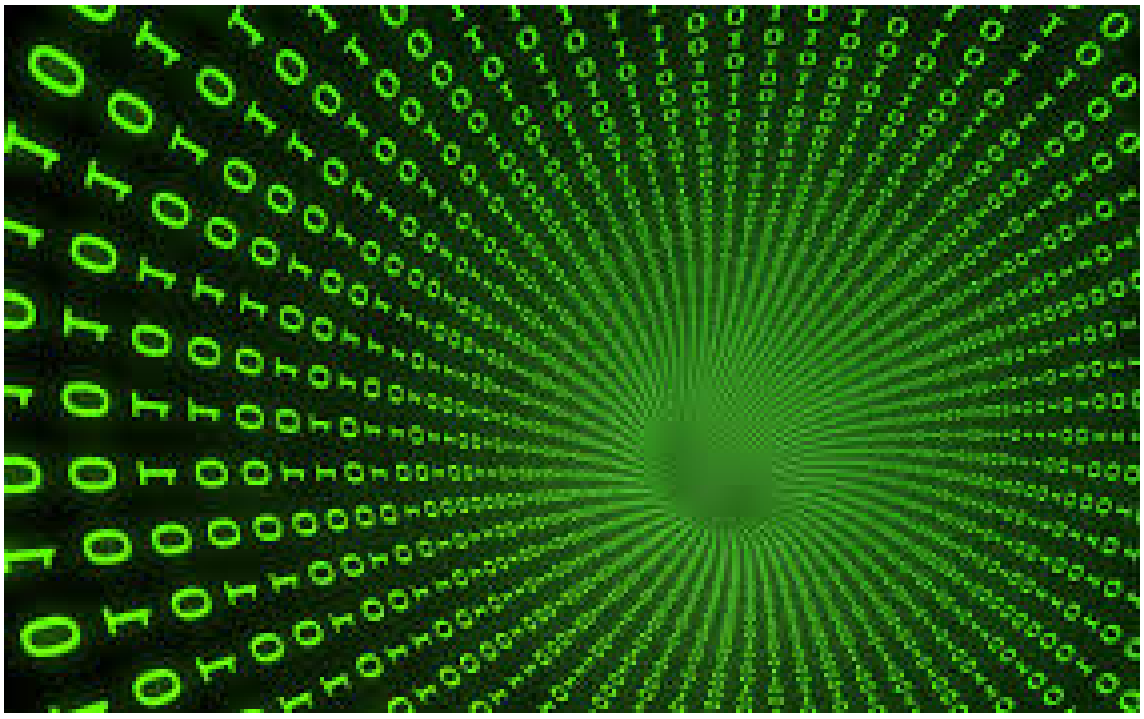
Université Catholique de Louvain

LINGI1113

Systèmes informatiques 2

Projet 1 : Multiplication de matrices creuses

Rapport



Peschke Lena 5826-11-00
Sedda Mélanie 2246-11-00

Professeur : Marc Lobelle

6 octobre 2013

Table des matières

1 Fonctionnement **1**
1.1 Implémentation de base d'une matrice creuse 1
1.2 Multiplication et multithreading 1

2 Mesures de performance et interprétations **2**
2.1 Scénario 1 : variation du nombre de zéros 2

1 Fonctionnement

1.1 Implémentation de base d'une matrice creuse

Nous avons choisi d'implémenter une matrice creuse à l'aide d'une structure de données contenant son nombre de lignes n, son nombre de colonnes m et un tableau de files simplement chaînées pointers. Dans ce tableau, la $i^{ème}$ file représente la $i^{ème}$ ligne de notre matrice et chaque noeud de la file représente un élément non-nul de cette ligne. Dans chaque noeud, nous stockons la valeur v de l'élément (qui sont certes tous des 1 au début comme stipulé dans l'énoncé mais peuvent être différents de 1 une fois qu'on commence à multiplier des matrices), la position j de l'élément au sein de la ligne et un pointeur vers le noeud suivant.

Soit par exemple la matrice

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

On a alors $n = 4$, $m = 4$ et pointers est représenté à la Figure 2.

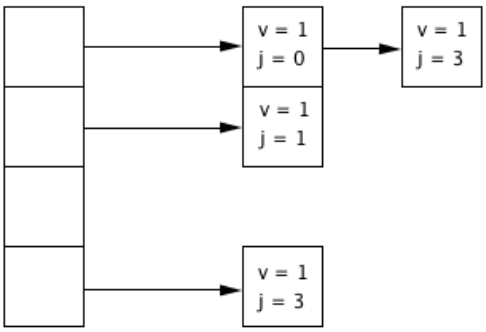


FIGURE 1 – Encodage d'une matrice creuse par lignes

1.2 Multiplication et multithreading

Nous nous sommes ensuite rendu compte que nous pouvions rendre la multiplication de deux matrices creuses A et B plus aisée en encodant A comme présenté ci-dessus mais en encodant B à l'aide de la même structure de données mais en colonnes, c'est-à-dire que la $i^{ème}$ file du tableau représente la $i^{ème}$ colonne de la matrice et plus la $i^{ème}$ ligne. Nous avons donc rajouté un booléen dans notre structure de matrice creuse indiquant si notre matrice est encodée en lignes ou en colonnes. Notre matrice exemple serait alors encodée comme représente à la Figure **??**.

Pour effectuer le produit d'un certain nombre de matrices creuses, c'est-à-dire calculer

$$X = M_1 M_2 M_3 M_4 ... M_N$$

nous avons encodé notre matrice M_1 en lignes et toutes les autres en colonnes. Nous effectuons alors le produit de $(M_1 M_2)$ que nous encodons en lignes, puis le multiplions par M_3, etc.

Nous n'avons pas multithreadé notre programme mais si nous l'avions fait, nous aurions effectué le plus de produit du type $(M_1 M_2)$, $(M_3 M_4)$, $(M_5 M_6)$, ... en même temps en encodant

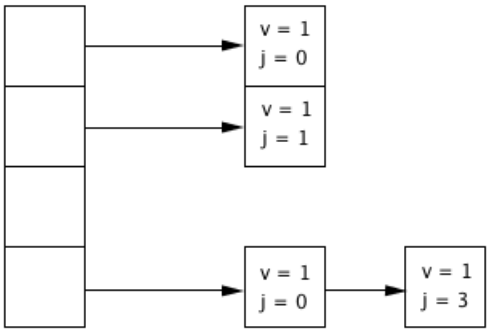


FIGURE 2 – Encodage d'une matrice creuse par colonnes

les matrices d'indices pairs en colonnes. Le résultat de (M_1M_2) serait alors encodé en lignes, (M_3M_4) en colonnes, etc. Dans cette optique, nous avons permis de stipuler à la fonction effectuant le produit de deux matrices creuses si nous voulons que le résultat du produit soit encodé en lignes ou en colonnes mais comme nous n'avons finalement pas multithreadé notre programme, nous l'utilisons exclusivement pour rendre un résultat en lignes.

2 Mesures de performance et interprétations

Il est intéressant d'analyser les performances de notre programme en comparaison avec une implémentation triviale des matrices pleines afin de voir dans quel cas il est intéressant de travailler avec des matrices creuses. Nous avons donc créé un deuxième programme qui effectue toutes les mêmes opérations mais en utilisant de simples tableaux à deux dimensions, et avons comparé les temps d'exécution des deux programmes sur un de nos ordinateurs personnels.

2.1 Scénario 1 : variation du nombre d'éléments non nuls

Nous avons tout d'abord voulu analyser l'influence du nombre de zéros sur le temps d'exécution. Nous avons pour cela généré des fichiers contenant deux fois une même matrice 1000x1000 avec respectivement 1, 3, 9, 27, 81 ou 243 nombres non nuls par ligne (des 1 en l'occurrence) par lignes, décalés de un vers la droite d'une ligne à l'autre pour éviter d'avoir des colonnes vides. Autrement dit, notre premier fichier contenait deux matrices identité de taille 1000, notre deuxième fichier deux matrices de taille 1000 aussi mais avec 3 un sur chaque ligne, etc. Nous avons ensuite calculé le temps d'exécution de nos deux programmes pour lire les deux matrices d'un fichier, les multiplier, et écrire le résultat dans un fichier de sortie.

Mesures Afin d'éviter d'avoir des valeurs aberrantes, nous avons effectué 3 mesures dans chacune des situations.

	1	3	9	27	81	243
creuses	0.257357 s	0.312883 s	0.646042 s	2.996151 s	43.934574 s	226.217773 s
	0.262014 s	0.304584 s	0.651323 s	3.018808 s	43.854031 s	229.002350 s
	0.264037 s	0.310822 s	0.639835 s	2.994909 s	43.921665 s	227.610432 s
pleines	17.634382 s	18.294992 s	18.809765 s	19.883987 s	18.096115 s	18.203505 s
	18.027992 s	20.234669 s	18.161089 s	20.196924 s	18.096998 s	18.051762 s
	21.453005 s	18.225401 s	18.242275 s	18.125992 s	18.777365 s	20.237141 s

Graphe Nous avons ensuite tracé un graphe logarithmique de ces résultats en utilisant les moyennes de ces 3 mesures dans chacun des cas.

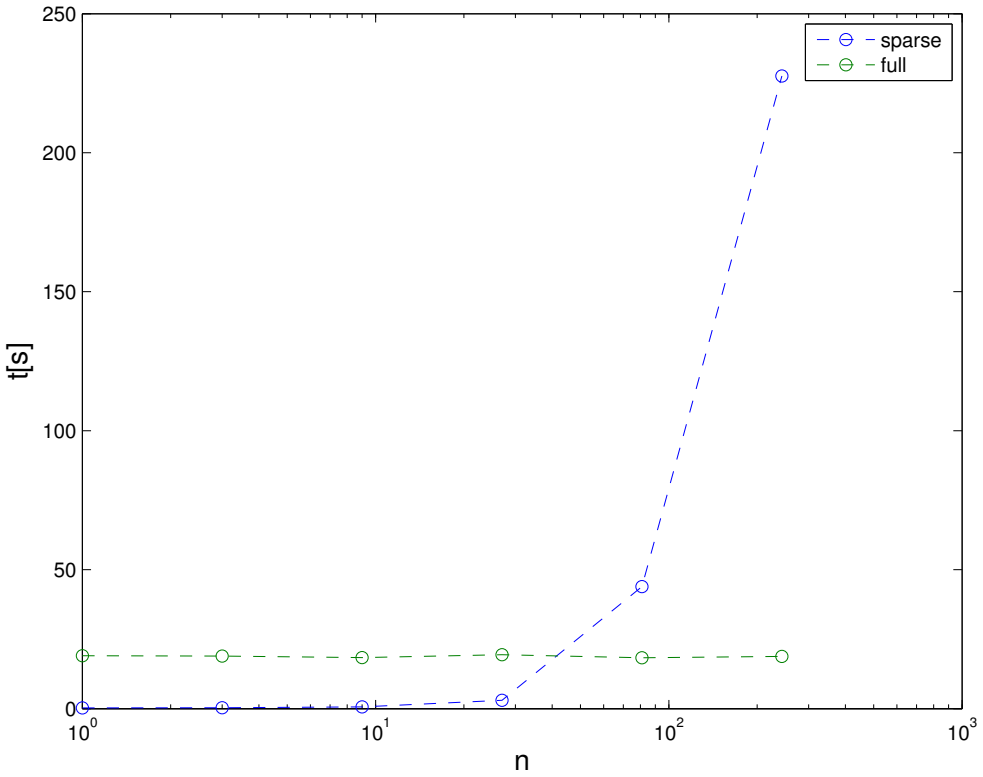


FIGURE 3 – Scénario 1 : variation du nombre d'éléments non nuls

Observations On remarque tout d'abord que pour des matrices pleines, le temps d'exécution ne dépend pas du nombre d'éléments non nuls dans la matrice. Cela est tout à fait logique car on utilise un simple tableau qu'il faut de toutes façons parcourir dans son entièreté lorsqu'on veut le multiplier par une autre matrice. Pour des matrices creuses par contre, le nombre d'éléments non nuls a de l'influence car nos files vont alors avoir plus de noeuds qu'il va falloir parcourir pour effectuer un produit.

On remarque ensuite que jusqu'à 2,7% d'éléments non nuls, il était nettement plus avantageux (environ 7 fois plus rapide) d'utiliser des matrices creuses tandis que lorsqu'il y en avait 8,1%, c'était environ $2,5$ fois plus lent. Cela est dû au fait que les structures de données utilisées pour nos matrices creuses sont