UNIVERSITÉ
LAVAL

TP 3 - OpenMPI

Réalisé par **l'équipe 1** constitué de :

Rémi Huguet - 536 795 651
Gabriel Phillipon - 536 795 815

Dans le cadre du cours :
GIF-7104 – Programmation parallèle et distribuée

Travail présenté à :
Marc Parizeau

Remis le :
24 Mars 2021

Objectif

L'objectif de ce TP est d'inverser une matrice carré par la méthode Gauss-Jordan en conservant une bonne stabilité numérique. Le but est également de mesurer le Speedup entre l'implémentation séquentielle et l'implémentation multiprocesseurs de ce programme.

Solution

La solution que nous avons implémentée est basée sur celle présentée dans le PDF associé au TP. Dans un premier temps on répartit chaque rangées sur plusieurs processus puis on calcule le pivot local sur chacun de ces processus, pour ensuite trouver le pivot global le plus grand pour plus de stabilité numérique. On partage ensuite la rangée contenant le pivot et la rangée courante avec tous les autres processus. Ensuite chaque processus se met à jour avec les précédentes rangées.

On peut ensuite échanger les rangées du pivot et la rangée courante s'il y a lieu (à cette étape chaque processus est à jour et connaît la rangée du pivot et possède une matrice augmentée partielle). On peut maintenant diviser la rangée courante par la valeur du pivot afin d'obtenir `1` sur la valeur à l'index du pivot.

Pour chaque rangée, on soustrait la rangée courante multipliée par l'élément k de la rangée courante.

Puis finalement, chaque processus envoient au `ROOT_PROCESS` les rangées inverse qu'il possède et ce, de manière cyclique (rangée `i` possédé par le processus `i % size`), afin de construite la matrice inverse. Par exemple pour reconstruire une matrice inverse 6x6 calculé avec 3 processeurs on a :

rang processeur	row index	matrice
rang 0	0	-0.798273, 0.423694, 0.467226, 2.51627, -0.686944
rang 1	1	0.66994, -1.68287, -1.27129, -3.32241, 3.8435
rang 2	2	0.198696, 1.07213, -1.03497, -0.692323, 0.178142
rang 1	3	-0.410088, 2.39149, 4.64947, 1.41719, -5.54013
rang 2	4	1.34326, -1.30338, -1.94302, 0.493515, 0.956331

Lors de la reconstruction, les opérations sur les matrices de chaque processus on déjà été effectué, et notre méthode de reconstruction nous permet d'éviter l'utilisation de Barrier (MPI_Barrier). Ainsi, notre programme ne contient aucune `barrier`, ce qui permet de diminuer le temps d'exécution.

Spécification Matériel

— OS : Ubuntu 20.10

— Processeur : Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8

— RAM : 8 Go

Résultats

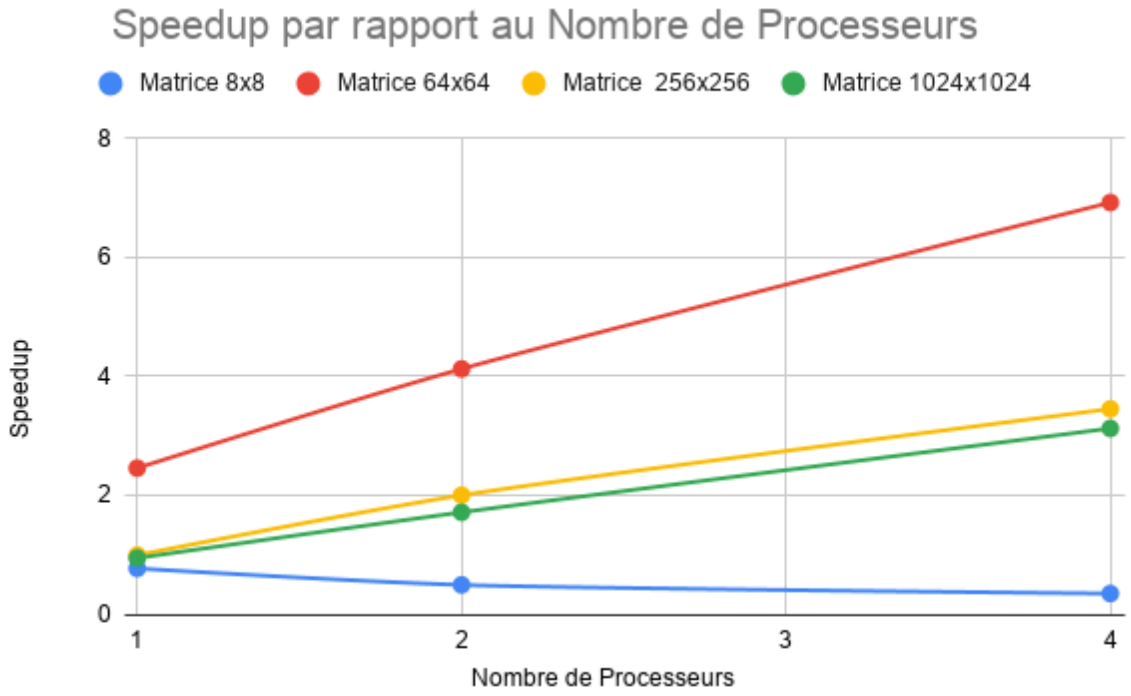

FIGURE 1 – Speedup

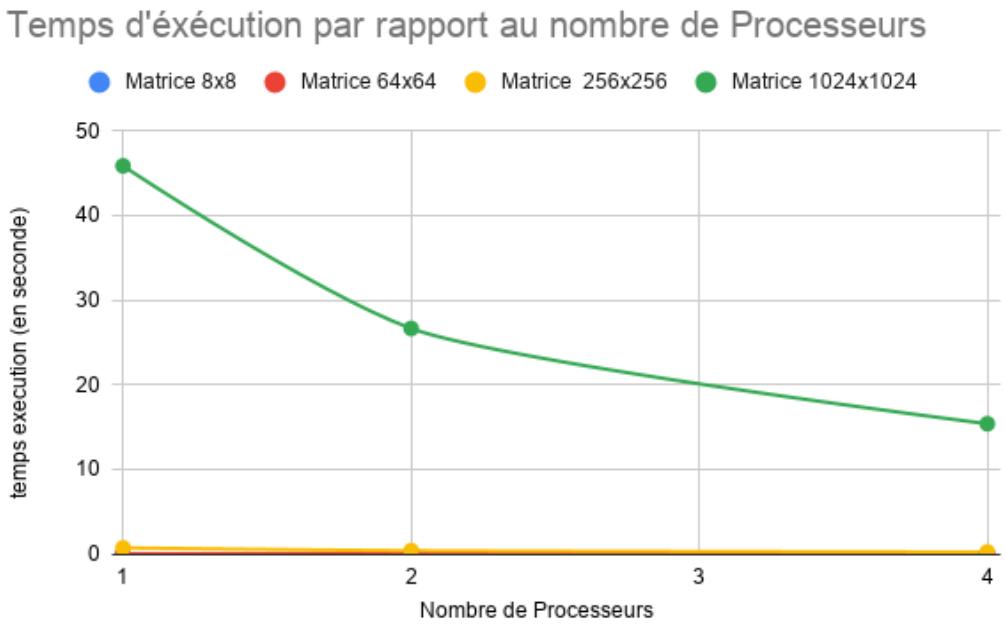

FIGURE 2 – Temps d'exécution

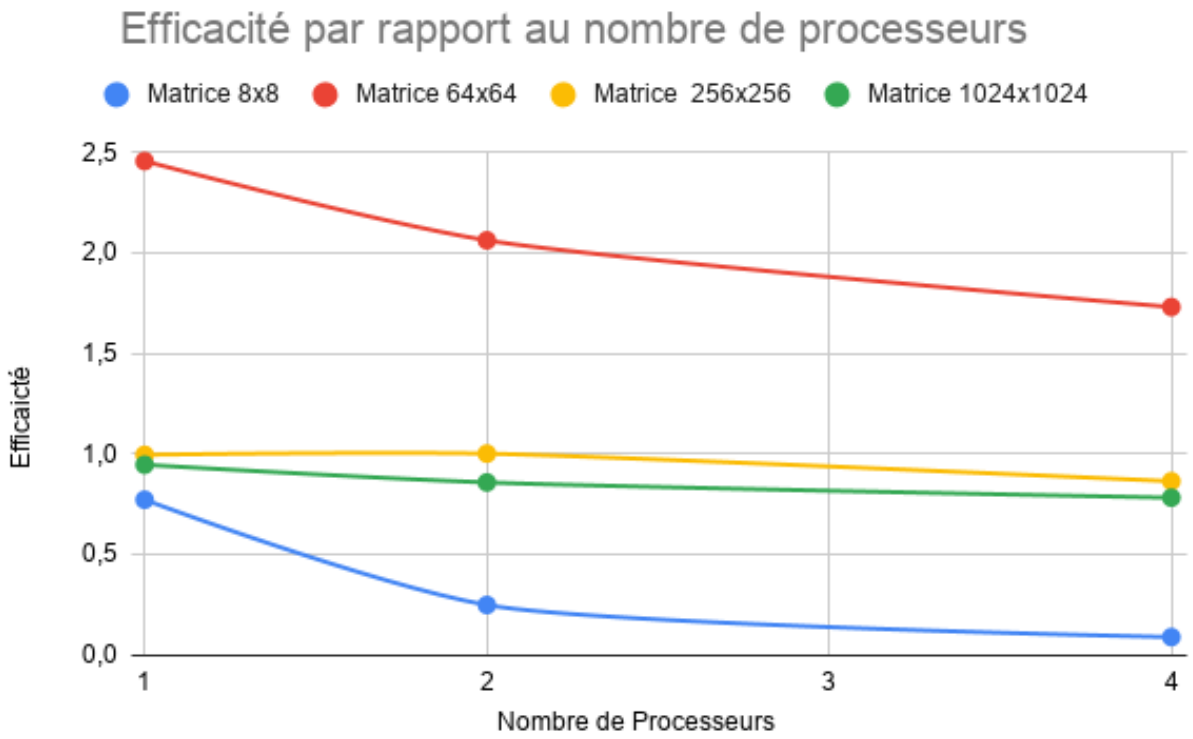

FIGURE 3 – Courbe d'efficacité

Voir les données brutes en Annexe.

Analyse

Comme prévu, pour les petites matrices (`dimension < 16`), les overheads sont trop important et le speedup est inférieur à 1. Puis pour les matrices de dimension plus élevé, le speedup devient significatif.

Dimension	64x64	256x256	1024x1024	8x8
Speedup (4 Processus)	Bon (6.9)	Bon (3.45)	Bon (3.12)	mauvais (0.34)

Mesures

Nous générons des matrices aléatoires ; Ainsi, les mesures, bien qu'opérant sur des données `semblable`, ne sont pas équivalentes. En effet, d'un calcul à l'autre il est possible qu'une matrice "plus facilement" inversible ait été générée et ait donc pris moins de temps à être inversée. De ce fait et à la différence des TPs précédents, nous enregistrons le temps d'exécution à 1 processeur et à `p` processeurs ($1 \leq p \leq 4$) pour chacune des mesures. (voir section 1)

Optimisations

Pour la compilation du programme nous faisons attention à utiliser le `wrapper mpic++`, qui est théoriquement plus apte à faire les optimisations nécessaires dans le cadre d'un programme MPI.

Piste d'amélioration

Pour moins de communication, on pourrait trouver une alternative au BCast. De plus, dans notre solution chaque processus possède en mémoire un tableau de la taille de la matrice augmentée. On pourrait donc trouver une solution permettant d'éviter les stockages et les accès en mémoire inutile. Pour ce TP nous avons travaillé avec l'interface Matrix fournit ; Cela nous a souvent obligé à convertir nos données en `Matrix` (`valarray<double>` à des tableaux de double (`double[]`) ce qui augmente nos temps d'exécutions.

0.1 Remarque

1 Annexe

Données brutes pour chaque fichier de test

Consultable sur un Google Sheet

Dimension Matrice = 8x8				
Nombre de Processeurs	Speedup	Temps exécution séquentiel (s)	Temps exécution parallèle (s)	Efficacité
1	0,7707898345	0,000161324	0,000209297	0,7707898345
2	0,4935800669	0,000173293	0,000351094	0,2467900334
4	0,3486320094	0,000219604	0,000629902	0,08715800236

Dimension Matrice = 64x64				
Nombre de Processeurs	Speedup	Temps exécution séquentiel (s)	Temps exécution parallèle (s)	Efficacité
1	2,456327424	0,0311902	0,0126979	2,456327424
2	4,123550163	0,0285908	0,00693354	2,061775082
4	6,919224075	0,0334723	0,00483758	1,729806019

Dimension Matrice = 256				
Nombre de Processeurs	Speedup	Temps exécution séquentiel (s)	Temps exécution parallèle (s)	Efficacité
1	0,9942403403	0,701188	0,70525	0,9942403403
2	2,000502133	0,752977	0,376394	1,000251067
4	3,451003381	0,759414	0,220056	0,8627508452

Dimension Matrice = 1024x1024				
Nombre de Processeurs	Speedup	Temps exécution séquentiel (s)	Temps exécution parallèle (s)	Efficacité
1	0,94	43,32	45,8536	0,9448200359
2	1,71	45,6519	26,6409	0,856801009
4	3,12	48,0321	15,3775	0,7808827833