TP3
MPI

GIF 4104
Programmation parallèle et distribuée

Pier-Luc Auger (PLAUG2 - 910 098 011)
Mathieu Garon (MAGAR220 - 111 005 164)
Colosse USER46

Université Laval

Rapport TP3

Mise en contexte

Pour ce troisième TP, il est demandé d'écrire un programme parallèle qui inverse une matrice selon la méthode de Gauss Jordan. Le programme doit pouvoir s'adapter en fonction de la taille de la matrice et du nombre de processeurs disponibles. Le tout écrit à l'aide du standard MPI.

Méthode utilisée

Algorithme de répartition de la matrice en sous matrices

L'algorithme présenté mise sur une sépartion de la matrice en plusieurs sous matrices pour tirer profit d'OpenMPI. La matrice est divisé en différente matrices ayant le même nombre de colonnes entre les processeurs. À cet égard, pour certaines tailles de matrices, des processeurs peuvent ne pas être réellement utiles.
Par exemple, dans le cas d'une matrice de 25x25, si 6 coeurs sont à disposition, le programme n'en utilisera réellement que 5, puisque 25%5 = 0 alors que 25%6 !=0. Il est à noter que les chornomètrage est lancé au tout début de l'algorithme et à la fin du rassemblement des données.

Algorithme:

Début du chronomètre;
root : génération de la matrice et de la matrice identité;
Scatter(colonne, matrice, nbColonneParProc);
Pour chaque rangées de matrice:

ProcessusControleur : Calculer l'index à échanger
BCast(index)
swap(index, rangéeActuel)

ProcessusControleur : déterminer le pivot actuel
BCast(pivot)
divise rangéeActuel par pivot

ProcessusControleur : détermination des coéficients pour chaque rangées
BCast(coéficients)
soustrait les rangées avec les coéficients

mise à jours du ProcessusControleur

Gather les colonnes des processus

Fin du chronomètre

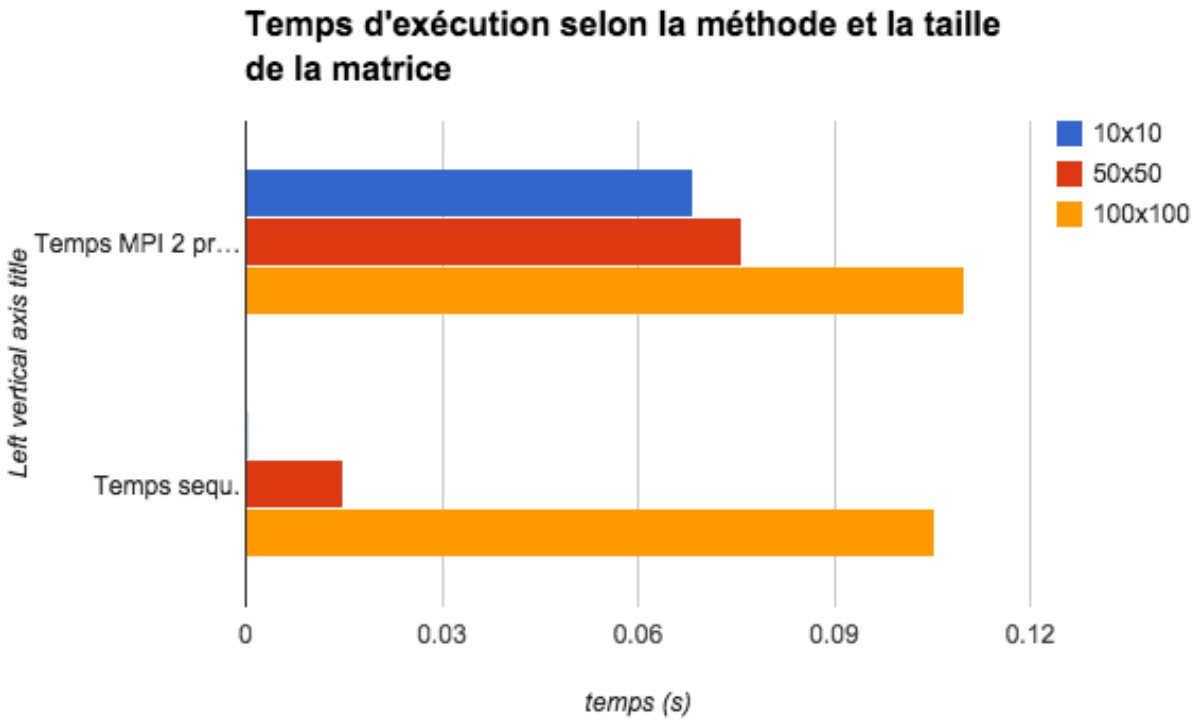
À la vue de cet algorithme, il faut savoir que nos temps d'exécution tiennent compte des overheads de communication et de séparation des tâches entre les processeurs qu'implique l'utilisation d'MPI.

Résultats obtenus

Efficacité sur de petites matrices

D'abord, il est essentiel de savoir que cette méthode est profitable et pertinente sur des matrices relativement grande. De fait, la méthode séquentielle est plus rapide sur les matrices de petite taille. Cela est prévisible, car la communication entre les processus est très longue.

Temps d'exécution selon la méthode et la taille de la matrice			
	10x10	50x50	100x100
Temps MPI 2 processeurs	0.0684	0.07588	0.11
Temps sequ.	0.00024	0.01485	0.1055

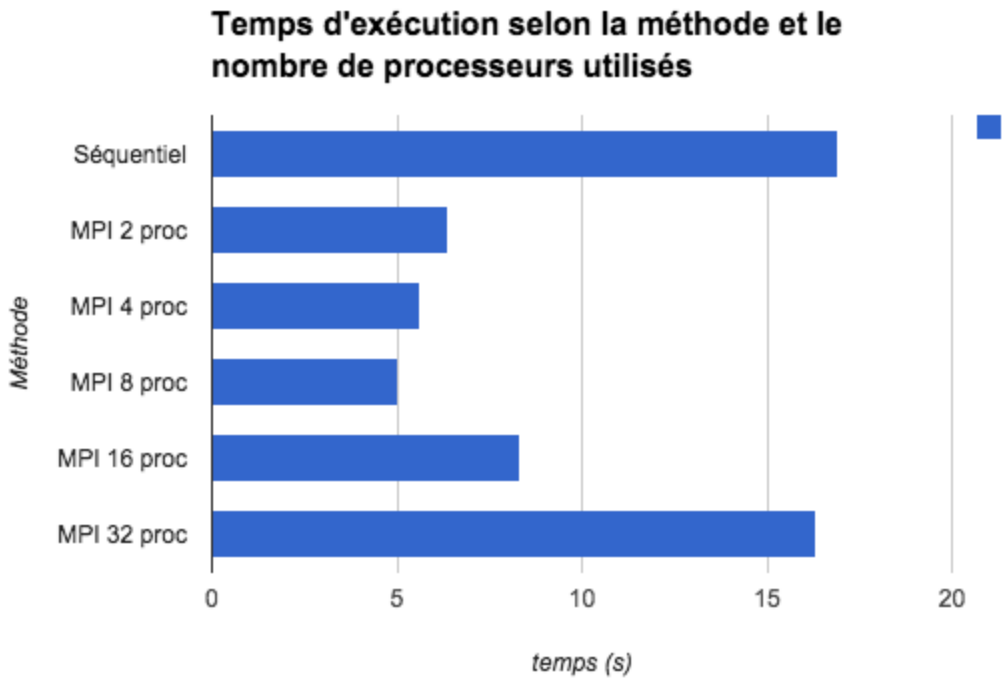


Efficacité sur des matrices de grandes tailles

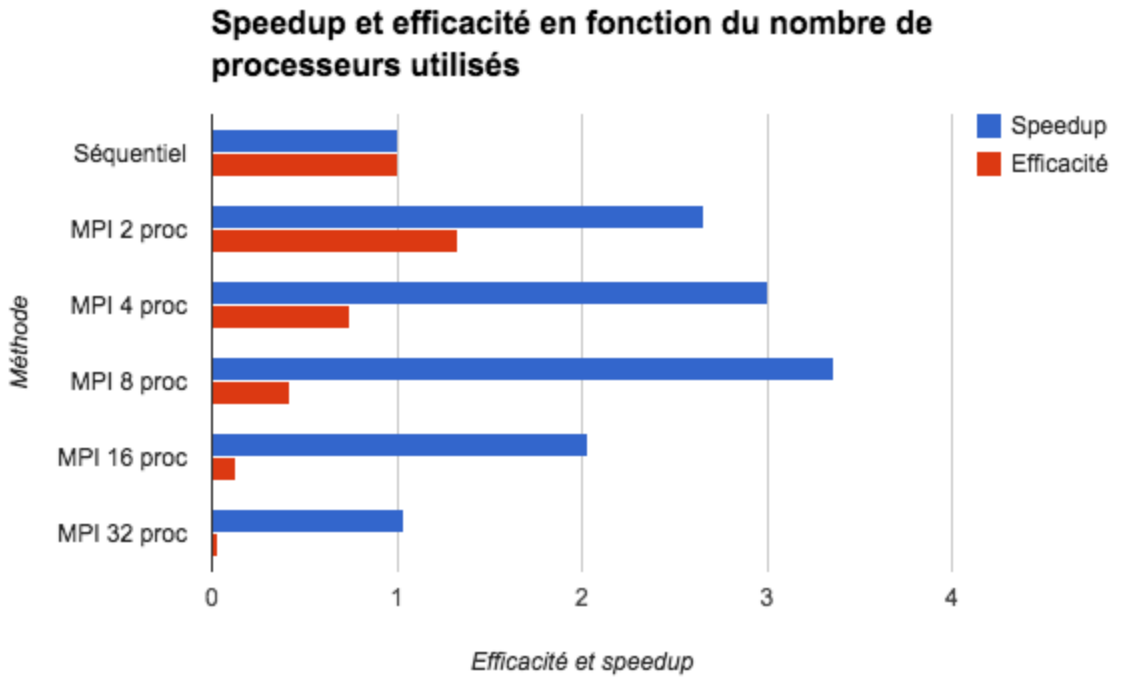
Pour les matrices de grandes tailles, un gain de performance est rapidement observé. Voyons d'abord avec des matrices de 512x512.

Matrices 512x512

Temps d'exécution selon la méthode et le nombre de processeurs utilisés	
	512x512
Séquentiel	16.93
MPI 2 proc	6.37
MPI 4 proc	5.64
MPI 8 proc	5.03
MPI 16 proc	8.33
MPI 32 proc	16.3464

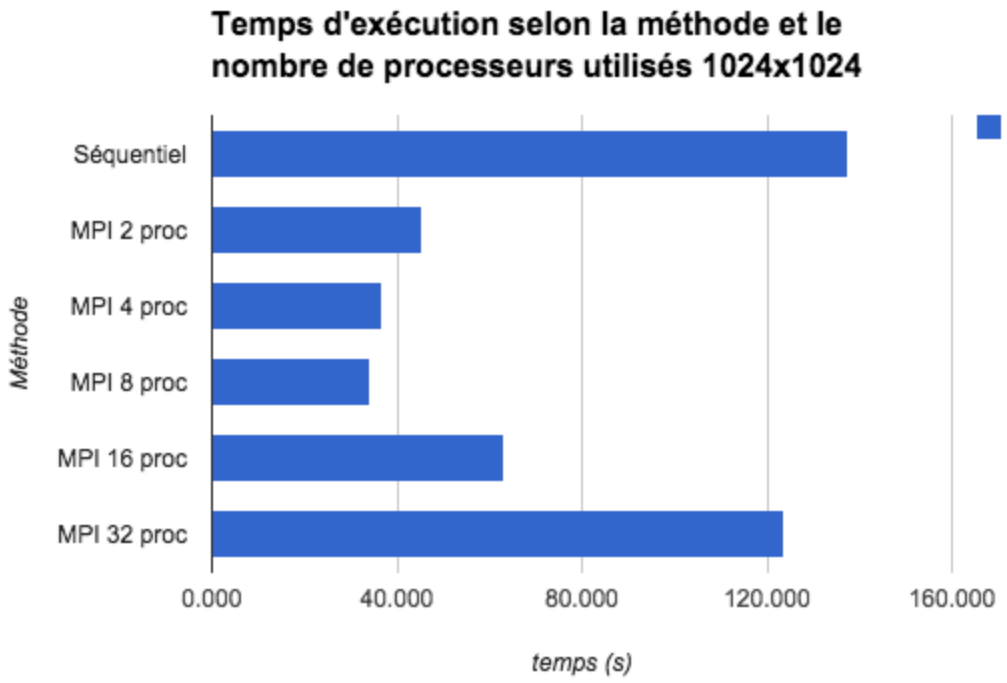


Speedup et efficacité en fonction du nombre de processeurs utilisés		
	Speedup	Efficacité
Séquentiel	1.000	100.00%
MPI 2 proc	2.658	132.89%
MPI 4 proc	3.002	75.04%
MPI 8 proc	3.366	42.07%
MPI 16 proc	2.032	12.70%
MPI 32 proc	1.036	3.24%

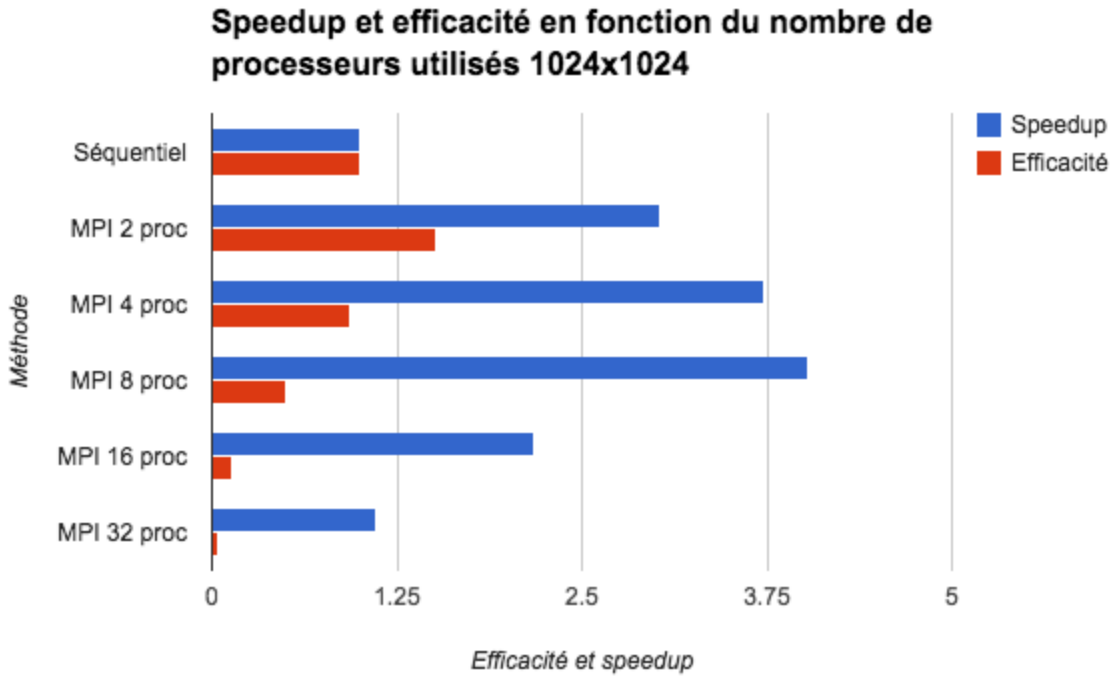


Matrice de 1024x1024

Temps d'exécution 1024x1024 selon la méthode et le nombre de processeurs utilisés			
	Temps	Speedup	Efficacité
Séquentiel	137.598	1.000	100.00%
MPI 2 proc	45.541	3.021	151.07%
MPI 4 proc	36.841	3.735	93.37%
MPI 8 proc	34.133	4.031	50.39%
MPI 16 proc	63.201	2.177	13.61%
MPI 32 proc	123.474	1.114	3.48%



Speedup et efficacité en fonction du nombre de processeurs utilisés 1024x1024		
	Speedup	Efficacité
Séquentiel	1.000	100.00%
MPI 2 proc	3.021	151.07%
MPI 4 proc	3.735	93.37%
MPI 8 proc	4.031	50.39%
MPI 16 proc	2.177	13.61%
MPI 32 proc	1.114	3.48%



Analyse des résultats

On peut ressortir certaines tendances des graphiques montrant nos résultats. On remarque bien que sur Colosse, la programmation parallèle est efficace et plus le nombre de coeur utilisé est grand plus le programme s'exécute rapidement.

Par contre, on remarque qu'il faut que les matrices soient suffisemment grande pour que les overheads qu'engendrent la communication MPI n'ait qu'un effet négligeable sur le temps d'exécution du programme.

Le speedup est très intéressant. On arrive à faire mieux que tripler la vitesse avec une efficacité autour de 50%.

Aussi, au-delà de 8 processeurs, l'efficacité chute. C'est peut-être du au fait que nous avons 8 processus par noeud sur Colosse. Il est à noter que l'algorithme présenté utilise beaucoup de communication vu la division en colonnes

```
# Shell à utiliser pour la tâche
#PBS -S /bin/bash
#PBS -N TACHE_EXAMPLE   # Nom de la tâche
#PBS -A colosse-users  # Identifiant Rap-ID
#PBS -l nodes=1:ppn=8   # Nombre de noeuds et nombre de processus par noeud
#PBS -l walltime=300    # Durée en secondes
```

En somme, nous sommes parvenus à montrer que la parallélisation avec MPI permettait d'augmenter significativement la performance du calcul de l'inverse d'une matrice.

Lancement de l'exécutable

Pour utiliser notre programme, il s'agit d'utiliser `cmake` pour générer le makefile. Le CMakeList est fourni avec le code. Cela permettra ensuite d'utiliser la commande `make` pour compiler le programme.

Sur colosse, le code est déjà disponible et exécutable. Des scripts Moab sont dans le répertoire `/home/user46/TP3/`. Les scripts dans le répertoire permettent de lancer notre programme.