RAPPORT FINAL
CALCUL SCIENTIFIQUE PARALLÈLE

Sujet: Application du calcul parallèle au produit matrice-matrice dans l'algorithme PageRank

VUONG Thi Anh Tuyet

Table des matières

Chapitre 1

Introduction

1.1 PageRank de Google

L'une des raisons pour laquelle le moteur de recherche de Google est si efficace est qu'il classe les pages selon un score d'importance. Ce score est calculé par l'algorithme PageRank, il représente la popularité d'une page web. Il est recalculé une fois par mois et déterminé complètement par la structure des liens entre les pages du Net mais pas leur contenu.Le principe est qu'une page web est autant importante qu'il y a plusieurs pages importantes pointant vers lui. Plus la valeur du PageRank (score d'importance) est élevée, plus la page en question est populaire.

Soit W l'ensemble de site web qui peut être atteint par une chaîne de liens à partir d'une certaine racine, et n le nombre de site web dans W. G est la matrice de connectivité de W, G est une matrice carrée de taille n, les coefficients de G sont déterminés par :

$$g_{ij} = 1 \text{ s'il y a un lien du page j vers la page i}$$
$$g_{ij} = 0 \text{ sinon}$$

La matrice G est très grande mais elle est creuse. La j-ième colonne liste les liens dans la page j et le nombre de coefficients non nuls dans G est le nombre de liens dans W. Si on note c_j la somme des coefficients sur la colonne j, c_j est est le degré sortant de la page j. c_j peut-être égale à 0, dans ce cas la page j est une impasse qui ne contient aucun lien externe. En arrivant à la page j, le surfeur va aller vers une page aléatoire qui n'a rien à voir ou très peu avec la page j. Donc pour décrire le comportement du surfeur, on autorise un saut vers une page aléatoire avec un probabilité très faible et on modélise par p : la probabilité que le surfeur suit un lien pour aller à la page suivant. La valeur typique de p est 0.85. La probabilité que le surfeur va vers une page aléatoire est $1 - p$ et la probabilité qu'il va vers une page spécifique dans le Net est $\delta = (1 - p)/n$

A partir de ces définitions, on a la matrice de liens A déterminée par :

$$a_{ij} = pg_{ij}/c_j + \delta \text{ si } c_j \neq 0$$
$$a_{ij} = 1/n \text{ si } c_j = 0$$

Remarquons que la j-ième colonne de la matrice A représente la probabilité d'aller de la page j vers d'autres pages dans le Net. Si la page j est une impasse,

alors on répartit la probabilité $1/n$ à toutes les pages du Net. La plupart des coefficients de A est égale à δ (très petit), c'est-à-dire la probabilité aller vers une page sans passer par un lien. On remarque aussi que les coefficients de A sont strictement positif et plus petit que 1, la somme des coefficients sur une colonne est 1. Pour ce type de matrice, on peut appliquer le théorème de Perron-Frobenius et constater que l'équation $x = Ax$ admet une unique solution de norme 1. Les coefficients de cette solution sont les valeurs de PageRank comme calculé par Google.

1.2 Calcul du score en parallèle par la méthode des puissances itérées

Pour calculer la valeur du PageRank, il s'agit donc de chercher la solution de l'équation $Ax = x$. Une méthode largement utilisée face à ce type de problème avec la dimension de A très grande est la méthode des puissances itérées. Elle consiste à répéter la relation de récurrence

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}$$

à partir d'un vecteur b_0 de norme 1 jusqu'à ce qu'on obtienne le vecteur b dans une tolérance souhaitée. Si on veut contrôler l'incertitude de b par le nombre d'itération, la relation suivante est utilisée :

$$b_k = \frac{A^k b_0}{\|A^k b_0\|}$$

La seule difficulté ici est de calculer A^N avec A très large (la dimension peut monter jusqu'à quelques billions et ne cesse pas de croît). Afin d'avoir un temps de calcul correct, la mise en place d'un algorithme de calcule en parallèle est nécessaire.

Chapitre 2

Calculer la puissance N d'une matrice en parallèle

2.1 Algorithme de base

Le problème de calculer la puissance N d'une matrice se généralise au problème de calcul du produit de deux matrices carrées. Une fois que l'algorithme parallèle pour obtenir $C = A * B$ (avec A et B deux matrices carrées de taille n) s'est mis en place, il suffit de faire B = A puis répéter l'algorithme N fois pour obtenir A^N.

La formule usuelle de calcul matricielle est :

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

Avec $C_{i,j}$ est le coefficient situé sur la ligne i et sur la colonne j de la matrice C. Il faut donc 3 boucles sur n pour calculer la totalité des coefficients de C :

```
for i from 1 to n do
  for j from 1 to n do
    for k from 1 to n do
      C[i][j] = C[i][j] +  A[i][k]*B[k][j]
    end for
  end for
end for
```

Remarquons que cet algorithme calcule en fait C = C + A*B. Afin d'obtenir C = A*B, il suffit d'initialiser les coefficients de C à 0.

2.2 Analyse de la dépendance entre les itérations

Dans la boucle k, considérons deux itérations k_1 et k_2 :

Il y a une dépendance de sortie car $OUT(k_1) = OUT(k_2)$: deux itérations écrivent sur la même variable $C[i][j]$

Il y a une dépendance de données car $OUT(k_1) \cap IN(k_2) = \{C[i][j]\}$

Donc on ne peux pas paralléliser la boucle k.

Itération $(i, j) : C[i][j] = C[i][j] + A[i][k] * B[k][j]$

Itération $(i \pm k_1, j \mp k_2) : C[i \pm k_1][j \mp k_2] = C[i \pm k_1][j \mp k_2] + A[i \pm k_1][k] * B[k][j \mp k_2]$

Entre la boucle i et j , il n'y a pas de dépendances de données ou de sorties entre les itérations (i,j) et $(i \pm k_1, j \mp k_2)$ (avec k_1, k_2 strictement positifs) donc on peut permuter les deux boucles. De même, on peut aussi permuter les boucles k et i ou k et j. La parallélisation est possible sur la boucle i ou j.

2.3 Algorithme par lignes ou par colonnes et ses désavantages

Si on parallélise la boucle i (resp.boucle j), il s'agit d'un algorithme par lignes (resp. par colonnes). Chaque processeur possède donc un certain nombre de lignes de deux matrices globales A et B, puis en échangeant ses coefficients avec les autres, il effectue le calcul pour obtenir un résultat local (les lignes correspondantes de la matrice du résultat C). Considérons d'abord un premier algorithme pour un processus p où la boucle k est à l'intérieur de la boucle j :

```
for i from i1 to i2 do
  for j from 1 to n do
    for k from 1 to n do
      C[i][j] = C[i][j] +  A[i][k]*B[k][j]
    end for
  end for
end for
```

A chaque itération k, le processus accès à la mémoire pour récupérer deux coefficients $A[i][k]$ et $B[k][j]$ donc en total 2n accès. À chaque itération j , en plus de 2n accès évoqués par la boucle k, le processus doit récupérer le coefficient $C[i][j]$ tout au début du calcul. En fin il lui reste une dernière écriture sur la mémoire pour le résultat final de $C[i][j]$. Donc, $2n + 2$ accès en mémoire est effectué à la fin de chaque boucle j.

$$C[i][j] = C[i][j] + A[i][k] * B[k][j]$$

$$1write - 1read - - - 2nread$$

Dans le deuxième cas où la boucle j est à l'intérieur de la boucle k :

```
for i from i1 to i2 do
  for k from 1 to n do
    for j from 1 to n do
      C[i][j] = C[i][j] +  A[i][k]*B[k][j]
    end for
  end for
end for
```

A chaque itération j, le processeur accès 2n fois à la mémoire pour lire la valeur de $C[i][j]$ et $B[k][j]$ mais aussi n fois pour écrire la nouvelle valeur de $C[i][j]$. A chaque itération k, le processeur doit aussi récupérer une fois la valeur de $A[i][k]$, donc $3n + 1$ accès en total.

$$C[i][j] = C[i][j] + A[i][k] * B[k][j]$$
$$nwrite - nread - 1read - nread$$

D'un premier coup d'oeil, le premier l'algorithme semble être le meilleur choix mais en réalité, il faut prendre en compte aussi le temps d'accès. Dans le premier algorithme, il y a les accès par ligne et par colonne (dans la boucle k) tant dis qu'il y a que les accès par colonne dans le deuxième algorithme. Si les données sont rangées par colonne (elles sont sur les cases contiguës dans la mémoire) le deuxième algorithme est plus avantageux car ses accès à la mémoire sont plus rapide que ceux du premier.

Le même raisonnement est valable pour un algorithme par colonne où la boucle j est à l'extérieur, on se retrouve dans les deux sous-cas qui ne sont pas optimaux.

2.4 Algorithme par blocs

Dans les deux cas précédents, seul le coefficient $C[i][j]$ ou $A[i][k]$ est conservé dans le registre du processeur, les autres coefficients sont relu/réécrit à chaque itération. Ce n'est pas tout à fait le cas en pratique. En effet, chaque processeur procède une mémoire cache près de lui afin de stocker temporairement les données dont il a besoin après les avoir récupérées de la mémoire centrale. Tant que les matrices sont de dimension assez petite, leurs coefficients sont enregistrés dans la mémoire cache et l'accès aux données dans ce cas est beaucoup plus rapide. Il est donc fortement plus avantageux que chaque processeur manipule une matrice locale de petite taille afin qu'il puisse stocker les coefficients dans sa mémoire cache : il s'agit d'un algorithme par blocs.

Dans ce type d'algorithme, les matrices A et B (de dimension n) sont divisées en $p = q^2$ matrices carrées. Supposons que q divise n, la dimension de chaque matrice est n/q. (Si ce n'est pas le cas, quelques solutions sera proposées dans la partie 3).

$$\begin{pmatrix} A_{11} & ... & A_{1J} & ... & A_{1q} \\ ... & ... & ... & ... & ... \\ A_{I1} & ... & A_{IJ} & ... & A_{Iq} \\ ... & ... & ... & ... & ... \\ A_{q1} & ... & A_{qJ} & ... & A_{qq} \end{pmatrix}$$
$$\begin{pmatrix} B_{11} & ... & B_{1J} & ... & B_{1q} \\ ... & ... & ... & ... & ... \\ B_{I1} & ... & B_{IJ} & ... & B_{Iq} \\ ... & ... & ... & ... & ... \\ B_{q1} & ... & B_{qJ} & ... & B_{qq} \end{pmatrix}$$

L'algorithme par bloc est :

```
for I from 1 to q do
  for J from 1 to q do
    for K from 1 to q do
      C[I][J] = C[I][J] +  A[I][K]*B[K][J];
    end for
  end for
```

```
end for
```

où $C[I][J] = C[I][J] + A[I][K] * B[K][J]$ est un calcul matriciel usuel réalisé avec un algorithme par ligne ou par colonne mentionné auparavant. Ce calcul ne pose pas de problème en terme d'optimalité parce que les matrices $A[I][K]$ et $B[I][J]$ sont de taille assez petite pour être stockées dans la mémoire cache du processeur.

Les coefficients de la matrice extraite peut être déduits du coefficient de la matrice globale grâce à la formule suivante :

$$A_{IJ}[i][j] = A[(I-1) * n/q + i][(I-1) * n/q + i]$$

En calcul parallèle, chaque processeur est assigné à un couple d'indice (I,J)(la position d'un bloc dans les matrices globales A,B). Ce couple (I,J) peut être déterminé grâce à son rang *rank* :

$$rank = (J-1) * q + (I-1)$$

ou bien :

$$I = rank\%q + 1$$

$$J = rank/q + 1$$

Chaque processeur possède donc un bloc d'indice (I,J) de la matrice A et un bloc de même indice de la matrice B. L'algorithme pour chaque processeur est réduit à

```
for K from 1 to q do
  C[I][J] = C[I][J] +  A[I][K]*B[K][J];
end for
```

À la fin de cet algorithme, le processus retourne le bloc C_{IJ} de la matrice du résultat. Pendant ses calculs, il fait appel aux blocs A[I][K] et B[K][J] qu'il ne possède pas. Remarquons que les blocs A[I][K] sont sur la même ligne et les blocs B[K][J] sont sur la même colonne que sa position dans la matrice des blocs. Il suffit donc pour le processus (I,J) de récupérer tous les blocs de la matrice A qui sont sur la même ligne que lui et tous les blocs de la matrice B qui sont sur la même colonne que lui.

2.5 Implémentation avec MPI

Les explications sur le déroulement du programme se trouve dans le fichier code sous forme commentaire et dans un fichier README dans le même dossier que les fichiers de code. Je vous présente ci-dessous quelques points généraux et des remarques supplémentaires sur le programme.

2.5.1 Structure générale du programme

Le programme Produit_2Matrices_Parallèle.c Ce programme calcule le produit de 2 matrices carrées **m1** et **m2** en parallèle.Dans le cadre de ce projet, les blocs des matrices **m1** et **m2** ne sont pas disponibles sur les processeurs comme en pratique. Il est nécessaire d'avoir un processeur maître (dans mon

programme c'est le processus de rang 0) qui crée les matrices globales **m1** et **m2**, divise ces matrices en blocs puis envoie les blocs aux autres processeurs. C'est aussi lui qui récupère tous les résultats locaux et les combiner pour reconstruire un résultat global. Après avoir envoyé les blocs aux autres processeurs, il effectue lui même un calcul sur les blocs d'indice (1,1).

Les matrices **m1** et **m2** sont déclarées et initialisées directement par le processus 0 avec la taille n prédéfinie. Actuellement la taille n est fixée à 12, mais il peut être changé à tout moment

2.5.2 Allocation de la mémoire

Pour des raisons pratiques, deux fonctions permettant d'allouer un tableau et de libérer la mémoire allouée à un tableau sont écrites dans mon programme. À noter que cette fonction d'allocation est spéciale pour allouer un tableau sur une mémoire contiguë. Une première tentation d'allouer la mémoire avec un tableau des pointeurs a toujours donné "segmentation fault" donc a été remplacée par cette solution.

2.5.3 Commutateur par ligne ou par colonne. Broadcasting.

À la fin de la partie 2.4 nous avons vu que chaque processus doit récupérer les blocs de la même ligne ou colonne que lui. Ceci est possible grâce à la fonction *MPI_Comm_Split(commutateur_initial , couleur , critère , commutateur_final)* de MPI. Tous les processus du *commutateur_initial* doit appeler cette fonction et il se retrouvera ensuite dans le *commutateur_final* avec tous les processus ayant la même *couleur* que lui. De plus, les processus dans le *commutateur_final* sont attribués leur nouveau rang basé sur la valeur croissante du *critère*. Donc pour créer un commutateur par ligne, il suffit de mettre I comme *couleur* et J comme *critère* :
MPI_Comm_Split(MPI_COMM_WORLD , I , J , commutateur_ligne) ;

De même pour le commutateur par colonne :
MPI_Comm_Split(MPI_COMM_WORLD , J , I , commutateur_colonne) ;

Un processus d'indice (I_0, J_0) a le rang $J_0 - 1$ dans le commutateur par ligne et le rang $I_0 - 1$ dans le commutateur par colonne.

Pour recevoir les blocs provenant des autres processus mais aussi envoyer ses matrices locales à tout le monde dans un commutateur, chaque processus utilise la même fonction *B_cast(B_temps, longueur, type, K, commutateur)*. Il envoie les données dans *B_temps* si son rang dans le commutateur est égale à K, et dans les autres cas il reçoit les données puis les stocke dans *B_temps*.

2.6 Application au calcul du score PageRank

Après avoir implémenter l'algorithme parallèle pour calculer le produit de deux matrices, il est assez direct d'en déduire l'algorithme pour calculer la puissance N d'une matrice A. Le principe est qu'à chaque itération, on remplace l'une des deux matrices de départ par le résultat de l'itération précédente :

```
m1 = m2 = A
for m from 1 to N do
  for I from 1 to q do
    for J from 1 to q do
      for K from 1 to q do
         res[I][J] = res[I][J] +  m1[I][K]*m2[K][J];
      end for
    end for
  end for
  m1 = res
end for
```

Au début du programme, chaque processus possède 2 matrices $m1\_local$ et $m2\_local$ égales à A[I][J]. Après que le calcul en parallèle soit effectué, chaque processus obtient un res_local correspond à $A[I][J]^2$. La matrice $m1\_local$ est ensuite remplacée par ce résultat local et la procédure de calcul en parallèle recommence. Cette procédure est répété N fois et à chaque fois $m1\_local = A[I][J]^k$ est remplacé par $res\_local = A[I][J]^{k+1}$. On obtient à la fin $res\_local = A[I][J]^N$ Ces résultats locaux sont récupérés par le processus 0 pour reconstruit A^N

Les coefficients du vecteur propre b recherché sont initialisés à 1/n et il est calculé par la formule évoqué plus haut dans la partie 1 :

$$b_N = \frac{A^N b_0}{\|A^N b_0\|}$$

Chapitre 3

Perspectives d'amélioration du programme

3.1 Sur la dimension de la matrice à computer

Rappelons que le programme pour ce projet ne fonctionne que dans le cas le nombre de processus est $p = q^2$ avec q un entier et q divise n pour avoir un calcul correct de la dimension des blocs. Dans le cas q ne divise pas n, on peut considérer une des solutions suivantes :

- Parmi les processeurs, un groupe calculera les matrices non-carrées de taille plus petite que les autres. Cette solution est possible à implémentée mais n'est pas très pratique : elle détruit l'uniformité du programme et donc rendre difficile voire impossible la communication interne dans les communicateurs ligne ou colonne. De plus, cette solution n'améliore pas le temps de calcul global. En effet, si tous les processus ont la même vitesse de calcul alors ils mettent un même temps pour effectuer leur calcul local. Le fait que certains processus calculent les matrices plus petites ne rapporte rien parce que le processus 0 doit quand même attendre les processus normaux pour combiner le résultat.

- Compléter la matrice de départ par des lignes et des colonnes contenant que des zéros afin d'obtenir une matrice globale de dimension n divisible par q. Comme expliqué ci-dessus, le fait d'ajouter les lignes ne modifie pas le temps de calcul global parce que c'est toujours le temps d'un processus, à condition que tous les processus font la même chose. Le programme est la même pour tous les processus et plus facile à contrôler. Cette solution semble raisonnable.

3.2 Sur la gestion de la mémoire

Comme la matrice A est utilisée sous forme bloc, le mieux c'est d'allouer la mémoire pour lui sous forme bloc aussi. Cela est possible en utilisant les tableaux à 4 dimensions A[i][j][I][J].

Chapitre 4

Bibliographie

Frédéric Magoulès, François-Xavier Roux, *Calcul scientifique parallèle.* Editeur : Dunod, Paris 2013

Moler, C.*Experiments with MATLAB, Chapter 7 : Google PageRank*, MathWorks, Inc., 2011
(https ://www.mathworks.fr/moler/exm/chapters/pagerank.pdf)