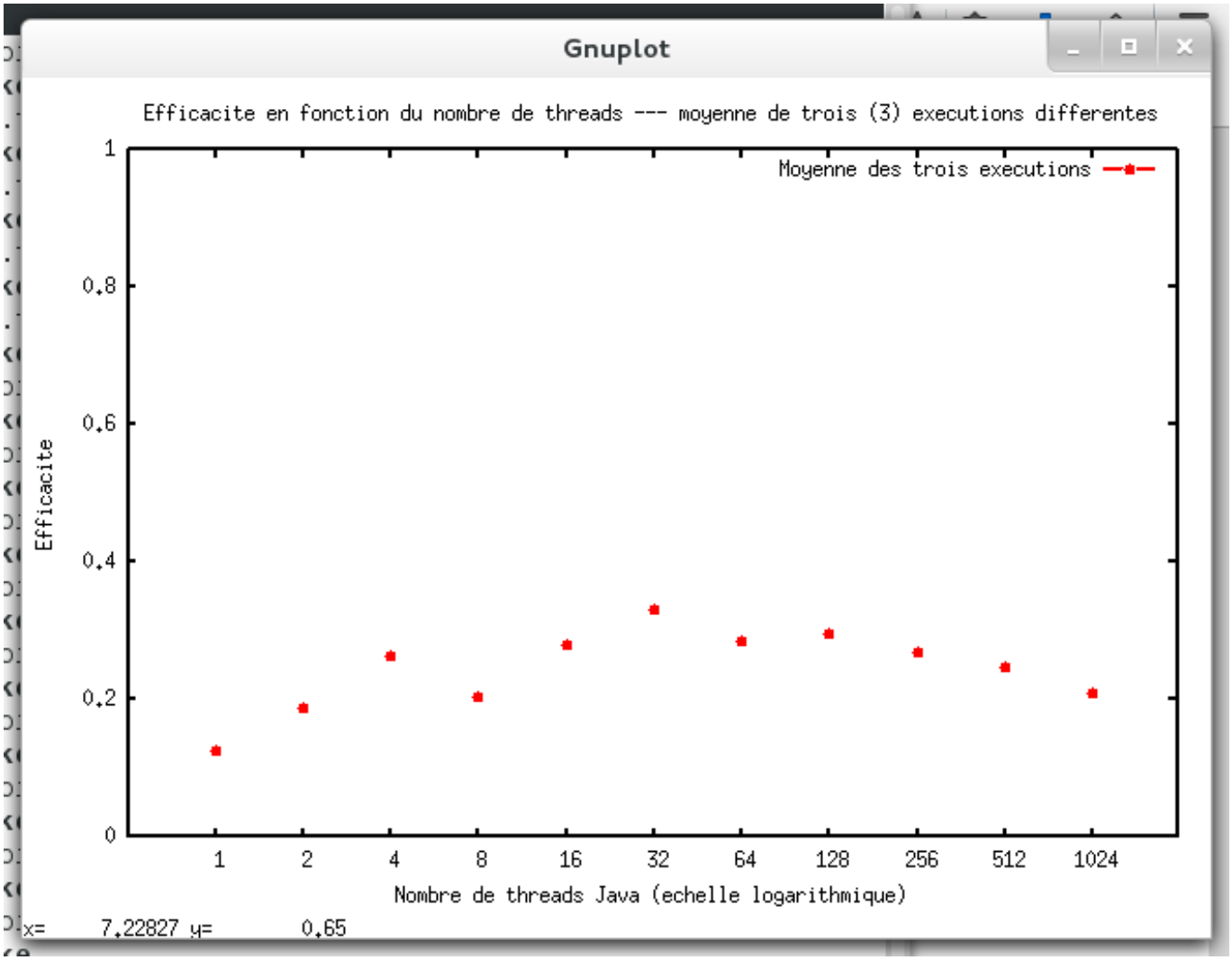



**Figure 7.9:** Graphe d'efficacité du programme `Pi.java` pour différents nombres de *threads*— moyenne de trois (3) éxécutions, pour huit (8) processeurs.