OS202 - Programming Parallel Computers

Travail Dirigée

12 septembre 2024

Guilherme Nunes Trofino
2022-2024

Table des matières

1. Introduction

Repository Hello! My name is Guilherme Nunes Trofino and this is my LaTeX notebook of OS202 - Programming Parallel Computers that can be found in my GitHub repository : https://github.com/tr0fin0/classes_ensta.

Disclaimer This notebook is made so it may help others in this subject and is not intend to be used to cheat on tests so use it by your on risk.

Suggestions If you may find something on this document that does not seam correct please reach me by e-mail : guitrofino@gmail.com.

1.1. Information Matier

Référence Dans cette matière le but sera de comprendre . Ce travail est sur https://github.com/ avec l'objectif d'étudier et démontrer l'augmentation de performance quand on utilise la programmation parallèle.

1.2. Caracteristiques Ordinateur

CPU On utilisé le commande `lscpu` pour avoir des informations sur le processeur de mon ordinateur en retournant le suivant :

```
Architecture:            x86_64
    CPU op-mode(s):          32-bit, 64-bit
    Address sizes:           39 bits physical, 48 bits virtual
    Byte Order:              Little Endian
    CPU(s):                  20
    On-line CPU(s) list:     0-19
    Vendor ID:               GenuineIntel
Model name:            12th Gen Intel(R) Core(TM) i7-12700H
    CPU family:          6
    Model:               154
    Thread(s) per core:  2
    Core(s) per socket:  14
    Socket(s):           1
    Stepping:            3
    CPU max MHz:         4700.0000
    CPU min MHz:         400.0000
```

On peut voir qui mon ordinateur a, théoriquement, 20 CPU's disponibles avec les mémoires suivants :

```
Caches (sum of all):
    L1d:     544  KiB    (14 instances)
    L1i:     704  KiB    (14 instances)
    L2:      11.5 MiB    ( 8 instances)
    L3:      24   MiB    ( 1 instance)
```

Ces données seront utilisés pour l'analyse des performances.

2. Produit Matrice-Matrice

Question 1

Résolution. Les tailles suivants on était essayés :

n	secondes	MFloops
1023	1.17911	1815.94
1024	2.89563	741.63
1025	1.22712	1755.14
2047	9.9369	1726.37
2048	33.1686	517.956
2049	10.2921	1671.68
avg	9.7832	1371.45

TABLE 2.1 : `ijk`

On note qu'il y a une grand différence entre l'exécution avec une matrix de taille égale à une puissances de 2.

Ce comportement peut être justifié avec la façon que la mémoire est gérer pour la CPU et pour la construction de la mémoire.

Quand on utilise une variable à la position `i` c'est commun d'utiliser la variable à la position `i+1`. Le CPU considère ce principe et enregistre des variables en sequence.

La mémoire est construit à partir des structures binaires donc elle aura une taille multiple de 2. Son addressage sera fait à partir du module de la taille de la mémoire.

Quand il y a une matrice d'une taille multiple de 2, le module se rendre toujours au même endroit et donc chaque fois qui le CPU veut enregistre une variable il faut recopier tous les données en prenant plus de temps.

Question 2

Résolution. On considère que la multiplication de 2 matrices sera fait avec les variables `i j k` :

```
C(i, j) += A(i, k) * B(k, j)
```

Les tableaux suivants représentent le temps necessaires pour calculer la multiplication avec des différents configurations de `loop` :

n	secondes	MFloops
1023	1.17911	1815.94
1024	2.89563	741.63
1025	1.22712	1755.14
2047	9.9369	1726.37
2048	33.1686	517.956
2049	10.2921	1671.68
avg	9.7832	1371.45

TABLE 2.2 : `ijk`

n	secondes	MFloops
1023	1.26207	1696.58
1024	3.01302	712.735
1025	1.37956	1561.21
2047	23.6272	726.057
2048	70.8302	242.55
2049	24.7906	694.016
avg	20.8171	938.86

TABLE 2.3 : `jik`

n	secondes	MFloops
1023	0.394346	5429.74
1024	0.407676	5267.63
1025	0.416262	5174.1
2047	4.91225	3492.23
2048	5.02442	3419.27
2049	5.12015	3360.27
avg	2.7125	4357.21

TABLE 2.4 : `jki`

n	secondes	MFloops
1023	2.38203	898.897
1024	7.31333	293.64
1025	2.13751	1007.61
2047	76.2133	225.088
2048	109.086	157.489
2049	73.7907	233.16
avg	45.1538	469.31

TABLE 2.5 : `ikj`

n	secondes	MFloops
1023	2.27681	940.437
1024	7.27702	295.105
1025	2.52188	854.038
2047	62.4818	274.555
2048	132.456	129.702
2049	73.0916	235.39
avg	46.6842	454.87

TABLE 2.6 : `kij`

n	secondes	MFloops
1023	0.571053	3749.56
1024	0.462365	4644.56
1025	0.454393	4739.91
2047	5.77844	2968.75
2048	5.63255	3050.11
2049	5.69804	3019.47
avg	3.0995	3695.39

TABLE 2.7 : `kji`

L'ordre `jki` était la plus efficace : les operations on pris moins de temps et la différence entre une matrice de taille multiple de 2 et une autre matrice n'était pas très significative.

Comment précise pour la question précédent les données de la mémoire sont enregistres en groupe donc l'ordre entre lignes et colognes va influencer le résultat.

Code considère :

```
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);
```

On peut voir :

1. `i` représente les lignes ;
2. `j` représente les colognes ;

Comment l'algorithme est plus rapide quand l'interation "plus frequente" est sur `i` ça veut dire que les données sont estoquées pour lignes.

Question 3

Résolution. Les tableaux suivants représentent le temps necessaires pour calculer la multiplication avec l'ordre la plus efficace du `loop`, `jki`, et une quantité n de threads représentes par `[n]` :

n	secondes	MFloops
1023	0.0970206	22069.5
1024	0.101690	21118.0
1025	0.110316	19523.8
2047	0.557667	30761.6
2048	0.455107	37749.1
2049	0.491632	34995.8
avg	0.3022	27702.97

TABLE 2.8 : `jki [1]`

n	secondes	MFloops
1023	0.102490	20891.8
1024	0.0908985	23625.1
1025	0.119587	18010.2
2047	0.497154	34505.9
2048	0.570854	30095.0
2049	0.500602	34368.7
avg	0.3149	26916.12

TABLE 2.9 : `jki [4]`

n	secondes	MFloops
1023	0.0997595	21463.6
1024	0.119725	17936.8
1025	0.105947	20328.8
2047	0.513362	33416.4
2048	0.471373	36446.5
2049	0.475105	36213.1
avg	0.2975	27634.2

TABLE 2.10 : `jki [8]`

n	secondes	MFloops
1023	0.0961568	22267.8
1024	0.0929003	23116.0
1025	0.0865847	24874.8
2047	0.513362	31964.4
2048	0.479124	35856.8
2049	0.486361	35375.1
avg	0.2924	28909.15

TABLE 2.11 : `jki [12]`

n	secondes	MFloops
1023	0.0848282	25241.6
1024	0.0990637	21677.8
1025	0.121633	17707.3
2047	0.513362	31244.3
2048	0.439260	39110.9
2049	0.483793	35561.8
avg	0.2903	28423.95

TABLE 2.12 : `jki [16]`

n	secondes	MFloops
1023	0.0998027	21454.3
1024	0.110846	19373.5
1025	0.108874	19782.3
2047	0.568054	30199.1
2048	0.473769	36262.1
2049	0.493240	34881.7
avg	0.3091	26992.17

TABLE 2.13 : `jki [20]`

On peut voir que l'utilisation de threads a rendu l'exécution du programme : 8.9758, 8.6138, 9.1176, 9.2767, 9.3438 et 8.7755 fois plus rapide pour 1, 4, 8, 12, 16 et 20 threads respectivement.

Quand une nouvelle thread est crée on gagne la capacité de faire plusieurs calcules au même temps sur différents parties du processeur au prix d'avoir moins de mémoire pour chaque thread car la quantité de mémoire disponible reste constant.

De cette façon, on peut voir que 16 était la configuration la plus rapide pour faire les calcules.

Code considère :

```
#pragma omp parallel  // parallel declaration
{
int i, j, k;
omp_set_num_threads(8);
#pragma omp for     // declare for function as parallel
for (j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); j++)
  for (k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); k++)
    for (i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)
      C(i, j) += A(i, k) * B(k, j);
const int szBlock = 32;
}
```

Informations utiles étaient prises de le site suivant : https://stackoverflow.com/

Question 4

Résolution. C'est bien sur possible d'avoir une amélioration encore plus significative car ici on a optimisé une parametre à la fois sans considérer tous combinations possibles pour chaque variable.

Question 5

Résolution. Les tableaux suivants représentent le temps necessaires pour calculer la multiplication avec des différents configurations de `loop` :

n	secondes	MFloops
1024	0.842497	2548.95
2048	7.283589	2358.71
avg	4.063043	2453.83

TABLE 2.14 : `mnjki [16]`

n	secondes	MFloops
1024	0.655394	3276.63
2048	5.342796	3215.52
avg	2.999095	3246.08

TABLE 2.15 : `mnjki [64]`

n	secondes	MFloops
1024	0.552740	3885.16
2048	4.578990	3751.89
avg	2.565865	3818.53

TABLE 2.16 : `mnjki [256]`

n	secondes	MFloops
1024	0.812334	2643.60
2048	7.745882	2217.94
avg	4.279108	2430.77

TABLE 2.17 : `nmjki [16]`

n	secondes	MFloops
1024	0.656772	3269.75
2048	5.463736	3144.34
avg	3.060254	3207.05

TABLE 2.18 : `nmjki [64]`

n	secondes	MFloops
1024	0.550865	3898.38
2048	4.661620	3685.39
avg	2.606242	3791.89

TABLE 2.19 : `nmjki [256]`

On peut voir que la configuration `nmjki` est l'ordre de `loop` la plus efficace et 256 comme taille de bloque.

Code considère :

```
{
int dim = std::max({A.nbRows, B.nbCols, A.nbCols});

int m, n, i, j, k;

for (m = 0; m < dim; m+=sizeBlock)
    for (n = 0; n < dim; n+=sizeBlock)
        for (j = n; j < n+sizeBlock; j++)
        for (k = 0; k < dim; k++)
            for (i = m; i < m+sizeBlock; i++)
                C(i, j) += A(i, k) * B(k, j);
}
```

Question 6

Résolution. On peut voir que l'utilisation des bloques rendre le programme aussi plus efficace que quand compare à la même configuration séquentielle sans parallélisme.

Ici seulement les tailles 1024 et 2048 étaient considères car, pour la division en bloques le code a besoin des tailles divisibles pour une valeur commun.

Question 7

Résolution. Les tableaux suivants représentent le temps necessaires pour calculer la multiplication avec des différents tailles de bloque représentes par `[s]` avec la configuration de `loop` la plus efficace, `nmjki` :

n	secondes	MFloops
1024	0.165874	12946.51
2048	1.252230	13719.42
avg	0.709052	13332.97

TABLE 2.20 : `nmjki [16]`

n	secondes	MFloops
1024	0.159252	13484.84
2048	1.128784	15219.81
avg	0.644018	14352.32

TABLE 2.21 : `nmjki [32]`

n	secondes	MFloops
1024	0.125556	17103.75
2048	0.950782	18069.19
avg	0.538169	17586.47

TABLE 2.22 : `nmjki [64]`

n	secondes	MFloops
1024	0.134669	15946.42
2048	0.674998	25451.72
avg	0.404833	20699.07

TABLE 2.23 : `nmjki [128]`

n	secondes	MFloops
1024	0.175878	12210.04
2048	0.771701	22262.34
avg	0.473790	17236.19

TABLE 2.24 : `nmjki [256]`

n	secondes	MFloops
1024	0.263427	8152.11
2048	1.064857	16133.50
avg	0.664142	12142.80

TABLE 2.25 : `nmjki [512]`

On peut voir que même avec l'utilisation des bloques l'exécution du code n'était pas plus efficace. Ici la quantité de threads utilise, 16, pourrait-être pas idéale pour le système.

Code considère :

```
#pragma omp parallel        // parallel declaration
{
int dim = std::max({A.nbRows, B.nbCols, A.nbCols});

int m, n, i, j, k;
omp_set_num_threads(16);

#pragma omp for
for (n = 0; n < dim; n+=sizeBlock)
    for (m = 0; m < dim; m+=sizeBlock)
        for (j = n; j < n+sizeBlock; j++)
            for (k = 0; k < dim; k++)
                for (i = m; i < m+sizeBlock; i++)
                    C(i, j) += A(i, k) * B(k, j);
}
```

Question 8

Résolution. Le makefile n'arrivait pas à créer le .exe pour faire cette essayé.

3. Parallélisation MPI

3.1. Circulation

partie 2 tous les questions sont dans les examples et propablement sur youtube, pas trop difficile a faire

```python
from mpi4py import MPI


PROCESS_MAX      = 10
INTERATION_MAX   = 25

def serialWhile() -> None:
    interation   = 0
    process      = 0
    value        = 0


    while(interation < INTERATION_MAX):

        if (process+1) == PROCESS_MAX:
            process = 0
            print('='*31)
        else:
            print(f'process {process:2.0f} send {value:4.0f} to rank {process+1:2.0f}')
            process      += 1
            value        += 1
            interation   += 1


def parallelWhile(value: int):
    comm = MPI.COMM_WORLD   # instantice the communication world
    size = comm.Get_size()  # size of the communication world
    rank = comm.Get_rank()  # process ID ('rank')
    # multiple instances of this programming is running at the same time

    print(rank)
    if rank != 0:
        comm.send(value, dest=(rank+1))
        value = comm.recv(rank)
        print(f'process {rank:2.0f} send {value:4.0f} to rank {rank+1:2.0f}')
    else:
        print("="*20)

    MPI.Finalize()


def main():
    serialWhile()
    # parallelWhile(0)


main()
```

3.2. Calcul π

Résolution. Le calcul de π propose est expliqué et implémenté sur le vidéo : Inside Code.

On considère le code suivant :

```
import time
import numpy as np
import matplotlib.pyplot as plt
```

Observation, pour y accéder il suifit clique sur le code. En comparant les résultats de chaque exécution on aura :

```
    pi: 3.141774 error: 0.000182
    serial: 0.529775 s

    CPUs: 20
    pi: 3.148800 error: 0.007207
    parrel: 0.025772 s
```

Dans cette essayé 1e4 points étaient utilisés pour estimer Pi. On peut voir que le code parallèle a été 20.56 fois plus rapide que la version séquentielle.

```
    pi: 3.141522 error: -0.000070
    serial: 73.425146 s

    CPUs: 20
    pi: 3.161720 error: 0.020127
    parrel: 0.024069 s
```

Dans cette essayé 1e5 points étaient utilisés pour estimer Pi. On peut voir que le code parallèle a été 3050.61 fois plus rapide que la version séquentielle.

On peut voir que la version parallèle est beaucoup plus rapide en temps d'exécution par contre son erreur est plus significative car au lieu d'avoir une grande quantité de points chaque CPU aura une quantité réduit. On essayé la version de blocs avec 1e8 points et on trouve :

```
    CPUs: 20
    pi: 3.141279 error: -0.000313
    parrel: 0.625430 s
```

Maintenant, avec beaucoup plus de points par thread, on trouvé une résultat plus précis et avec un temps de calcule plus modeste.

3.3. Hypercube

Résolution.