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

# **Exercice 8. Agglomération et Optimisation de la Concurrence**

### **Cas d’étude : Calcul de la somme d’un grand tableau en parallèle**

### **Instructions :**

1. Implémentez une version naïve de la somme parallèle.
2. Optimisez cette version en appliquant l’agglomération et en limitant les communications inutiles.
3. Expérimentez avec différents nombres de tâches et de processeurs.
4. Analysez l’impact sur le temps d’exécution et la communication.

### **Questions :**

1. Quelle est la réduction du nombre de communications après agglomération ?
2. Comment choisir la taille optimale d’une tâche après regroupement ?
3. Comment l’agglomération affecte-t-elle l’évolutivité de l’algorithme ?
4. Peut-on appliquer cette technique à d’autres algorithmes de calcul parallèle ?

**IMPLEMENTATION DE LA METHODE NAÏVE**

#include <stdio.h>

#include <assert.h>

#include <time.h>

#include <stdlib.h>

#include <mpi.h>

#define N 10000000 // Taille du tableau

const MAX\_NUMBS = 100; // Valeur maximale des éléments du tableau

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

int rank, size;

int\* tab = NULL; // Stocké uniquement par le maître

int\* tab\_esclave = NULL; // Partie traitée par chaque esclave

int taille\_tab\_esclave;

int somme\_esclave = 0; // Stocke la somme que chaque esclave a trouvé

int somme\_totale = 0; // Stocke la somme totale obtenue par le maître

double temps\_debut, temps\_fin; // Variables pr lancer et arrêter le chrono

MPI\_Init(&argc, &argv);

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

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

// Amorcez le générateur de nombres aléatoires pour obtenir des résultats différents à chaque fois

srand(time(NULL));

// Le maître initialise le tableau global

if (rank == 0) {

tab = (int\*)malloc(N \* sizeof(int)); //allocation mémoire

int i;

for (i = 0; i < N; i++) {

tab[i] = (rand()% MAX\_NUMBS+1); //Nbre entre 0 et 100

}

temps\_debut = MPI\_Wtime();

taille\_tab\_esclave = N / (size - 1); // Taille des tableaux que chaque esclave doit considérer.

}

// Envoie de la taille des sous-tableaux à tous les esclaves par la diffusion

MPI\_Bcast(&taille\_tab\_esclave, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Allocation mémoire pour les sous-tableaux (esclaves)

if (rank != 0) {

tab\_esclave = (int\*)malloc(taille\_tab\_esclave \* sizeof(int));

}

// Le maître envoie les données aux esclaves

if (rank == 0) {

int i;

for (i = 1; i < size; i++) {

int debut = (i - 1) \* taille\_tab\_esclave; //Le debut du tableau pour chaque esclave

// Envoie bloquant des sous tableaux à chaque esclave (communication synchrone)

MPI\_Send(&tab[debut], taille\_tab\_esclave, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

}

} else {

// Réception des sous tableaux par chaque esclave

MPI\_Recv(tab\_esclave, taille\_tab\_esclave, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

// Calcul de la somme par chaque esclaves

if (rank != 0) {

somme\_esclave = 0;

int i;

for (i = 0; i < taille\_tab\_esclave; i++) {

somme\_esclave += tab\_esclave[i];

}

printf("L'esclave %d a trouvé une somme de : %d\n", rank, somme\_esclave);

}

// Les esclaves envoient leur résultat au maître

if (rank != 0) {

MPI\_Send(&somme\_esclave, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

}

// Le maître somme les résultats reçus

if (rank == 0) {

somme\_totale = 0;

int i;

for (i = 1; i < size; i++) {

int temp;

MPI\_Recv(&temp, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

somme\_totale += temp;

}

// Gestion du reste (si N n'est pas divisible par (size-1))

int reste = N % (size - 1);

if (reste != 0) {

int i;

for (i = N - reste; i < N; i++) {

somme\_totale += tab[i];

}

}

temps\_fin = MPI\_Wtime();

printf("La somme totale est : %d\n", somme\_totale);

printf("Le temps d'exécution est de %f secondes\n", temps\_fin - temps\_debut);

}

// Nettoyer les tableaux

if (rank == 0) {

free(tab);

} else {

free(tab\_esclave);

}

MPI\_Finalize();

return 0;

}

**OPTIMISATION DU CODE EN APPLIQUANT L’AGGLOMERATION ET EN REDUISANT LA COMMUNICATION**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <time.h>

#define N 10000000 // Taille du tableau

const MAX\_VALUE = 100; // Valeur maximale des éléments du tableau

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

int rank, size;

int \*tab = NULL; // Tableau créé par le maître

int \*tab\_esclave = NULL; // Sous tableau affecté aux esclaves

int taille\_tab\_esclave; // Taille des sous tableau

int somme\_esclave = 0; // Somme des éléments pour chaque esclave

int somme\_totale = 0; // Somme totale calculée par le maître

double temps\_debut, temps\_fin;

MPI\_Init(&argc, &argv);

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

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

// Le maître initialise le tableau global

if (rank == 0) {

tab = (int \*)malloc(N \* sizeof(int));

srand(time(NULL));

int i;

for (i = 0; i < N; i++) {

tab[i] = rand() % MAX\_VALUE + 1; // générer des nombres aléatoires entre 0 et 100

}

temps\_debut = MPI\_Wtime();

}

// Calcul des tailles pour les esclaves et des décalages avec gestion du reste

int \*NbreEnvoi = (int \*)malloc(size \* sizeof(int));

int \*deplacement = (int \*)malloc(size \* sizeof(int));

int reste = N % size;

int sum = 0;

int i;

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

NbreEnvoi[i] = N / size + (i < reste ? 1 : 0); //si i <reste, on ajoute 1 sinon on ajoute 0 à N/size

deplacement[i] = sum;

sum += NbreEnvoi[i];

}

taille\_tab\_esclave = NbreEnvoi[rank];

tab\_esclave = (int \*)malloc(taille\_tab\_esclave \* sizeof(int)); // Allocation du sous-tableau local

// Distribution avec MPI\_Scatterv (gère les tailles inégales)

MPI\_Scatterv(tab, NbreEnvoi, deplacement, MPI\_INT, tab\_esclave, taille\_tab\_esclave, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Somme locale (chaque processus traite son sous-bloc)

if (rank != 0) {

somme\_esclave = 0;

int i;

for (i = 0; i < taille\_tab\_esclave; i++) {

somme\_esclave += tab\_esclave[i];

}

printf("L'esclave %d a trouvé une somme de : %d\n", rank, somme\_esclave);

}

// Réduction en arbre (MPI\_Reduce utilise une optimisation similaire)

MPI\_Reduce(&somme\_esclave, &somme\_totale, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// Affichage du résultat (maître)

if (rank == 0) {

temps\_fin = MPI\_Wtime();

printf("La somme totale est : %d\n", somme\_totale);

printf("Le temps d'exécution est de %f secondes\n", temps\_fin - temps\_debut);

}

// Nettoyer les tableaux

if (rank == 0) free(tab);

free(tab\_esclave);

free(NbreEnvoi);

free(deplacement);

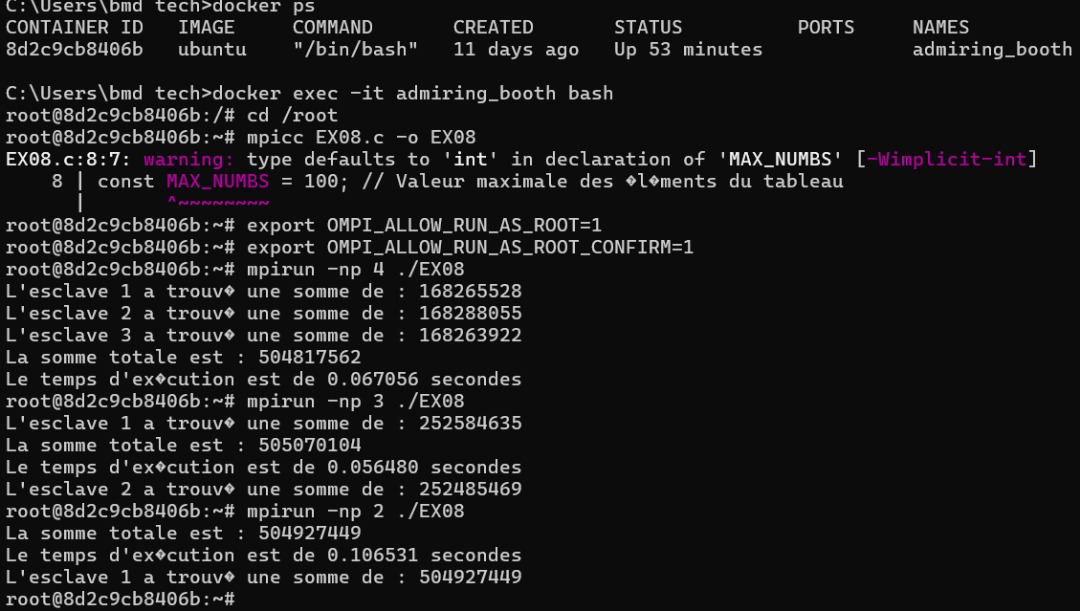
MPI\_Finalize();

return 0;

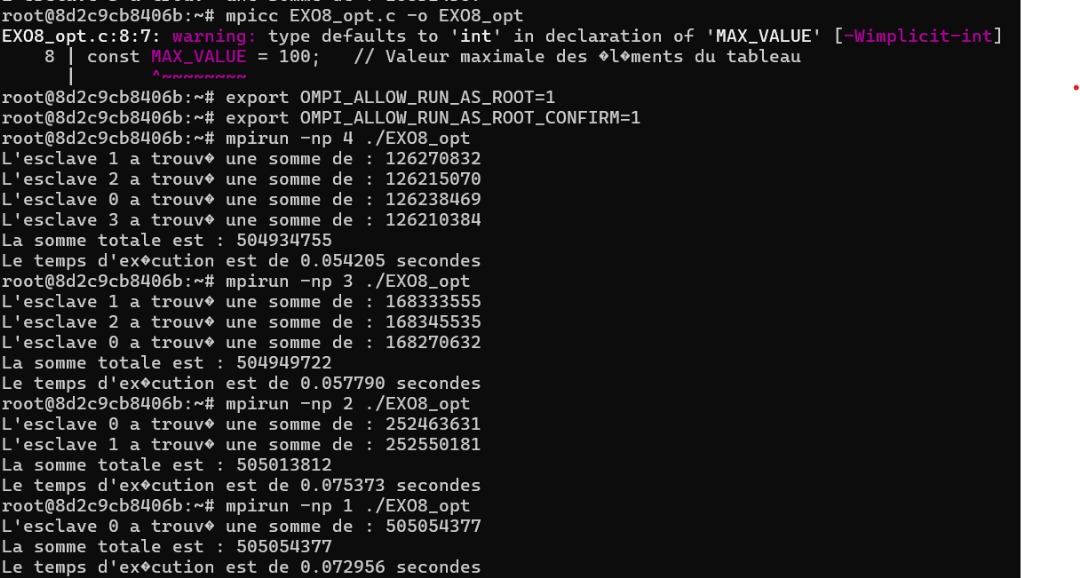
}

**EXPERIENCE AVEC DIFFERENTS NOMBRES DE TACHES ET DE PROCESSEURS**

Méthode naïve



Méthode optimisée :



**ANALYSE DE L’IMPACT SUR LE TEMPS D’EXECUTION ET LA COMMUNICATION**

Les temps d’exécution avec la méthode naïve :

Pour 4 processeurs : 0.067056 secondes

Pour 3 processeurs : 0.056480 secondes

Pour 2 processeurs : 0.106431 secondes

Nous remarquons que le temps d’exécution est très grand pour 2 processeurs par rapport aux deux autres temps (cas de 3 et 4 processeurs). En suite le temps d’exécution pour 4 processeurs est plus grand que celui de 3 processeurs. Lorsque nous avons 2 processeurs, il y n’y a qu’un seul esclave qui fait tout le calcul. On pourrait dire que le calcul n’est pas parallélisé raison pour laquelle il y a un long temps d’exécution. Donc la communication n’est pas forcément un facteur limitant. Dans les deux autres cas, le temps d’exécution dépend du type de communication. Plus on a de processeurs, plus le temps est long et ceci peut être dû à la synchronisation entre les processeurs lors de l’envoi et de la réception des données ; c’est-à-dire au type de communication entre les processeurs (MPI\_Send, MPI\_Recv). Il faut donc une optimisation pour que la communication ne soit plus un facteur limitant.

Les temps d’exécution avec la méthode optimisée :

Pour 4 processeurs : 0.054205 secondes

Pour 3 processeurs : 0.057790 secondes

Pour 2 processeurs : 0.075373 secondes

Pour 1 processeur : 0.072956 secondes

Nous remarquons de façon générale que plus le nombre de processeurs est grand, plus le temps d’exécution est rapide. La communication n’est plus un facteur limitant car elle a été optimisée grâce aux communications collectives de sorte que le parallélisme ne soit plus un problème.

**Reponses**

1. Le nombre de réduction pour 4 processeurs:

Pour la méthode naïve, en supposant un envoi non optimisé à travers MPI\_Bcast, nous avons 3 communication entre le maître et les esclaves sinon 2 (diffusion en anneau). S’ajoute à cela 6 communication à travers MPI\_Send et 6 autres à travers MPI\_Recv entre maître et esclaves. Au total il y a 15 communications pour les envois séquentiels des messages de diffusion et 14 sinon.

Pour la méthode d’agglomération, comme précédemment, le nombre de communication dépend du type de MPI. Pour un MPI optimisé, il y aura 2 communications à travers MPI\_Scatterv et 2 à travers MPI\_Reduce entre le maître et les esclaves ; ce qui donne 4 au total. Par contre pour un MPI naïf, il y aura une communication séquentielle donc 3 pour chaque fonction ce qui donne un total de 6 communications.

En définitive, le nombre de réduction dépend du type de MPI. Pour un MPI non séquentiel, il y a une réduction de 10 communications et 9 pour un MPI séquentiel.

1. La taille optimale a été choisie par rapport au nombre de processeurs. Le but est d’affecter approximativement, le même nombre de tâche à chaque processeur.
2. L’agglomération a permis d’améliorer le temps d’exécution en réduisant le nombre de communication, en équilibrant les tâches à réaliser par tous les processeurs y compris le maître.
3. Oui, on peut affecter cet algorithme à d’autres type de programmation parallèles.