GIF-4104 - TP 3

Équipe 1 : Vincent Commin & Louis Leenart

1 Introduction

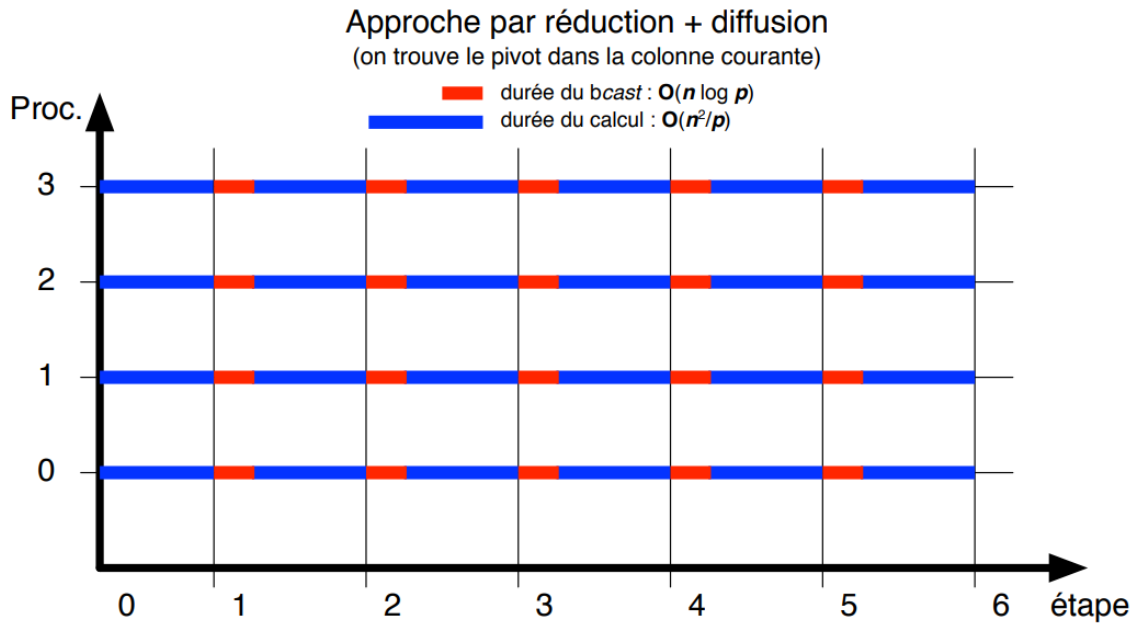
Pour ce troisième TP, nous avons implémenté la parallélisation d'inversion d'une matrice en utilisant méthode de *Gauss Jordan* avec la librairie *OpenMPI*. En se basant sur l'implémentation séquentielle proposée, nous avons suivi l'approche par réduction puis diffusion successives.

2 Notre approche

Dans ce TP, nous avons fait de notre mieux pour paralléliser l'inversion d'une matrice en simplifiant au maximum la lecture du code, tout en donnant des performances satifsaisantes. Pour inverser une matrice via la méthode de *Gauss Jordan*, il existe deux approches : par réduction puis diffusion successives, et par pipeline. Débutant avec MPI, nous avons implémenté la méthode par réduction puis diffusion successives. L'algorithme que nous avons mis en place est le suivant :

```
AI : Matrix [A I]
size, rank : integer

FOR k = 0 TO AI.rows DO
    max = 0, pivotIndex = k
    FOR i = k TO AI.rows DO // Find greatest pivot for column k
        IF (i mod size) == rank && abs(AI[i, k]) > max DO
            max = abs(AI[i, k])
            pivotIndex = i
        DONE
    DONE

    // share and find greatest pivot with all workers
    Allreduce({v:max, i:pivotIndex}, res, size, MPI_DOUBLE_INT, MPI_MAXLOC)
    pivotIndex = res.i

    Bcast(AI[pivotIndex, 0], AI.cols, MPI::DOUBLE, pivotIndex mod size)

    IF IA[pivotIndex, k] == 0 DO throw exception DONE

    IF pivotIndex != k DO AI.swapRows(pivotIndex, k) DONE

    v = AI[k, k] // Normalisation
    FOR j = 0 TO AI.cols DO
        AI[k, j] = AI[k, j] / v
    DONE

    FOR i = 0 TO AI.rows DO // For each rows
        IF i mod size == rank && i != k DO
            AI[i] -= AI[k] * AI[i, k]
        DONE
    DONE

    FOR i = 0 TO AI.rows DO
        Bcast(AI[i, 0], AI.rows, MPI::DOUBLE, i mod size)
    DONE
DONE

FOR i = 0 TO AI.rows DO // Copy right side of AI into result
    A = AI.getData()[slice(i * AI.cols + AI.rows, AI.rows, 1)]
DONE
```

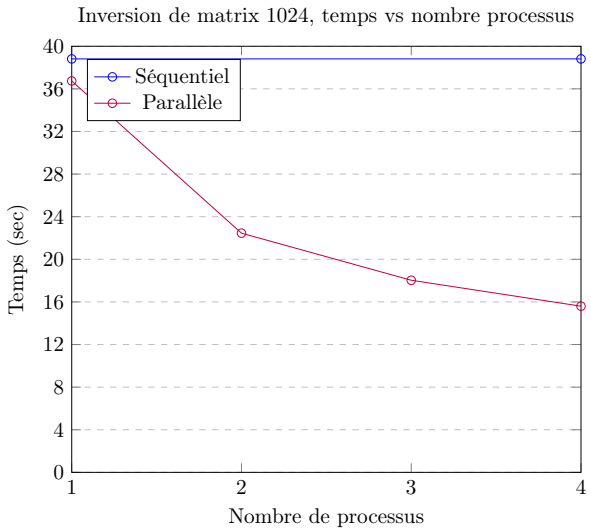
Avec MPI, nous pouvons répartir les différents calculs sur plusieurs processus. Notre algorithme suit la répartition suivante (source : Cours GIF-4104) :

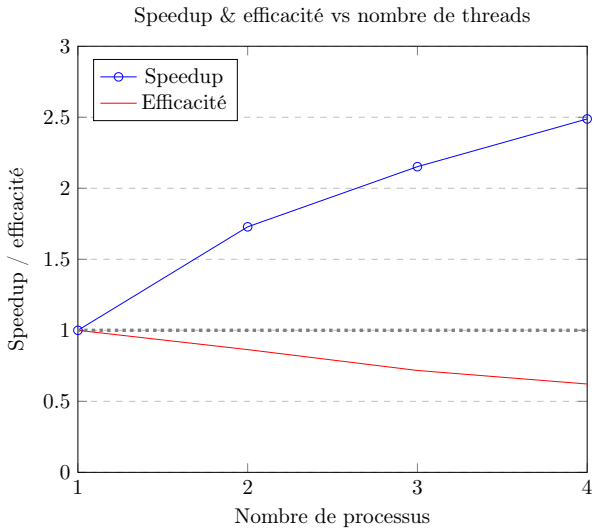


3 Machine utilisée pour les tests de performance

Modèle	intel i7-8550U
Architecture	x86_64
OS	Archlinux
Fréquence CPU	3.4GHz
Cœurs physiques	4
Ram	16 Go, 2400 MT/s
OpenMPI	version **4.1.2-1**

4 Résultats obtenus





5 Analyse

Tout d'abord, nous avons conduit notre analyse sur l'inversion de matrices de tailles variées. Les résultats ayant la même tendance par rapport à la taille, nous avons décidé de nous concentrer seulement sur l'inversion d'une matrice 1024x1024 aléatoire (les données brutes pour d'autres tailles sont disponibles) dans l'archive.

L'utilisation de MPI, en contraste avec OpenMP, permet d'établir un environnement de travail spécifique à chaque processus de calcul. En effet, pour que deux processus communiquement, il ne faut plus simplement appeler une fonction, il faut faire une transmission de message. Une des méthodes de transmission de message s'appelle le 'Broadcast' et est particulièrement couteuse étant donné quelle fait appel à tous les processus. MPI possède de nombreux moyens de transmettre et reçevoir des messages, cependant, le 'Broadcast' est ce que nous avons utilisé pour notre algorithme.

Pour en revenir aux performances de notre algorithme, nous pouvons remarquer avec le premier graphique que le temps de calcul pour une matrice 1024 décroit comme attendu quand le nombre de processus augmente. Ce comportement est celui attendu étant donné le travail de parallélisation. De plus, on note que le temps de calcul initial est assez conséquent (environ 40 secondes), et le minimum obtenu pour 4 processus est autour de 16 secondes. Ces résultats sont satifsaisants, mais nous pouvons étudier le speedup et l'efficacité pour mieux comprendre les gains en performance.

D'après le graphique, on remarque que pour 4 processus, le speedup atteint 2.5 ce qui est non négligeable pour une application parallèle. De plus, l'efficacité atteint 0.6 pour 4 processus, ce qui n'est pas particulièrement impressionnant, mais cela reste correct. On peut donc en déduire que la parallélisation de l'algorithme d'inversion de matrice selon la méthode de Gauss Jordan est fonctionnelle.

Cependant, il est évident que l'approche naïve du problème ne nous permet pas d'utiliser le plein potentiel de parallélisation. En effet, il existe au moins une autre approche pour cette algorithme, dite par "répartition cyclique" ou par "pipeline" permettant d'éliminer un maximum les temps d'attente des processus imposés par notre approche.

6 Conclusion

Avec la prise en main de MPI, nous avons pu mettre en pratique une nouvelle approche de la parallélisation de calculs. Dans un premier temps, nous nous sommes lancés sur l'implémentation de la méthode naïve pour l'algorithme de Gauss Jordan, avec succès. Même si les résultats se sont montrés satifsaisants, nous avons tenté de mettre en place l'algorithme par pipeline, mais rattrapés par le temps, nous n'avons pas eu le temps de l'achever.

Cependant, la méthode de Gauss Jordan n'est pas la seule façon d'inverser une matrice. Par exemple, la méthode de *Cayley-Hamilton* dont la formule se résume à :
$A^{-1} = \frac{1}{det(A)} \sum_{k_1,k_2,\ldots,k_n} \prod_{n-1}^{l=1} \frac{(-1)^{k_l+1}}{l^{k_l} k_l!} tr(A^l)^{k_l}$
Aussi, pour les matrices dites *positive definite*, l'application de la décomposition de *Cholesky*, avec la formule suivante : $A^{-1} = (L^*)^{-1} L^{-1}$