***Oumy GAYE P28 2551 et Kossivi GNOZIGUE P33 6569***

**Exercice 19 : CHIMIE COMPUTATIONNELLE.**

Résumé en Chimie : INDICATION DU TRAVAIL A FAIRE

Nous avons développé deux algorithmes parallèles pour la construction de la matrice de Fock.

1. Première approche : réplication totale

○ Les matrices F et D sont répliquées dans chacune des N tâches.

○ Les calculs d’intégrales sont répartis entre ces tâches.

○ Un algorithme de sommation est utilisé pour additionner les contributions à la matrice F accumulées dans les différentes tâches.

○ Cet algorithme est simple, mais non évolutif (non scalable).

2. Deuxième approche : réplication partielle et agglomération

○ Les matrices F, D et A sont partagées entre les N tâches, avec une petite quantité de réplication.

○ Les calculs d’intégrales sont agglomérés en un certain nombre de tâches de calcul, chacune contenant plusieurs intégrales.

○ Ces tâches sont ensuite affectées aux processeurs, soit de manière statique, soit via un ordonnancement dynamique.

Analyse des compromis : Cette étude de cas met en lumière certains arbitrages dans la conception des algorithmes :

● Le premier algorithme réduit drastiquement les coûts de communication et de développement logiciel, mais n’est pas évolutif.

● Le second algorithme a un coût de communication plus élevé, mais il est très évolutif : ses besoins en mémoire augmentent uniquement avec la taille du problème, et non avec le nombre de processeurs. Pour choisir entre ces deux algorithmes, il est nécessaire de quantifier leurs performances parallèles et d’évaluer l’importance de l’évolutivité, en fonction des exigences de l’application et des caractéristiques de l’ordinateur parallèle cible.

**REPLICATION TOTALE**

#include <stdio.h>

#include <stdlib.h>

#include <stddef.h>

#include <mpi.h>

#include<time.h>

#define N 100

void init\_matrix(double\* mat, int size) {

for (int i = 0; i < size; i++) {

mat[i] = (double) rand() / RAND\_MAX;

}

}

int main(int argc, char\*\* argv) {

int rank, size;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

srand(time(NULL) + rank); // Seed unique par processus

double \*D = malloc(N \* N \* sizeof(double));

double \*A = malloc(N \* N \* N \* N \* sizeof(double)); // Simplification en 1D

double \*F\_local = calloc(N \* N, sizeof(double));

double \*F\_total = NULL;

if (rank == 0) {

init\_matrix(D, N);

for (int i = 0; i < N\*N\*N\*N; i++) {

A[i] = (double) rand() / RAND\_MAX;

}

}

MPI\_Bcast(D, N\*N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(A, N\*N\*N\*N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

int rows\_per\_proc = N / size;

int start\_row = rank \* rows\_per\_proc;

int end\_row = (rank == size - 1) ? N : start\_row + rows\_per\_proc;

double start\_time = MPI\_Wtime();

for (int i = start\_row; i < end\_row; i++) {

for (int j = 0; j < N; j++) {

double sum = 0.0;

for (int k = 0; k < N; k++) {

for (int l = 0; l < N; l++) {

int idx = ((i \* N + j) \* N + k) \* N + l;

sum += D[k \* N + l] \* A[idx];

}

}

F\_local[i \* N + j] = sum;

}

}

if (rank == 0) F\_total = calloc(N \* N, sizeof(double));

MPI\_Reduce(F\_local, F\_total, N \* N, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

double end\_time = MPI\_Wtime();

if (rank == 0) {

printf("Temps d'exécution (réplication totale) : %f secondes\n", end\_time - start\_time);

}

free(D); free(A); free(F\_local);

if (rank == 0) free(F\_total);

MPI\_Finalize();

return 0;

}

**REPLICATION PARTIELLE**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <time.h>

#include <stddef.h>

#define N 100 // Taille de la matrice (modifiable)

int main(int argc, char\*\* argv) {

int rank, size;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

srand(time(NULL) + rank);//pour avoir des donnees deffirentes dans chaque processeur

int nbre\_lign\_proc = N / size;

int debut\_lign = rank \* nbre\_lign\_proc;

int fin\_lign = (rank == size - 1) ? N : debut\_lign + nbre\_lign\_proc;

double \*D\_local = malloc(N \* nbre\_lign\_proc \* sizeof(double));//: chaque processus n’a que les lignes de D qui l’intéressent

double \*A = malloc(N\*N\*N\*N \* sizeof(double));

double \*F\_local = calloc(N \* nbre\_lign\_proc, sizeof(double));// résultat local pour les lignes calculées

double \*F\_total = NULL;

if (rank == 0) {

for (int i = 0; i < N\*N\*N\*N; i++) A[i] = (double) rand() / RAND\_MAX;

}

MPI\_Bcast(A, N\*N\*N\*N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

for (int i = 0; i < N \* nbre\_lign\_proc; i++) {

D\_local[i] = (double) rand() / RAND\_MAX;

}

double debut = MPI\_Wtime();

for (int a = 0; a< nbre\_lign\_proc; a++) {

int i = debut\_lign + a;

for (int j = 0; j < N; j++) {

double sum = 0.0;

for (int k = 0; k < N; k++) {

for (int l = 0; l < N; l++) {

int idx = ((i \* N + j) \* N + k) \* N + l;

sum += D\_local[a \* N + k] \* A[idx];

}

}

F\_local[a \* N + j] = sum;

}

}

if (rank == 0) F\_total = calloc(N \* N, sizeof(double));

MPI\_Gather(F\_local, N \* nbre\_lign\_proc, MPI\_DOUBLE, F\_total, N \* nbre\_lign\_proc, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

double Fin = MPI\_Wtime();

if (rank == 0) {

printf("Temps d'exécution (réplication partielle) : %f secondes\n", Fin- debut);

}

free(D\_local); free(A); free(F\_local);

if (rank == 0) free(F\_total);

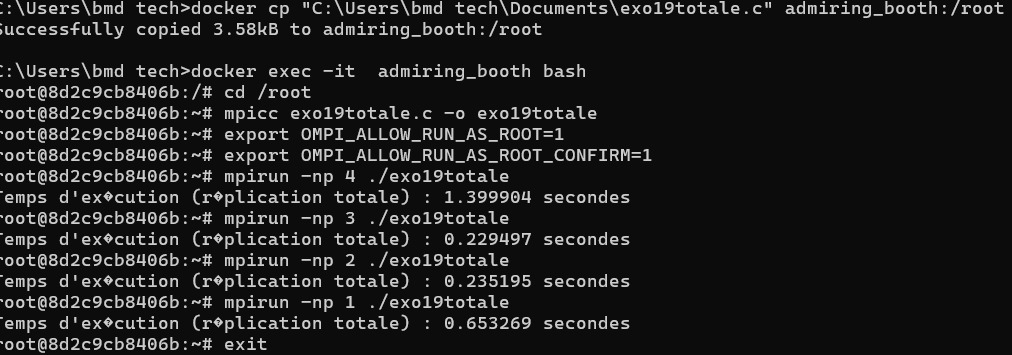
MPI\_Finalize();

return 0;

}

**RESULTATS APRES EXECUTION** :

REPLICATION TOTALE



REPLICATION PARTIELLE

