Méthodes et Programmation Numériques Avancées

TP1

Produit Scalaire - Produit Matrice Vecteur - Gram-Schmidt

Maxime Kermarquer - M2 CHPS

Table des matières

I Produit scalaire **2**

1) Cas séquentiel **2**

2) Cas parallèle **2**

3) Résultats obtenus **2**

II Produit Matrice Vecteur **3**

4) Cas séquentiel **3**

5) Cas parallèle **4**

6) Résultats obtenus **4**

III Gram-Schmidt **5**

7) Description de la méthode **5**

8) Résultats obtenus **6**

Introduction

Afin de nous familiariser avec les méthodes numériques, il nous a été demandé d'étudier et d'implémenter le produit scalaire, le Produit Matrice Vecteur et l'algorithme de Gram-Schmidt.

Nous allons présenter une analyse des complexités de ces différents problèmes en séquentiel mais aussi en parallèle. Nous présenterons les courbes de performances obtenues, ainsi que nos observations sur celle ci.

Dans ce rapport, nous commencerons par présenter les travaux réalisés sur le produit scalaire, puis nous détaillerons le Produit Matrice Vecteur et la dernière partie sera consacrée à l'algorithme de Gram-Schmidt.

Première partie

Produit scalaire

1) Cas séquentiel

La méthode utilisée pour réaliser le produit scalaire séquentielle est décrite dans la figure 1.

input : $\overrightarrow{a}$ et $\overrightarrow{b}$ deux vecteurs de taille n
output: *resultat* le produit scalaire de $\overrightarrow{a}$ et $\overrightarrow{b}$

$resultat \leftarrow 0$

for $i \leftarrow 0$ **to** n **do**
  $tmp \leftarrow a_i * b_i$;
  $resultat \leftarrow resultat + tmp$;
end

return *resultat*

FIGURE 1 – L'algorithme du produit scalaire en séquentiel

On peut calculer la complexité en temps de la méthode, on a :
— n multiplications
— n additions
Ce qui nous fait une complexité en temps de : $\mathcal{O}(2n)$
Pour la complexité en espace, on a deux vecteurs de taille n : $\mathcal{O}(2n)$

2) Cas parallèle

La méthode utilisée pour réaliser le produit scalaire en parallèle est décrite dans la figure 2.

Pour p processus
input : $\overrightarrow{a}$ et $\overrightarrow{b}$ deux vecteurs de taille n
output: *resultat* le produit scalaire de $\overrightarrow{a}$ et $\overrightarrow{b}$

$resultat \leftarrow 0$;
$resultat.processus \leftarrow 0$;
$k \leftarrow \frac{n}{p}$;

for $i \leftarrow 0$ **to** k **do**
  $tmp \leftarrow a_i * b_i$;
  $resultat.processus \leftarrow resultat.processus + tmp$
end
`Reduction(`*resultat.processus, ADDITION, resultat*`);`

return *resultat*

FIGURE 2 – L'algorithme du produit scalaire en parallèle

Le nombre d'opérations sera divisé par les p processus on a donc un complexité en temps est de : $\mathcal{O}(2n/p)$
Pour la complexité en espace, on a toujours : $\mathcal{O}(2n)$

3) Résultats obtenus

Les tests ont étaient réalisés sur une machine à quatre cœurs. On a fait varier la taille des vecteurs afin d'étudier le temps d'exécution du produit scalaire. Les résultats sont présentés dans la figure 3.

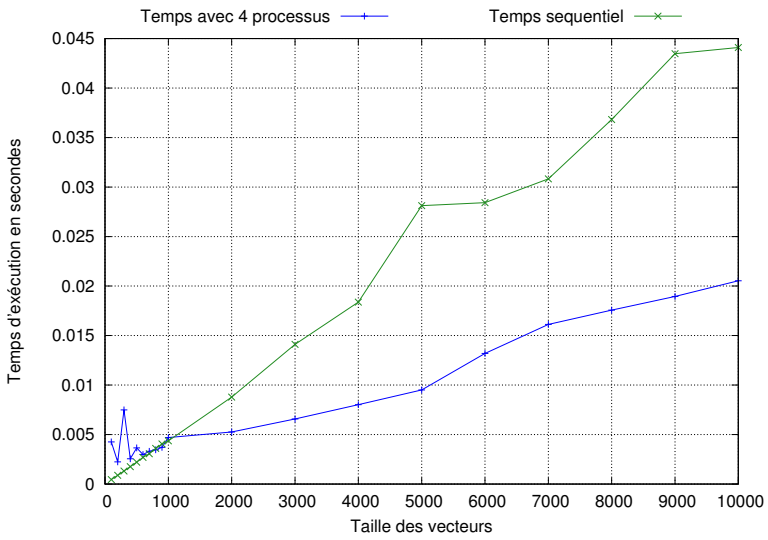

FIGURE 3 – Évolution du temps de calcul du produit scalaire en fonction de la taille des vecteurs

Les deux courbes ont une croissance linéaire. La courbe parallèle a néanmoins une croissance plus faible. On observe que la méthode séquentielle est plus efficace pour les petites tailles de vecteurs comme on peut le voir dans la figure 4. Le coût du parallélisme est alors plus important que son gain. Le parallélisme devient intéressant pour des produits scalaire de taille supérieur à environ 1100.

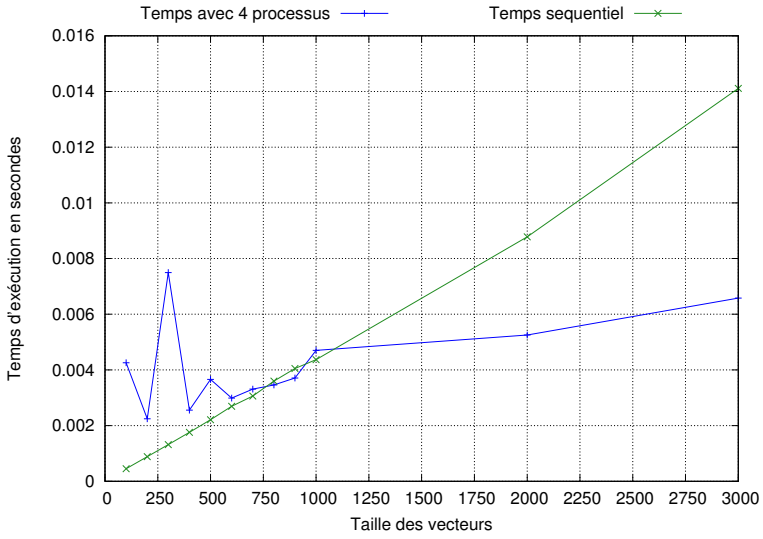

FIGURE 4 – Évolution du temps de calcul du produit scalaire en fonction de la taille des vecteurs

Deuxième partie

Produit Matrice Vecteur

4) Cas séquentiel

La méthode utilisée pour réaliser le Produit Matrice Vecteur séquentiellement est décrite dans la figure 5.

input : A une matrice de taille $m \times n$ et $\overrightarrow{v}$ un vecteur de taille n
output: $\overrightarrow{r}$ le vecteur résultant du Produit Matrice Vecteur de A et $\overrightarrow{v}$ de taille m

for $i \leftarrow 0$ **to** m **do**
  $r_i \leftarrow \langle A[i], v \rangle$;
end

return $\overrightarrow{r}$

FIGURE 5 – L'algorithme du Produit Matrice Vecteur séquentiel

On réalise m produit scalaire, on a donc une complexité de : $\mathcal{O}(2mn)$. Lorsque $m \to n$: $\mathcal{O}(n^2)$.
Pour la complexité en espace, on a :
— Une matrice de $m \times n$
— Un vecteur de taille n
— Un vecteur de taille m
Ce qui nous fait une complexité en espace de : $\mathcal{O}(mn + m + n)$.

5) Cas parallèle

La méthode utilisée pour réaliser le Produit Matrice Vecteur en parallèle est décrite dans la figure 6.

Pour p processus
input : A une matrice de taille $m \times n$ et $\overrightarrow{v}$ un vecteur de taille n
output: $\overrightarrow{r}$ le vecteur résultant du Produit Matrice Vecteur de A et $\overrightarrow{v}$ de taille m

$k \leftarrow \frac{m}{p}$;

`r.processus est de taille k`
$\overrightarrow{r.processus}$;

for $i \leftarrow 0$ **to** k **do**
  $r.processus_i \leftarrow \langle A[i], v \rangle$;
end

`Regroupement des` $\overrightarrow{r.processus}$ `dans` $\overrightarrow{r}$
`Regroupement(`*resultat.processus*`, r);`

return $\overrightarrow{r}$

FIGURE 6 – L'algorithme du Produit Matrice Vecteur en parallèle

Comme pour le produit scalaire, la complexité en temps du Produit Matrice Vecteur sera divisé par p, auquel on ajoute un coût de parallélisation C : $\mathcal{O}(\frac{n^2}{p} + C)$.
La complexité en espace reste la même : $\mathcal{O}(mn + m + n)$.

6) Résultats obtenus

Comme pour le produit scalaire, on a fait varier la taille du vecteur et de la matrice afin d'étudier le temps d'exécution du Produit Matrice Vecteur. Les résultats sont présentés dans la figure 7.

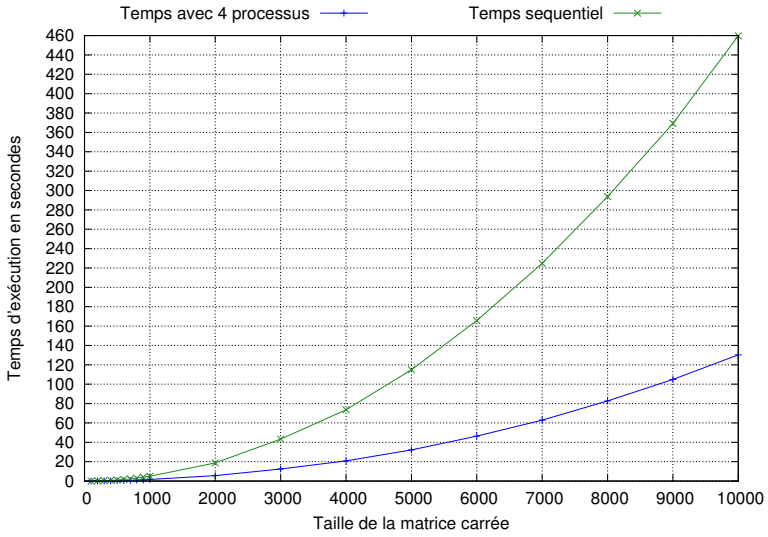

FIGURE 7 – Évolution du temps de calcul Produit Matrice Vecteur en fonction de la taille des données

Bien que l'échantillon de test est assez petit, les courbes ont l'air d'avoir des croissances exponentielles. Ce qui correspond aux complexités théoriques décrites précédemment.

Troisième partie

Gram-Schmidt

Nous nous sommes ensuite concentrés sur l'algorithme de Gram-Schmidt et sa version modifiée.

7) Description de la méthode

L'algorithme va réaliser une orthonormalisation d'un ensemble de vecteurs. La version modifiée permet de corriger les erreurs d'arrondis de la version classique. Les détails de ces algorithmes sont donné dans la figure 8.

$A \in \mathbb{R}^{m \times n}$ avec $m \geq n$, et A a n colonnes linéairement indépendantes $a_1, a_2, \cdots, a_n$.

Classique
for $k \leftarrow 1$ **to** n **do**
  $w \leftarrow a_k$
  for $j \leftarrow 1$ **to** $k-1$ **do**
    $r_{jk} \leftarrow q_j^t w$
  end
  for $j \leftarrow 1$ **to** $k-1$ **do**
    $w \leftarrow w - r_{jk} q_j$
  end
  $r_{kk} \leftarrow ||w||_2$
  $q_k \leftarrow \frac{w}{r_{kk}}$
end

Modifié
for $k \leftarrow 1$ **to** n **do**
  $w \leftarrow a_k$
  for $j \leftarrow 1$ **to** $k-1$ **do**
    $r_{jk} \leftarrow q_j^t w$
    $w \leftarrow w - r_{jk} q_j$
  end
  $r_{kk} \leftarrow ||w||_2$
  $q_k \leftarrow \frac{w}{r_{kk}}$
end

FIGURE 8 – Algorithme de Gram-Schmidt classique et modifié

8) Résultats obtenus

On a augmenter le nombre de vecteurs à orthonormaliser, afin d'observer le comportement des deux version de l'algorithme. Nous pouvons ainsi étudier l'erreur de précision de chacun.

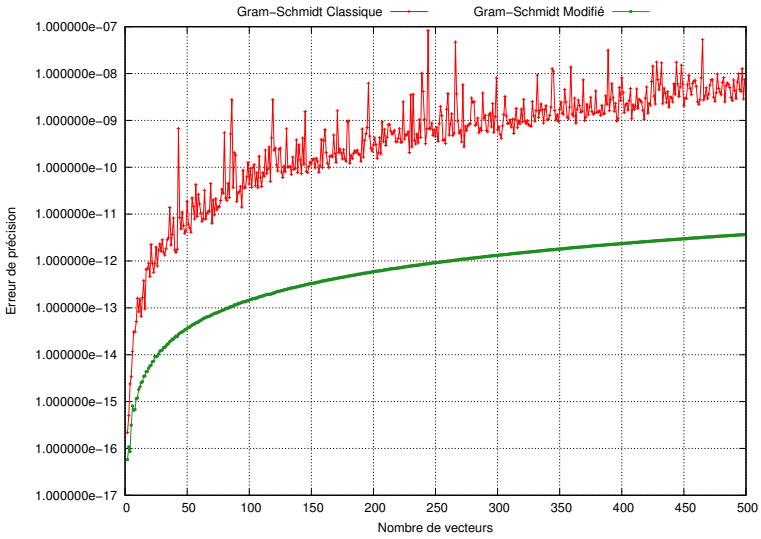

FIGURE 9 – Évolution de l'erreur de précision en fonction de la taille des données

On observe que l'erreur est croissante. La version classique a de nombreux pics dans sa progression, alors que la courbe de la version modifiée est régulière. La version modifiée offre une meilleure précision.

Conclusion

Dans ce premier TP, nous avons été initier aux méthodes numériques, et comment les adapter pour des architectures parallèles. Nous avons étudier leurs complexités en temps, espace et précision. Avec Gram-Schmidt, nous avons pus voir que certaines optimisations anodines permettent un réel gain de complexité.

L'apprentissage de cette approche, des métriques de performances nous sera utile pour les prochaines études des méthodes itératives.