Rapport Tp n°3 - OpenMPI
GIF-7104 programmation parallèle et distribuée

Equipe 9

Sébastien Marsaa-poey

Pierre Snell

26 mars 2019

Table des matières

Table des figures

Liste des tableaux

List of Algorithms

1 Introduction

L'objectif de ce TP est de réaliser un programme multifilaire d'inverser une matrice grâce à la méthode de Gauss-Jordan. Le programme est écrit en C++, et la parallélisation repose sur la librairie OpenMPI.
Le programme doit gérer une matrice de taille quelconque et potentiellement non-inversible. On souhaite également que la sortie, envoyée dans stdout, soit l'erreur entre le produit de la matrice et de son inverse et la matrice identité (cette erreur est due au calcul sur machine qui, même performant n'est pas exact).

2 Méthodes

Pour ce troisième Tp, nous avons effectué une parallélisation sur les lignes de la matrice.
On répartit donc les ressources afin de pouvoir :

— Trouver le maximum selon une colonne

— Faire des opérations linéaires sur chacunes des lignes en parallèle.

2.a Trouver le pivot

À chaque itération, les lignes restantes sont divisées en groupes pour chaques processeurs. Chaque processeur calcule un minimum sur les valeurs qui lui sont affectées et une opération de réduction trouve et distribue ensuite le maximum global.
De ce fait on essaye de répartir au mieux la charge de travail sur toutes les ressources.

2.b Échanger le pivot

Deux cas peuvent se présenter :

— La ligne du pivot et la ligne courante sont gérées par le même processeur
Cas simple, on échange directement les lignes dans la mémoire de ce processeur.

— La ligne du pivot et la ligne courante sont gérées par deux processeurs différents
On est obligé de faire communiquer ces deux processus pour qu'ils s'échangent les lignes
Cela demande plus de temps car la communication oblige à utiliser les ressources d'openMPI.

2.c Faire les opérations linéaires

Le principal avantage de la parallélisation est ici.
Chaque ligne effectue une opération linéaire avec les valeurs du pivot partagé.
C'est le partie la plus demandante en ressources et sa parallélisation donne un gros gain de temps.

2.d Regrouper les résultats

À ce stade, chaque processus a calculé ses lignes de la matrice inverse. Il ne reste qu'à les mettre en commun dans une seule matrice. Pour cela, chaque processus effectue un broadcast sur toutes ses lignes. Elles sont récupérées par tout les autres processus . Cela signifie qu'à la fin, tous les processus possèdent la matrice inverse.
Cela n'est pas nécessaire mais pratique si l'on veut étendre notre programme et faire effectuer d'autres calculs sur ce résultat.

Algorithm 1 Version Parallele

Entrées : Nombre de threads
Sorties : Affiche l'erreur d'inversion
$\mathcal{X}$: Fait pour chaque processus différent

$\mathcal{N} \leftarrow$ Nombre de thread
debut $\leftarrow$ démarre chrono
démarre MPI et alloue les structures
for $i <$ taille(matrice) **do**
    for $k = i <$ taille(matrice) **do**
        $\mathcal{X}$: donne à chaque processeurs ses valeurs de pivots
    $\mathcal{X}$: trouve le maximum local des pivots donnés ▷ utilisation d'un map en C qui est déjà trié
    pivot ←ReduceAll(column i) ▷ Trouve le maximum des maximas locaux et le partage
    if (line_i, pivot) ∈ process_i **then**
        switch pivot ↔ line_i ▷ Même process pas de soucis
    else
        Communicate with pivot_process ▷ Send and receive lines with mpi
        switch pivot ↔ line_i
    normalize pivot_line
    for $k = i <$ taille(matrice) **do**
        $coeff \leftarrow$ matrice(i,k)
        $line = line - coeff * pivot$
$fin \leftarrow$ arrête chrono
Affiche $fin - debut$

3 Résultats expérimentaux

OS	Linux mint 19.1	Archlinux (Noyeau linux 4.20.11)
Architecture :	x86_64	
Processeur(s) :	8	
Thread(s) par cœur :	2	
Cœur(s) par socket :	4	
Socket(s) :	1	
Nom de modèle :	Intel i7-7700HQ 2.80GHz	Intel i7-4700HQ 2.4GHz
Vitesse maximale du processeur en MHz :	2800	3400
Vitesse minimale du processeur en MHz :	800	
Cache L1d L1i L2 L3 :	32K 32K 256K 6144K	
Ram :	16Go DDR4 2400MHz	

TABLE 1 – Spécifications des machines utilisée pour les tests

Les programmes ont été testés avec une matrice aléatoire de taille 2000*2000.
La compilation a été optimisée avec l'option -O3.

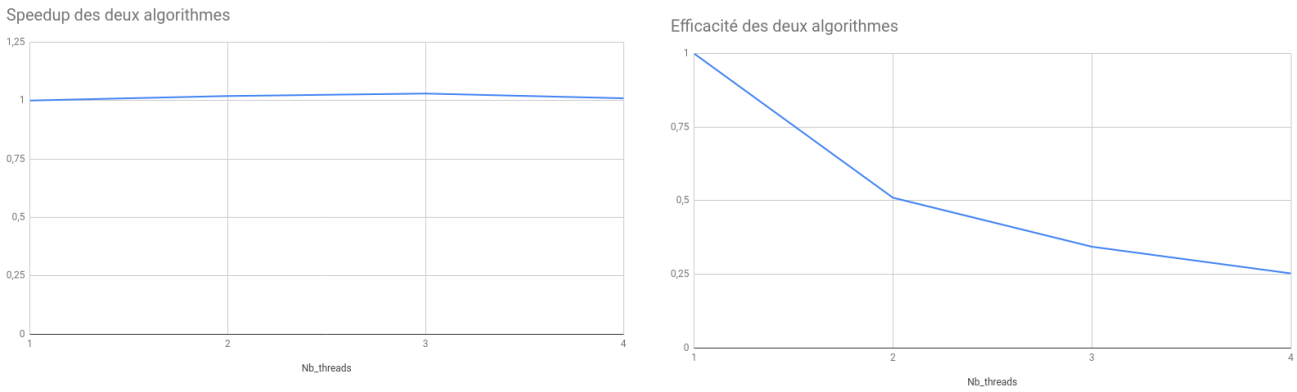

FIGURE 1 – Speedup et efficacité de l'algorithme

4 Discussion

4.a Analyse des résultats

Le speedup n'est pas très élevé, peut-être un goulot d'étranglement que nous n'avons pas repéré dans notre programme... (peut-être le regroupement de la matrice finale ?)
Dans tout les cas notre programme prend bien le nombre de thread en entrée et réalise une exécution parallèle.
Même si l'algorithme est en $O(n^3)$, un speed-up un peu plus important était attendu...
Avec ces résultats, on pourrait penser qu'une implémentation séquentielle est suffisante et qu'il n'est pas nécessaire de paralléliser l'algorithme.
En revanche, nous savons que l'on peut augmenter le speedup grâce à une parallélisation et nos résultats vont à l'encontre de ce que l'on attendait. Par manque de temps nous ne pouvons pas investiguer plus sur ce problème.

4.b Améliorations possibles

Il serait possible de réduire le temps perdu dans la communication entre processus avec une meilleure utilisation de MPI.
Pour notre première utilisation de la bibliothèque, nous nous sommes restreints aux fonctionnalités les plus simples.
Il faudrait aussi s'assurer d'avoir une consommation de RAM raisonnable ou d'adapter notre programme afin de la gérer.
Dans notre cas, il nous est impossible de travailler sur des matrices de taille 10000 sans que notre programme ne soit tué par l'OS pour sa consommation de RAM trop importante ou que la lib STD du C nous bloque (bad alloc).

4.c Difficultés

Le transfert de lignes et le reduce de la matrice via les fonctions de MPI s'est révélé peu intuitif et nous a fait perdre un certain temps.
De plus, éliminer les communications et énoncés inutiles pour garder des performances correctes n'était pas chose aisée.
Aussi, les types de valeurs de la bibliothèque de matrice ne sont pas intuitif, (le mélange entre slice_array et valarray et leurs casts).