Rapport – Travaux Pratique 3

Sommaire

[Exercice 1 1](#_Toc94459730)

[Exercice 2 2](#_Toc94459731)

[Exercice 3 3](#_Toc94459732)

[Exercice 4 4](#_Toc94459733)

[Exercice 5 5](#_Toc94459734)

# Exercice 1

|  |  |  |
| --- | --- | --- |
| Taille des tableaux | Moyenne du temps d’exécution d’un MAP pour calculer le carré des éléments d’un tableau (µs) | Moyenne du temps d’exécution d’un MAP pour calculer l’addition des éléments de deux tableaux (µs) |
| 1 024 | 13 | 8 |
| 16 384 | 17 | 14 |
| 262 144 | 55 | 45 |
| 4 194 304 | 1 500 | 2 100 |
| 67 108 864 | 27 000 | 35 000 |
| 268 435 456 | 110 000 | 140 000 |

Avec peu de valeurs le temps d’exécution est légèrement plus rapide pour le calcul de l’addition que pour le calcul du carré. Le phénomène inverse se produit quand on augmente les valeurs, c’est alors le calcul du carré qui est plus rapide que l’addition.

Cela peut s’expliquer par le fait que le calcul du carré s’exécute avec le même tableau alors que le calcul de l’addition s’exécute sur deux tableaux différents.

# Exercice 2

|  |  |  |  |
| --- | --- | --- | --- |
| Taille du tableau | Moyenne du temps d’exécution d’un REDUCE pour calculer la somme des éléments d’un tableau (µs) | Moyenne du temps d’exécution d’un REDUCE (*transform* puis *redude*) pour calculer la somme des carrés des éléments d’un tableau (µs) | Moyenne du temps d’exécution d’un REDUCE (*transform\_reduce*) pour calculer la somme des carrés des éléments d’un tableau (µs) |
| 1 024 | 13 | 19 | 8 |
| 16 384 | 17 | 45 | 14 |
| 262 144 | 55 | 350 | 35 |
| 4 194 304 | 875 | 9 700 | 900 |
| 67 108 864 | 15 500 | 142 000 | 15 000 |
| 268 435 456 | 57 000 | 576 000 | 59 000 |

Les temps d’exécution du REDUCE pour calculer la somme des éléments d’un tableau et celui pour calculer la somme des carrées avec un unique *transform\_reduce*, sont à peu près équivalent avec parfois l’un plus rapide que l’autre et inversement.

Pour le REDUCE non optimisé pour calculer la somme des carrés des éléments d’un tableau avec l’utilisation d’un *transform* puis un *reduce*, on observe que le temps d’exécution est 10 fois plus lent que les deux autres REDUCE. C’est normal puisqu’on parcourt deux fois le tableau et qu’on additionne le temps d’exécution d’un MAP pour calculer le carré des éléments d’un tableau et d’un REDUCE pour calculer la somme des éléments d’un tableau.

# Exercice 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taille des tableaux | Moyenne du temps d’exécution d’un MAP avec découpage fixe par bloc pour calculer le carré des éléments d’un tableau (µs) | Moyenne du temps d’exécution d’un MAP avec découpage fixe par bloc pour calculer l’addition des éléments de deux tableaux (µs) | Moyenne du temps d’exécution d’un MAP avec découpage fixe par modulo pour calculer le carré des éléments d’un tableau (µs) | Moyenne du temps d’exécution d’un MAP avec découpage fixe par modulo pour calculer l’addition des éléments de deux tableaux (µs) |
| 1 024 | 1 000 | 1 000 | 1 000 | 1 000 |
| 16 384 | 1 000 | 1 000 | 1 000 | 1 000 |
| 262 144 | 1 000 | 1 000 | 1 200 | 1 200 |
| 4 194 304 | 1 850 | 2 400 | 10 000 | 12 000 |
| 67 108 864 | 27 000 | 35 000 | 250 000 | 350 000 |
| 268 435 456 | 110 000 | 140 000 | 1 100 000 | 1 600 000 |

Avec peu de valeurs le temps d’exécution est significativement plus long avec nos versions de la fonction *transform* par rapport à la version standard de C++.

Lorsque les valeurs augmentent on remarque deux résultats différents selon le découpage qu’on a choisi de faire pour notre version de la fonction *transform* :

* Si le découpage est fixe et effectué par bloc alors on obtient le même temps d’exécution que la fonction de la bibliothèque standard de C++.
* Si le découpage est fixe par modulo le temps d’exécution est 10 fois supérieur à la bibliothèque standard et au découpage fixe par bloc.

On va donc préférer un découpage fixe par bloc pour ces calculs, si on a un grand nombre de valeurs, sinon le mieux reste la fonction *transform* de la librairie standard qui est optimisé pour toute taille de tableau. On peut deviner que cette fonction va automatiquement choisir le nombre de threads à lancer selon la taille du tableau, ainsi si le tableau est petit, la fonction lance un petit nombre de threads et si le tableau est grand alors la fonction lance le maximum de threads disponible.

# Exercice 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Taille des tableaux | Moyenne du temps d’exécution d’un GATHER avec découpage fixe par bloc (µs) | Moyenne du temps d’exécution d’un SCATTER avec découpage fixe par bloc (µs) | Moyenne du temps d’exécution d’un GATHER avec découpage fixe par modulo (µs) | Moyenne du temps d’exécution d’un SCATTER avec découpage fixe par modulo (µs) |
| 1 024 | 1 000 | 1 000 | 1 000 | 1 000 |
| 16 384 | 1 000 | 1 000 | 1 000 | 1 000 |
| 262 144 | 1 000 | 1 000 | 1 200 | 1 200 |
| 4 194 304 | 2 800 | 4 000 | 16 000 | 17 000 |
| 67 108 864 | 42 000 | 61 000 | 400 000 | 450 000 |
| 268 435 456 | 170 000 | 240 000 | 1 800 000 | 2 100 000 |

Avec peu de valeurs le temps d’exécution des patrons GATHER et SCATTER est le même qu’avec le patron MAP.

Quand les valeurs augmentent, on se rend compte que les patrons GATHER et SCATTER ont un temps d’exécution plus long que le patron MAP. On voit également que le patron GATHER s’exécute plus rapidement que le patron SCATTER.

Pour la différence d’exécution entre le découpage fixe par bloc et le découpage fixe par modulo, on constate les mêmes observations que dans l’exercice précédent avec le patron MAP.

# Exercice 5

|  |  |
| --- | --- |
| Taille des tableaux | Moyenne du temps d’exécution d’un REDUCE pour calculer la somme des éléments d’un tableau (µs) |
| 1 024 | 1 000 |
| 16 384 | 1 000 |
| 262 144 | 1 000 |
| 4 194 304 | 1 300 |
| 67 108 864 | 15 500 |
| 268 435 456 | 57 000 |

Encore une fois on remarque que notre version de *reduce* n’est pas adaptée pour les petits tableaux. En revanche pour les grands tableaux le temps d’exécution entre notre *reduce* et celui de la bibliothèque standard en C++, sont équivalent.