Rapport TD1

Programmation parallèle

JODAR SOARES Gustavo
LOPEZ NOREÑA Vanessa

github link:
https://github.com/Gustavo-Jodar/Programmation_Parallele

01/2023

1.1. Le temps de calcul du produit matrice--matrice avec différentes dimensions

```
gustavo@gustavo-X510URR:~/Documentos/facul/ENSTA/OS202/Course2023/TravauxDirigés/TD_numero_1/sources$ ./TestProductMatrix.exe 1022
Test passed
Temps CPU produit matrice-matrice naif : 1.72386 secondes
MFlops -> 1238.45
gustavo@gustavo-X510URR:~/Documentos/facul/ENSTA/OS202/Course2023/TravauxDirigés/TD_numero_1/sources$ ./TestProductMatrix.exe 1023
Test passed
Temps CPU produit matrice-matrice naif : 1.74163 secondes
MFlops -> 1229.42
gustavo@gustavo-X510URR:~/Documentos/facul/ENSTA/OS202/Course2023/TravauxDirigés/TD_numero_1/sources$ ./TestProductMatrix.exe 1024
Test passed
Temps CPU produit matrice-matrice naif : 11.7805 secondes
MFlops -> 182.292
gustavo@gustavo-X510URR:~/Documentos/facul/ENSTA/OS202/Course2023/TravauxDirigés/TD_numero_1/sources$ ./TestProductMatrix.exe 1025
Test passed
Temps CPU produit matrice-matrice naif : 1.74353 secondes
MFlops -> 1235.3
gustavo@gustavo-X510URR:~/Documentos/facul/ENSTA/OS202/Course2023/TravauxDirigés/TD_numero_1/sources$ ./TestProductMatrix.exe 1026
Test passed
Temps CPU produit matrice-matrice naif : 1.75579 secondes
MFlops -> 1230.27
gustavo@gustavo-X510URR:~/Documentos/facul/ENSTA/OS202/Course2023/TravauxDirigés/TD_numero_1/sources$ ./TestProductMatrix.exe 1027
Test passed
Temps CPU produit matrice-matrice naif : 1.85851 secondes
MFlops -> 1165.67
gustavo@gustavo-X510URR:~/Documentos/facul/ENSTA/OS202/Course2023/TravauxDirigés/TD_numero_1/sources$
```

Les temps obtenus correspondent au temps que le programme a utilisé pour calculer la multiplication matricielle.

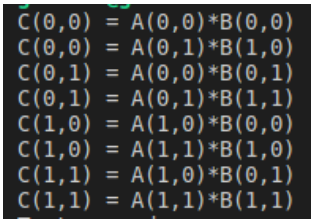
On peut voir une différence assez grande quand on fait la multiplication entre deux matrices de taille 1024 et elle est causée par le fait que l'espace utilisé pour stocker la matrice est exactement la même taille utilisée pour stocker le prochain nombre de la matrice. En d'autres termes, quand le programme lit la ligne de la matrice pour faire la multiplication, il enregistre le nombre dans exactement le même addresse destiné pour le prochain nombre. De cette manière on n' utilisera pas la capacité de cache et on aura besoin de faire le changement tout les fois. Ce qui gère cette quantité de temps pour calculer le produit.

1.2. La configuration que rend le calcul le plus efficace est:

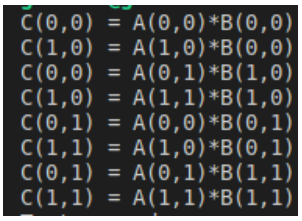
```
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {
  printf("EAE ");

  for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); j++)
    for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); k++)
      for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)
        C(i, j) += A(i, k) * B(k, j);
}
```

Avec un temps d'exécution pour une matrice 1023 de 1.15seg (en savant qui pour la configuration i-j-k, une matrice 1023 c'était 1.80seg)
et celui a une relation directe avec le code pour définir le classe matrice, parce que les matrices A et B qu'ont été créées sont stockées par colonnes. Alors, lorsqu'on change la façon de lire les matrices, en faisant par colonnes (j-k-i comme se voie dans le figure ci-dessus), le cache n'aura pas besoin de faire trop des réécriture, une fois qu'ils sont déjà disponibles dans le cache et il ne doit pas recharger les valeurs pour faire l'opération.



Boucles ijk



Boucles jki

1.3. A l'aide d'OpenMP et le command `#pragma omp parallel for num_threads("Mettre le nombre des threads")`, qui permet au compilateur de créer un nombre spécifique de threads pour une boucle donnée. La boucle "for" sera alors exécutée par le nombre spécifié de threads, ce qui peut améliorer la performance et la vitesse d'exécution du programme. Les résultats sont:

- ➔ Avec 2 threads, le temps obtenu c'était 0.65 seg
- ➔ Avec 4 threads, le temps obtenu c'était 0.45 seg
- ➔ Avec 8 threads, le temps obtenu c'était 0.33 seg

Entre plus threads, le temps obtenu diminue parce que le "for" est divisé en plusieurs parties et chaque processus pour s'exécuter indépendamment.

1.4. Il est possible d'améliorer le résultat obtenu avec la parallélisation, car la parallélisation peut être utilisée pour répartir le travail entre plusieurs threads, ce qui permet d'accélérer le traitement des données, en optimisant les allocations caches pour lire les colonnes et en répartissant le travail de manière équilibrée entre les threads, le temps d'exécution peut être considérablement réduit.

1.5. On a ajouté 3 boucles pour parcourir les lignes et colonnes diviser par un constante déjà définie et comparer le temps qui prend si on fait varier la taille des blocs.

```
namespace {
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {

    //#pragma omp parallel for num_threads(2)
    //#pragma omp parallel for
    for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); j++){
     for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); k++){
      for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)
       {
       C(i, j) += A(i, k) * B(k, j);
       //std::cout<<"C("<<i<<","<<j<<") = "<<"A("<<i<<","<<k<<")*B("<<k<<","<<j<<")\n";
       }
      }
    }

}
//const int szBlock = 32;
}  // namespace

Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix C(A.nbRows, B.nbCols, 0.0);

  int szBlock = 16;
  for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock)
    for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock)
      for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock)
        prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);

  return C;
```

1.6. Résultats du temps qui a pris la matrice par bloc pour faire les calculs.

szblock = 8, matrice 1023x1023 -> 1.39 seconds
szblock = 16, matrice 1023x1023 -> 1.12 seconds
szblock = 32, matrice 1023x1023 -> 1.00 seconds
szblock = 64, matrice 1023x1023 -> 0.90 seconds
szblock = 128, matrice 1023x1023 -> 0.84 seconds
szblock = 256, matrice 1023x1023 -> 0.79 seconds

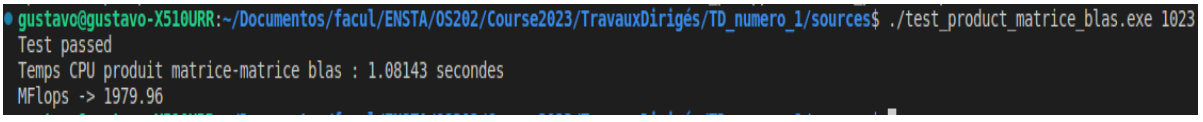
Etant donné que le temps de calcul avant sans le parallélisation était de 1.15 secondes, on peut voir que le produit matrice-matrice par bloc c'est plus efficace que le "scalaire". Si la matrice est divisée plusieurs fois, les opérations vont s'effectuer plus rapidement a cause que le cache va stocker moins des données et ils vont être plus disponibles pour le calcul.

1.7. Résultats du temps qui a pris la parallélisation du produit matrice-matrice par bloc pour faire les calculs.

En utilisant le szblock = 256, matrice 1023x1023
num_threads = 2, 0.44 seconds
num_threads = 4, 0.33 seconds
num_threads = 8, 0.37 seconds
num_threads = 16, 0.33 seconds

Comme prévu, la parallélisation est plus efficace en distribuant les processus sur plusieurs threads qui permet de travailler simultanément et ainsi réduire le temps de traitement. De plus, il avait l'amélioration de matrice par bloc, alors, il était déjà optimisé.

1.8. La comparaison avec blas nous a donné un temps moins efficace que les codes optimisant précédemment.



2.1. On a utilisé le communicateur `MPI.COMM_WORLD` qui représente un système de processeurs qui peuvent communiquer entre eux avec des commandes MPI. Pour le programme de circulation d'un jeton dans un anneau, on a fait une séquence des fonctions conditionnelles "if" pour évaluer chaque le 3 cases: Le premier jeton, le dernier jeton qui envoie les données au processus 0 et les jetons intermédiaires.

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank() #Le nombre des processus
size = comm.Get_size() #Nombre total des processus dans le communicateur

if rank == 0:
    print("Hi, I am", rank)
    data = 1 #jeton
    req = comm.isend(data, dest=1, tag=1) #Envoie les données a l'autre processus
    req.wait()
    end_process = comm.irecv(source=size-1,tag=1) #Recevoir jetons finals
    data = end_process.wait()
    print(f"Hi, am I  {rank}, Result:  {data}")

elif(rank == size - 1): #Dernier processus
    print("Hi, I am", rank)

    req = comm.irecv(source=rank-1, tag=1)
    data = req.wait()
    data = data + 1
    req = comm.isend(data, dest=0, tag=1) #Envoie jusqu'à processus 0

else:
    print("Hi, I am", rank)
    req = comm.irecv(source=rank-1, tag=1)
    data = req.wait()
    data = data + 1
    req = comm.isend(data, dest=rank+1, tag=1)

```

Résultats avec size = 4.

```
gustavo@gustavo-X510URR:~
Hi, I am 1
Hi, I am 2
Hi, I am 3
Hi, I am 0
Hi, am I  0, Result:  4
```

2.2. Pour le calcul très approché de pi, on a généré des points aléatoires pour remplir l'espace et chaque processus va compter ses points afin de passer les données au processus suivant et le prochain doit considérer les points déjà comptes pour le processus precend.

```python
import random
import math
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank() #process rank
size = comm.Get_size() # n process

#function pour generer des points
def generate_point():
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)
    return (x,y)

#function pour calculer le distances
def dist(p1, p2):
    return math.sqrt(pow((p1[0]-p2[0]), 2) + pow((p1[1]-p2[1]), 2))

#function pour savoir si c'est dans le circle ou pas
def in_circle(p):
    return dist(p, (0,0)) <= 1

#nombre de points dans le circle
n_in = 0
#nombre de iterations
reps = 10000
#calcule de plusieurs points
for i in range(reps):
    if(in_circle(generate_point())):
        n_in = n_in + 1

#le processus 0 va faire son calcul et recevoir tout le donnes et faire le calcule final
if rank == 0:
    all_in = n_in
    n_points = reps
    print(f"Hi, I am {rank} my pi is = {4 * all_in/n_points}")
    for i in range(1,3):
        end_process = comm.irecv(source=i,tag=1)
        data = end_process.wait()
        all_in = all_in + data[0]
        n_points = n_points + data[1]

    print(f"Hi, am I  {rank}, this is the result:  {4 * all_in/n_points}")

#les autres processus vont faire le calcul et envoier a le processus 0
else:
    print(f"Hi, I am {rank} my pi is = {4 * n_in/reps}")
    data = (n_in, reps)
    req = comm.isend(data, dest=0, tag=1)

```

Résultat avec size = 4.

```
Hi, I am 3 my pi is = 3.1304
Hi, I am 0 my pi is = 3.1496
Hi, I am 1 my pi is = 3.1364
Hi, I am 2 my pi is = 3.1544
Hi, am I  0, this is the result:  3.1468
```