**Programmation parallèle et distribuée**
**GIF-4104**

# TP 3
# MPI

**Équipe 2**
Gérémy Desmanche, 111 232 013
Samuel Cloutier, 111 232 614

Le travail consiste à créer avec OpenMPI un programme parallèle d'inversion de matrice de taille variable supportant un nombre quelconque de processeurs. Celui-ci produit en sortie des informations quant à la rapidité et l'exactitude du procédé d'inversion.

**Description du programme:**

L'algorithme d'inversion séquentiel choisi est le suivant, car il permet l'implémentation parallèle avec pipeline:

Soit A une matrice de taille $n \times n$.

Soit AI la matrice A augmentée de la matrice identité de taille $n \times n$.

Soit $a_{ij}$ l'élément à la ligne $i$ et la colonne $j$ de AI (les indices commencent à 0).

Soit $a_i$ la ligne $i$ entière de AI.

```
Construire AI
pos_max ← []
pour tout 0 ≤ k < n:
      colonne ← j tel que |a_kj| = max_{0≤l<n} |a_kl|
      si ( max_{0≤l<n} |a_kl| = 0) Erreur!
      a_k ← a_k / a_{k,colonne}
      pour tout 0 ≤ i < n, i ≠ k:
            a_i ← a_i − a_{i,colonne} · a_k
      pos_max[k]← colonne
i ← 0
tant que i < n:
      si pos_max[i] = i:
            i ← i + 1
      sinon:
            a_i ↔ a_{pos_max[i]}
            pos_max[i] ↔ pos_max[pos_max[I]]
Retourner la moitié droite de AI
```

Sélection

Réduction

Enregistrement

Réarrangement

La première boucle revient à appliquer la réduction de Gauss-Jordan de haut en bas puis de bas en haut, mais de manière à ce que la ligne pivot de l'itération k soit la ligne k. On choisi la valeur maximale dans la ligne afin de réduire l'erreur numérique.
La deuxième boucle réordonne les lignes de manière à ce que la partie gauche de AI soit l'identité.

Pour l'implémentation parallèle, on distribue les lignes de façon cyclique entre les processus.

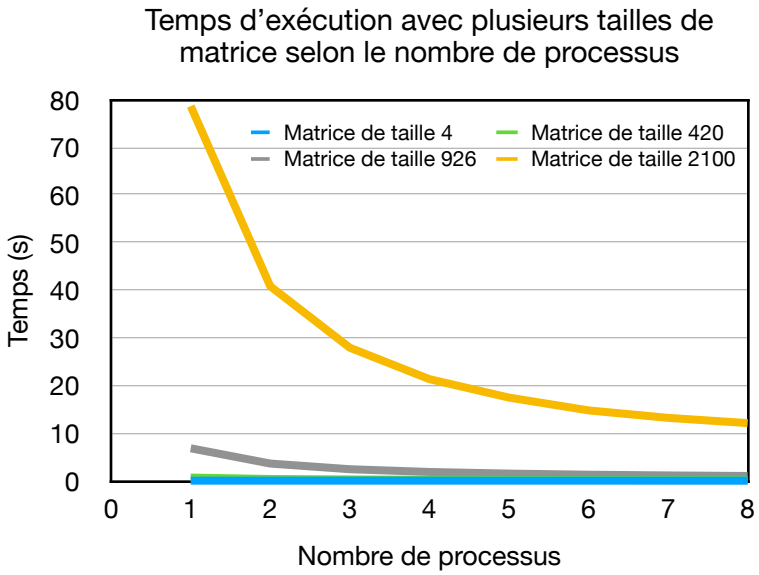
Initialement, le processus 0, qui détient la matrice entière, envoie à tous les autres processus les lignes qui leur appartiennent.

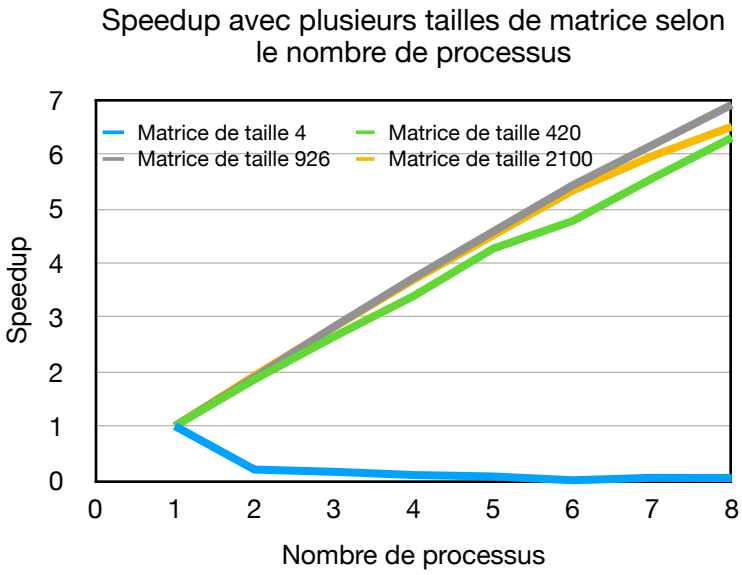
À l'itération k, le processus possédant la ligne $a_k$ effectue Sélection puis envoie la ligne $a_k$ résultante au processus suivant (0 pour le dernier). Il envoie également au suivant la colonne à réduire, puis effectue Réduction sur l'ensemble des lignes qu'il détient (qui ne sont pas $a_k$).
Le processus suivant reçoit ces deux informations et les renvoie à son propre processus suivant, sauf si $a_k$ appartient audit suivant. Il effectue finalement Réduction sur les lignes qu'il détient.

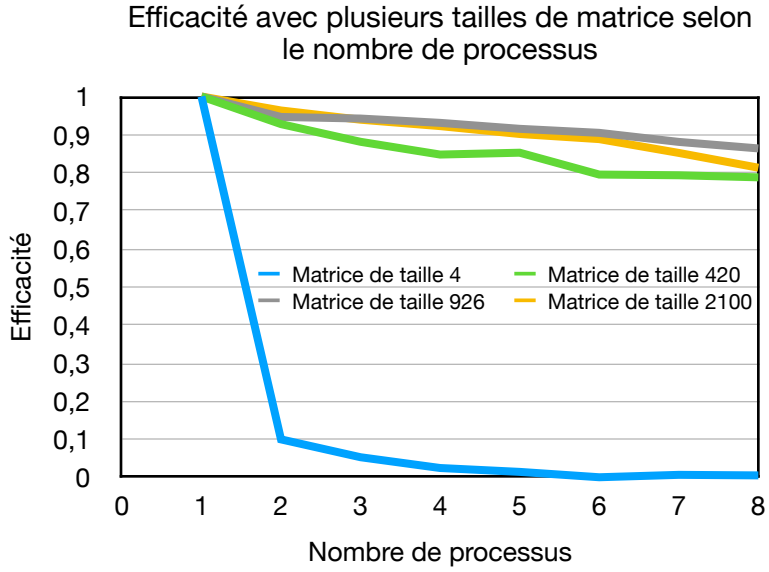
Dès que le processus 0 à l'information requise, il procède à Enregistrement.

Finalement, les lignes sont rapatriées sur le processus 0, qui effectue ensuite Réarrangement.

**Résultats expérimentaux:**



**Analyse des résultats:**

Sans surprise, la solution parallèle appliquée sur de petites matrice est peu efficace. Cela est sans nul doute dû aux coûts de communications, dont la latence peut alors être plus longue que le traitement à un processus.

L'efficacité est acceptable (entre 0,788 et 0,963) lorsque les matrices sont plus grandes. On remarque cependant une diminution progressive à mesure que le nombre de processus augmente. Cela peut être dû à la partie Réarrangement de l'algorithme, qui effectue séquentiellement $2n^2$ assignations en pire cas. De plus, la distribution des lignes, au début, pendant et à la fin de l'algorithme prend un temps de plus en plus significatif.

Les résultats sont valides, comme le montre le tableau des erreurs en annexe 2.

Les speedups sont calculés en fonction du temps parallèle à 1 processus, car l'algorithme implanté est plus rapide que l'algorithme séquentiel fourni (voir annexe 1).

**Discussion:**

Les résultats obtenus sont satisfaisants. L'implémentation choisie tire profit d'un pipeline tout en minimisant sa trace en mémoire, chaque processus ne gardant que les valeurs avec lesquelles il travaille. Cependant, la communication, qu'on a finalement laissée synchrone par manque de ressources temporelles pour l'implémenter de façon asynchrone, nuit à l'efficacité du pipeline. En effet, les communications bloquent les processus plus longtemps que nécessaire, puisqu'ils envoient de manière synchrone des données qu'ils n'ont plus à modifier. Le code remis illustre l'intention et la manière d'utiliser *Isend*, qui a finalement été défini à *Send* par une directive de préprocesseur.

En théorie, en prenant séquentiellement les lignes pivot (plutôt qu'en sélectionnant la ligne dont la valeur de pivot est maximale), l'algorithme risque une plus grande instabilité numérique. En pratique, cela n'a pas été observé.

**Améliorations possibles:**

Tel que mentionné, on voudrait faire fonctionner les communications asynchrones, ou mieux encore, passer à une approche utilisant des fenêtres sur la mémoire de chaque processus.

Aussi, il pourrait être pertinent de considérer les utilitaires de communications collectives spécialisés pour la distribution initiale et le rapatriement final des lignes, ce qui aplanirait la pente des courbes d'efficacité.

Pour obtenir des données empiriques plus intéressantes et ainsi pouvoir mieux étudier l'efficacité de différentes implémentations, il faudrait paralléliser le calcul d'erreur, par exemple en implémentant Strassen avec OpenMPI, car il occupe la majeure partie du temps d'exécution des tests, pouvant dépasser quatre minutes pour une inversion d'une quinzaine de secondes.

**Annexe 1: Tableaux des résultats**

Résultats du programme parallèle sur des matrices de taille 4

| Nombre de processus | Temps de calcul (s) | Speedup ($T_1/T_p$) | Efficacité |
|---|---|---|---|
| 1 | 0,000012597 | 1,000 | 1,000 |
| 2 | 0,000063007 | 0,200 | 0,100 |
| 3 | 0,000079060 | 0,159 | 0,053 |
| 4 | 0,00012689 | 0,099 | 0,025 |
| 5 | 0,00017200 | 0,073 | 0,015 |
| 6 | 0,0037409 | 0,003 | 0,001 |
| 7 | 0,00026167 | 0,048 | 0,007 |
| 8 | 0,00028991 | 0,043 | 0,005 |

Résultats du programme parallèle sur des matrices de taille 420

| Nombre de processus | Temps de calcul (s) | Speedup ($T_1/T_p$) | Efficacité |
|---|---|---|---|
| 1 | 0,65141 | 1,000 | 1,000 |
| 2 | 0,35117 | 1,855 | 0,927 |
| 3 | 0,24645 | 2,643 | 0,881 |
| 4 | 0,19214 | 3,390 | 0,848 |
| 5 | 0,15283 | 4,262 | 0,852 |
| 6 | 0,13655 | 4,770 | 0,795 |
| 7 | 0,11731 | 5,553 | 0,793 |
| 8 | 0,10336 | 6,302 | 0,788 |

Résultats du programme parallèle sur des matrices de taille 926

| Nombre de processus | Temps de calcul (s) | Speedup ($T_1/T_p$) | Efficacité |
|---|---|---|---|
| 1 | 6,8004 | 1,000 | 1,000 |
| 2 | 3,5933 | 1,893 | 0,946 |
| 3 | 2,4061 | 2,826 | 0,942 |
| 4 | 1,8266 | 3,723 | 0,931 |
| 5 | 1,4862 | 4,576 | 0,915 |
| 6 | 1,2530 | 5,427 | 0,905 |
| 7 | 1,1030 | 6,165 | 0,881 |
| 8 | 0,9842 | 6,910 | 0,864 |

Résultats du programme parallèle sur des matrices de taille 2100

| Nombre de processus | Temps de calcul (s) | Speedup ($T_1/T_p$) | Efficacité |
|---|---|---|---|
| 1 | 78,6448 | 1,000 | 1,000 |
| 2 | 40,8215 | 1,927 | 0,963 |
| 3 | 27,9164 | 2,817 | 0,939 |
| 4 | 21,3186 | 3,689 | 0,922 |
| 5 | 17,4466 | 4,508 | 0,902 |
| 6 | 14,7589 | 5,329 | 0,888 |
| 7 | 13,1886 | 5,963 | 0,852 |
| 8 | 12,0909 | 6,504 | 0,813 |

Résultats du programme séquentiel selon la taille de la matrice

| Taille de la matrice | Temps de calcul $T_{seq}$ (s) | Temps de calcul $T_1$ (s) | Speedup ($T_{seq}/T_1$) |
|---|---|---|---|
| 4 | 0,00002043 | 0,000012597 | 1,622 |
| 420 | 2,4780 | 0,65141 | 3,804 |
| 926 | 26,8641 | 6,8004 | 3,950 |
| 2100 | 309,2310 | 78,6448 | 3,932 |

Les résultats ont été obtenus sur un ordinateur ayant les spécificités suivantes:

OS: Arch Linux x86_64
Kernel: 5.11.8-arch1-1
Shell: bash 5.1.4
Terminal: urxvt
CPU: AMD Ryzen 7 2700X (16) @ 3.700GHz
GPU: AMD ATI Radeon Pro WX 5100
Memory: 2995MiB / 32121MiB

**Annexe 2: Tableaux des erreurs**

Erreur sur la solution trouvée selon la taille de la matrice

| Taille de la matrice | Erreur séquentielle | Erreur parallèle |
|---|---|---|
| 4 | 0,0 | 0,0 |
| 420 | -7.27596e-12 | -4.26326e-12 |
| 926 | -7.85576e-11 | 2.54659e-11 |
| 2100 | -1.10595e-09 | -9.0904e-10 |

**Annexe 3: Procédure de test**

```
import subprocess
import sys

binaries = ['./tp3_sequential', './tp3_parallel']
test_count = 1
sizes = [4, 42, 420, 926, 2100]
thread_counts = [8, 7, 6, 5, 4, 3, 2, 1]
seed = 0

tests_seq = [{'bin': './tp3_sequential', 'size': s, 'durations': [],
'errors': []}
     for s in sizes
]

tests_para = [{'bin': './tp3_parallel', 'size': s, 'thread_count': c,
'durations': [], 'errors': []}
     for s in sizes
     for c in thread_counts
]
for i in range(test_count):
     for test in tests_seq:
          sorties = (subprocess.run([test['bin'], str(test['size']),
str(seed)], text=True,

stderr=subprocess.PIPE).stderr).split("\n")
          test['durations'] += [float((sorties[2].split(" "))[-1])]
          test['errors'] += [float((sorties[3].split(" "))[-1])]
          with open(sys.argv[1], 'a') as f:
               f.write(str(test) + '\n')

for i in range(test_count):
     for test in tests_para:
          sorties = (subprocess.run(['mpirun', '-np',
str(test['thread_count']),
                                          test['bin'],
str(test['size']), str(seed)], text=True,

stderr=subprocess.PIPE).stderr).split("\n")
          test['durations'] += [float((sorties[2].split(" "))[-1])]
          test['errors'] += [float((sorties[3].split(" "))[-1])]
          with open(sys.argv[1], 'a') as f:
               f.write(str(test) + '\n')
```