



Cours : I3340

Année : 2018-2019

Durée : 2H

Examen : Ss2

Exercice 1

On veut paralléliser une fonction qui produit un histogramme pour une série de nombres entiers. L'extrait de code C suivant présente cette fonction.

```
/*
 * Fonction pour le calcul d'un histogramme.

 Données d'entrée:
 - elems: Tableau d'entiers non-négatifs (0 ≤ elems[i] ≤ valMax)
 - n: Taille du tableau elems
 - valMax = Valeur maximum parmi les éléments d'elems

 Résultat:
 - Pointeur vers le tableau (dynamique) de l'histogramme résultant
 */
int* genererHistogramme( int elems[], int n, int valMax )
{
    // On alloue le tableau pour l'histogramme résultant.
    int *histo = (int*) malloc( (valMax+1) * sizeof(int) );

    // On initialise l'histogramme.
    for( int i = 0; i <= valMax; i++ ) {
        histo[i] = 0;
    }

    // On construit l'histogramme en analysant les données.
    for( int i = 0; i < n; i++ ) {
        histo[elems[i]] += 1;
    }

    return histo;
}
```

Chaque nombre du tableau *elems* est un entier compris entre 0 et *valMax* (inclusivement). Soit alors *k* un entier compris inclusivement entre 0 et *valMax* et soit *histo* le tableau retourné par un appel à la fonction *histogramme(elems, n, valMax)*. On aura alors *histo[k]* = nombre d'occurrences de *k* dans *elems*. Par exemple, supposons qu'on ait les 12 valeurs suivantes et l'appel à la fonction *histogramme* comme suit :

```
elems = { 10, 1, 3, 3, 2, 9, 1, 1, 1, 3, 10 }
histo = histogramme( elems, 12, 10 )
```

Alors l'histogramme résultant serait comme suit :
histo = { 0, 4, 1, 4, 0, 0, 0, 0, 1, 2 }

Écrivez une version parallèle de la fonction histogramme en utilisant OpenMP.

Exercice 2

On veut écrire un programme MPI qui suppose l'existence de plusieurs processus où chaque processus possède une valeur entière générée d'une façon aléatoire, l'objectif est d'afficher le rang du processus qui possède la valeur **max** ainsi que le rang du processus qui possède la valeur **min**.

Par exemple, supposons l'existence de cinq processus où la valeur conservée par chacun est simplement le double de son numéro de processus (rang).

L'exécution produirait alors le résultat suivant :

Moi processus de rang 0 je possède la valeur min =0

Moi processus de rang 4 je possède la valeur max =8

Donner le programme MPI pour réaliser cette tâche et ce de deux façons différentes :

1. En utilisant des opérations de communication point à point, uniquement avec MPI_Send et MPI_Recv.
2. En utilisant des opérations de communication collectives, donc sans l'utilisation de MPI_Send et MPI_Recv.

Exercice 3

Soit n un entier positif. On considère de grosses matrices carrées, de taille $n \times n$, et on veut déterminer, pour chaque ligne et chaque colonne, la valeur maximum. De plus, ces maximums devront être connus d'un seul et unique processus, disons uniquement le processus de rang 0.

Par exemple, on a ci-dessous une matrice 9×9 avec, à droite et en bas, les différents maximums, qui doivent ultimement être tous connus du processus 0 :

15	23	34	36	74	81	91	11	2	91
46	74	24	14	95	52	31	42	18	95
21	29	38	33	34	32	35	36	37	38
48	55	69	77	86	11	14	22	13	86
24	22	36	58	65	73	77	79	71	79
71	21	13	29	22	24	28	25	26	71
79	76	71	75	73	37	22	18	14	79
2	8	7	4	1	9	16	13	5	16
93	24	25	22	28	26	21	27	29	93

93 76 71 77 95 81 91 79 71

On travaille sur une machine parallèle à mémoire distribuée qui comporte p processeurs. On veut écrire un programme MPI/C qui utilise uniquement des opérations collectives de communication, donc pas de communication point-à-point (pas de MPI_Send ou MPI_Recv).

Soit n la taille des matrices à traiter. On suppose que n est divisible par p ainsi que par \sqrt{p} , et que p soit un carré parfait, donc il existe k tel que $k^2 = p$. La valeur de n peut être très grande, donc n peut également être supérieur à p de plusieurs ordres de grandeur, $p \ll n$.

Q1. Présentez et comparez deux façons différentes pour résoudre ce problème, quels sont les avantages/désavantages, forces vs. faiblesses de différentes approches? Vous n'avez pas à écrire de vrai code MPI.

Vous devez simplement décrire les grandes lignes de vos approches. Notamment, il est important d'indiquer les opérations collectives qui seraient utilisées, mais sans nécessairement indiquer les détails de tous les arguments. Dans les deux solutions proposées il faut indiquer l'impact du choix de distribution des matrices sur le fonctionnement et l'efficacité. Concluez en indiquant l'approche que vous suggérez d'utiliser, de façon à bien exploiter les ressources de la machine.

Q2. Ecrire le code MPI/C pour l'approche que vous suggérez d'utiliser.

Bon travail

3.5 Réduction

```
int MPI_Reduce(
    const void * message_emis,
    void * message_recu,
    int longueur,
    MPI_Datatype type,
    MPI_Op operation,
    int rang_dest,
    MPI_Comm comm)
{
    operation = MPI_MAX | MPI_MIN | MPI_SUM | MPI_PROD |
    MPI_BAND | MPI_BOR | MPI_BXOR | MPI_BAND |
    MPI_LOR | MPI_LXOR;
```

Annexe Parallélisme Open MP

- `#pragma omp parallel` : Construction de la région parallèle.
- `#pragma omp parallel default(private) private(x)` : Changer le statut par défaut de tous les variables et de déterminer ce statut obligatoirement, `private(x)` : `x` sera trouver dans la pile de chaque thread.
- `#pragma omp parallel firstprivate(x)` : Initialiser `x` par la dernière valeur de `x` avant le parallèle.
- `#pragma omp parallel for lastprivate(x)` : Enregistrer la valeur de `x` de la dernière itération de la boucle `for` à la sortie du parallèle.
- `#pragma omp threadprivate(x)` : Pour privatiser la variable statique `x` dans la région parallèle.
 - `#pragma omp parallel copyin(x)` : Pour prendre la valeur de `x` (`x` : statique)
- `#pragma omp parallel num_threads(...)` : Pour déterminer le nombre des threads dans cette région.
- `#pragma omp parallel nowait` : Pour n'attendre pas tous les autres threads de terminer leur tâches.
- `#pragma omp parallel for schedule(static)` : Diviser les itérations en paquets de taille donnée (nb itérations/nb threads) et l'attribuer cycliquement à chaque thread.
- `#pragma omp parallel for schedule(dynamic,...)` : Division des itérations selon nous.
- `#pragma omp parallel schedule(runtime)` : division à l'aide du variable d'environnement.
- `#pragma omp parallel for ordered`: pour une exécution ordonnée.
 - `#pragma omp ordered` : Où appliquer l'ordre.
- `#pragma omp for reduction (+: s)`: Calcul partiel dans chaque thread puis synchronisation et calcul final du résultat.
- `#pragma omp sections nowait` : Partie exécutée par 1 seul thread.
 - `#pragma omp section`
- `#pragma omp single` : Exécuté par 1 seul thread (n'importe quel).
 - `#pragma omp single copyprivate(x)` : Enregistrer la valeur de `x` durant la partie parallèle.
- `#pragma omp master`: Exécuté par 1 seul thread (thread 0).
- `#pragma omp barriere` : Chaque thread attend tous les autres threads de terminer pour continuer ensemble.
- `#pragma omp atomic`: Protéger 1 variable, (accès protégé par 1 seul thread).
- `#pragma omp critical` : Protéger un ensemble des variables.

```
omp_get_thread_num();
omp_get_num_threads();
omp_get_wtime();

Pour random: #include<time.h>
    time_t t;
    srand((unsigned) time(&t));
    rand()....;
```

1 Environnement

1.1 Initialisation de l'environnement MPI

```
int MPI_Init(int * argc, char ***argv)
```

1.2 Rang du processus

```
int MPI_Comm_rank(MPI_Comm comm, int * rang)
```

1.3 Nombre de processus

```
int MPI_Comm_size(MPI_Comm comm, int * nb_procs)
```

1.4 Désactivation de l'environnement MPI

```
int MPI_Finalize(void)
```

1.5 Arrêt d'un programme MPI

```
int MPI_Abort(MPI_Comm comm, int error)
```

1.6 Prise de temps

```
double MPI_Wtime(void)
```

2 Communications point à point

2.1 Envoi de message

```
int MPI_Send(
    const void * message,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette,
    MPI_Comm comm)
```

2.2 Envoi non bloquant de message

```
int MPI_Isend(
    const void * message,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette,
    MPI_Comm comm,
    MPI_Request * requete)
```

3 Réception de message

```
int MPI_Recv(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    int etiquette,
    MPI_Comm comm,
    MPI_Status * statut)
```

Aide memoire MPI en C

2.4 Réception non bloquant de message

```
int MPI_Recv(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    MPI_Comm comm,
    MPI_Request * requete)
```

2.5 Envoi et réception de message

```
int MPI_Sendrecv(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    int rang_dest,
    void * message_recu,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_source,
    int etiquette_message_recu,
    MPI_Comm comm,
    MPI_Status * statut)
```

```
int MPI_Sendrecv_replace(
    void * message_emis_recu,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette_message_emis,
    int rang_source,
    int etiquette_message_recu,
    MPI_Comm comm,
    MPI_Status * statut)
```

2.6 Attente de la fin d'une communication non bloquante

```
int MPI_Wait(MPI_Request * requete, MPI_Status * statut)
```

2.7 Test de la fin d'une communication non bloquante

```
int MPI_Test(
    MPI_Request * requete,
    int drapeau,
    MPI_Status * statut)
```

3 Communications collectives

3.1 Diffusion générale

```
int MPI_Bcast(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    MPI_Comm comm)
```

3.2 Diffusion sélective

```
int MPI_Scatter(
    const void * message_a_repartir,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_recu,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_source,
    MPI_Comm comm)
```

3.3 Collecte

```
int MPI_Gather(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_dest,
    MPI_Comm comm)
```

```
int MPI_Allgather(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    MPI_Comm comm)
```

3.4 Collecte et diffusion

```
int MPI_Alltoall(
    const void * message_a_repartir,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    MPI_Comm comm)
```



Cours : I3340

Année : 2017-2018

Durée : 2H

Examen : Ss2

Exercice 1

On considère un tableau à deux dimensions $N * M$ qui contient la liste des notes des étudiants dans une classe. N représente le nombre des étudiants et M représente le nombre des matières pour chaque étudiant. On souhaite faire un programme qui permet de chercher l'indice de l'étudiant classé premier ainsi que son moyen général.

Afin de trouver l'étudiant classé premier sur une machine à p processeurs, on divise le tableau en p sous tableaux, chaque sous tableau contient N/p étudiants. On calcule la moyenne des étudiants dans chaque sous tableau, la moyenne la plus élevée ainsi que l'indice de l'étudiant seront toujours gardés par le processus qui fait le calcul. Lorsque les p processus terminent leur calcul, le programme principal récupère les résultats, il calcule ensuite la moyenne la plus élevée et l'indice de l'étudiant classé premier.

Le calcul au niveau de chaque partie se fait en parallèle. Chaque processus prend en charge le calcul pour une partie de ce sous tableau, pour cela il a besoin de connaître la référence du tableau supposé global, l'indice de début et l'indice de fin de ce sous tableau.

1. On considère une machine avec p processeurs, écrire le programme c openmp qui permet de chercher l'étudiant classé premier.
2. Modifier votre programme afin de faire la recherche d'une façon dynamique. Expliquer si cette nouvelle décomposition permet d'améliorer la performance de votre algorithme.

Exercice 2

On considère n processus possédant chacun un entier initialisé au lancement du processus d'une façon aléatoire. Le but du programme est de trouver la valeur max.

Communications point-à-point

1. Ecrire le programme mpi qui permet de trouver la valeur max par échange circulaire entre les processus ainsi le premier processus envoie son entier au processus 1 qui calcul la valeur max. Le processus i reçoit la valeur max du processus i-1 et l'envoie au processus i+1. Le programme se termine lorsque le processus zéro reçoit la valeur envoyée par le dernier processus. Le programme doit prendre un soin particulier pour s'assurer qu'il ne se bloque pas. En d'autres termes, le processus zéro s'assure qu'il a terminé son premier envoi avant d'essayer de recevoir la valeur du dernier processus. Tous les autres processus appellent MPI_Recv (pour la réception de leur processus inférieur voisin) et MPI_Send (pour l'envoi de la valeur à leur processus supérieur voisin) pour passer la valeur le long de l'anneau. MPI_Send et MPI_Recv se bloqueront jusqu'à ce que le message ait été transmis.

Communications collectives

Afin de faciliter le programme, on souhaite calculer la valeur max en utilisant la communication collective.

2. Ecrire le programme mpi qui calcule la valeur max en utilisant la fonction MPI_Gather.
3. Modifier le programme en utilisant la fonction MPI_Reduce.

Exercice 3

On souhaite faire un programme qui permet de calculer le minimum et le maximum des valeurs dans un tableau d'entier. Exemple : tab= {1, 3, 4, 6, 7, 8, 9, 121}, min=1 et max=121. Pour calculer ces deux valeurs sur une machine à p processeurs on divise le tableau en p sous tableaux de telle sorte que la recherche au niveau de chaque sous tableau se fait en parallèle. On suppose que le deux variables min et max sont deux variables partagés. Chaque processus prend en charge le calcul pour un sous tableau, pour cela il a besoin de connaître la référence du tableau supposé global et le deux variables partagés min et max.

1. Ecrire le programme C OpenMP afin de calculer d'une façon parallèle le deux valeurs min et max.

On souhaite utiliser plusieurs machines pour la recherche de min et max.

2. Ecrire le programme c (mpi) qui permet de paralléliser la recherche de min et max en utilisant les fonctions MPI_Scatter et MPI_Gather.
3. Modifier le programme qui permet de paralléliser la recherche de min et max en utilisant les fonctions MPI_Scatter et MPI_Reduce.

Bon travail

3.5 Réduction

```
int MPI_Reduce(
    const void * message_emis,
    void * message_recu,
    int longueur,
    MPI_Datatype type,
    MPI_Op operation,
    int rang_dest,
    MPI_Comm comm)
operation = MPI_MAX | MPI_MIN | MPI_SUM | MPI_PROD |
            MPI_BAND | MPI_BOR | MPI_BXOR | MPI_BAND |
            MPI_LOR | MPI_LBXOR
```

Annexe Parallélisme Open MP

- #pragma omp parallel : Construction de la région parallèle.
- #pragma omp parallel default(private) private(x) : Changer le statut par défaut de tous les variables et de déterminer ce statut obligatoirement, private(x) : x sera trouver dans la pile de chaque thread.
- #pragma omp parallel firstprivate(x) : Initialiser x par la dernière valeur de x avant le parallèle.
- #pragma omp parallel for lastprivate(x) : Enregistrer la valeur de x de la dernière itération de la boucle for à la sortie du parallèle.
- #pragma omp threadprivate(x) : Pour privatiser la variable statique x dans la région parallèle.
 - #pragma omp parallel copyin(x) : Pour prendre la valeur de x (x : statique)
- #pragma omp parallel num_threads(...) : Pour déterminer le nombre des threads dans cette région.
- #pragma omp parallel nowait : Pour n'attendre pas tous les autres threads de terminer leur tâches.
- #pragma omp parallel for schedule(static) : Diviser les itérations en paquets de taille donnée (nb itérations/nb threads) et l'attribuer cycliquement à chaque thread.
- #pragma omp parallel for schedule(dynamic,...) : Division des itérations selon nous.
- #pragma omp parallel schedule(runtime) : division à l'aide du variable d'environnement.
- #pragma omp parallel for ordered: pour une exécution ordonnée.
 - #pragma omp ordered : Où appliquer l'ordre.
- #pragma omp for reduction (+: s): Calcul partiel dans chaque thread puis synchronisation et calcul final du résultat.
- #pragma omp sections nowait : Partie exécutée par 1 seul thread.
 - #pragma omp section
- #pragma omp single : Exécuté par 1 seul thread (n'importe quel).
 - #pragma omp single copyprivate(x) : Enregistrer la valeur de x durant la partie parallèle.
- #pragma omp master: Exécuté par 1 seul thread (thread 0).
- #pragma omp barriere : Chaque thread attend tous les autres threads de terminer pour continuer ensemble.
- #pragma omp atomic: Protéger 1 variable, (accès protégé par 1 seul thread).
- #pragma omp critical : Protéger un ensemble des variables.

```
omp_get_thread_num();  
omp_get_num_threads();  
omp_get_wtime();
```

```
Pour random: #include<time.h>  
time_t t;  
srand((unsigned) time(&t));  
rand()....;
```

1 Environnement

1.1 Initialisation de l'environnement MPI

```
int MPI_Init(int * argc, char ***argv)
```

1.2 Rang du processus

```
int MPI_Comm_rank(MPI_Comm comm, int * rang)
```

1.3 Nombre de processus

```
int MPI_Comm_size(MPI_Comm comm, int * nb_procs)
```

1.4 Désactivation de l'environnement MPI

```
int MPI_Finalize(void)
```

1.5 Arrêt d'un programme MPI

```
int MPI_Abort(MPI_Comm comm, int error)
```

1.6 Prise de temps

```
double MPI_Wtime(void)
```

2 Communications point à point

2.1 Envoi de message

```
int MPI_Send(
    const void * message,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette,
    MPI_Comm comm)
```

2.2 Envoi non bloquant de message

```
int MPI_Isend(
    const void * message,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette,
    MPI_Comm comm,
    MPI_Request * requete)
```

2.3 Réception de message

```
int MPI_Recv(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    int etiquette,
    MPI_Comm comm,
    MPI_Status * statut)
```

2.4 Réception non bloquant de message

```
int MPI_Irecv(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    int etiquette,
    MPI_Comm comm,
    MPI_Request * requete)
```

2.5 Envoi et réception de message

```
int MPI_Sendrecv(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    int rang_dest,
    int etiquette_message_emis,
    void * message_recu,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_source,
    int etiquette_message_recu,
    MPI_Comm comm,
    MPI_Status * statut)
```

```
int MPI_Sendrecv_replace(
    void * message_emis_receu,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette_message_emis,
    int rang_source,
    int etiquette_message_receu,
    MPI_Comm comm,
    MPI_Status * statut)
```

2.6 Attente de la fin d'une communication non bloquante

```
int MPI_Wait(MPI_Request * requete, MPI_Status * statut)
```

2.7 Test de la fin d'une communication non bloquante

```
int MPI_Test(
    MPI_Request * requete,
    int * drapeau,
    MPI_Status * statut)
```

3 Communications collectives

3.1 Diffusion générale

```
int MPI_Bcast(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    MPI_Comm comm)
```

3.2 Diffusion sélective

```
int MPI_Scatter(
    const void * message_a_repartir,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_recu,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_source,
    MPI_Comm comm)
```

3.3 Collecte

```
int MPI_Gather(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_dest,
    MPI_Comm comm)
```

```
int MPI_Allgather(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    MPI_Comm comm)
```

3.4 Collecte et diffusion

```
int MPI_Alltoall(
    const void * message_a_repartir,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    MPI_Comm comm)
```



Cours : I3340

Année : 2017-2018

Durée : 2H30

Examen : P+F

Partie P (Partiel) : (sur 100 points pour ~ 45 minutes).

Exercice 1

Une manière de déterminer les nombres premiers dans un intervalle est de découper l'espace de recherche en plusieurs sous-intervalles et d'affecter la recherche dans chaque intervalle à un processus. Ainsi si on découpe l'intervalle $[I, N]$ en p sous-intervalles, le Thread k recherche dans l'intervalle $[kN/p + I, (k + 1)N/p]$, $k = 0, \dots, p - 1$. Cette technique permet, si on dispose d'une machine équipée de p processeurs, de paralléliser la recherche (et d'essayer de retourner le résultat p fois plus vite).

- On considère une machine avec p processeurs, écrire le programme c openmp qui permet de chercher les nombres premiers dans un intervalle $[I, N]$.

La décomposition statique possède l'inconvénient d'affecter des espaces de recherche dont les temps d'exploration sont inégaux. Ainsi le processus 0 terminera la recherche sur $[I, N/p]$ bien avant le processus $p - 1$. Il (et donc un des processeurs de la machine) sera donc inactif jusqu'à la fin du programme. La charge de travail entre les processus (et donc les processeurs) est inégalement répartie.

Une solution (celle que vous devez programmer) pour repartir la charge de travail, consiste à utiliser un « pool » de p processus et d'affecter successivement des travaux de recherche de nombres premiers dans de petits intervalles (taille $T << N/p$). Quand un processus a fini sa recherche il poursuit la recherche dans un intervalle encore inexploré.

- Expliquer comment cette nouvelle décomposition permet d'améliorer la performance de votre algorithme
- Modifier votre programme afin de faire la recherche d'une façon dynamique.

Partie F (Final) : (sur 100 points pour ~ 105 minutes)

Exercice 2

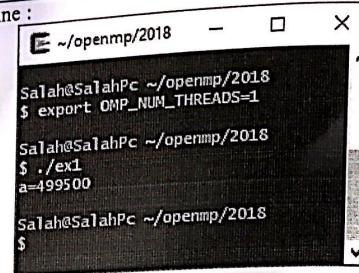
On considère le programme suivant :

```
include <stdio.h>
include <stdlib.h>
include <omp.h>
nt main()

int i, n=1000;
int a=0;
#pragma omp parallel for
for(i=0;i<n;i++)
{
    a+=i;
}
```

```
    printf("a=%d \n",a);
    return EXIT_SUCCESS;
}
```

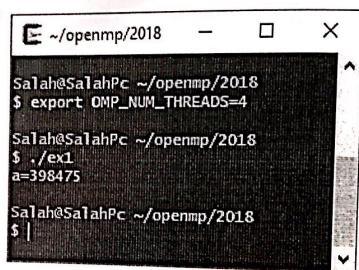
L'exécution avec un thread donne :



```
~$ ./ex1
a=499500
```

1. Expliquer le but du programme.

L'exécution avec 4 threads donne:



```
~$ ./ex1
a=398475
```

2. Expliquer la différence entre les deux exécutions et modifier le programme en utilisant la directive « atomic ».
3. L'utilisation de la directive « atomic » réduit l'efficacité du programme, expliquer pourquoi. Proposer une nouvelle solution qui permet d'accélérer l'exécution du programme.

Exercice 3 – Send-Receive circulaire

On souhaite écrire un programme qui effectue l'échange d'une valeur entre les différents processus de façon circulaire. La valeur échangée est initialisée par le processus zéro d'une façon aléatoire, et la valeur est transmise entre les différents processus. Le programme se termine lorsque le processus zéro n'a pas reçu la valeur du dernier processus. Le programme doit prendre un soin particulier pour s'assurer qu'il ne se bloque pas. En d'autres termes, le processus zéro s'assure qu'il a terminé son premier envoi avant d'essayer de recevoir la valeur du dernier processus. Tous les autres processus appellent MPI_Recv (pour la réception) et MPI_Send (pour l'envoi de la valeur à leur processus supérieur) pour passer la valeur le long de l'anneau. MPI_Send et MPI_Recv se bloqueront jusqu'à ce que le message ait été transmis. La fonction printf doit se produire dans l'ordre dans lequel la valeur est passée. En utilisant cinq processus, la sortie devrait ressembler à ceci.

```
Process 0 initialize token to 123
Process 1 received token 123 from process 0
Process 2 received token 123 from process 1
Process 3 received token 123 from process 2
Process 4 received token 123 from process 3
Process 0 received token 123 from process 4
```

le programme c (mpi) qui permet d'implémenter l'échange circulaire.

te faire un programme qui permet de chercher l'existence d'un nombre entier v dans un tableau de

cher le nombre entier v en utilisant le standard mpi, on divise le tableau en plusieurs sous tableaux
rche le nombre v, lorsqu'un processus trouve v il arrête son exécution, on pourra ainsi récupérer
u nombre recherché dans le tableau principal. La recherche au niveau de chaque sous tableau se
rallèle.

quer ce qu'il faut faire lorsqu'un processus trouve le nombre recherché.

é le programme c (mpi) qui permet de paralléliser la recherche de v.

wuhaite compter le nombre d'occurrence du nombre recherché. Ecrire le programme c qui permet de
nter le nombre d'occurrence d'une façon parallèle en utilisant les fonctions MPI_Scatter et
_Gather.

ifier le programme qui compte le nombre d'occurrence en utilisant les fonctions MPI_Scatter et
_Reduce.

Bon travail

3 Réduction

```
MPI_Reduce(  
    const void * message_emis,  
    void * message_recu,  
    int longueur,  
    MPI_Datatype type,  
    MPI_Op operation,  
    int rang_dest,  
    MPI_Comm comm)  
operation = [MPI_MAX | MPI_MIN | MPI_SUM | MPI_PROD |  
            MPI_BAND | MPI_BOR | MPI_BXOR | MPI_BAND |  
            MPI_LOR | MPI_LBXOR]
```

Annexe Parallélisme Open MP

- #pragma omp parallel : Construction de la région parallèle.
- #pragma omp parallel default(private) private(x) : Changer le statut par défaut de toutes les variables et de déterminer ce statut obligatoirement, private(x) : x sera trouver dans la pile de chaque thread.
- #pragma omp parallel firstprivate(x) : Initialiser x par la dernière valeur de x avant le parallèle.
- #pragma omp parallel for lastprivate(x) : Enregistrer la valeur de x de la dernière itération de la boucle for à la sortie du parallèle.
- #pragma omp threadprivate(x) : Pour privatiser la variable statique x dans la région parallèle.
 - #pragma omp parallel copyin(x) : Pour prendre la valeur de x (x : statique)
- #pragma omp parallel num_threads(...) : Pour déterminer le nombre des threads dans cette région.
- #pragma omp parallel nowait : Pour n'attendre pas tous les autres threads de terminer leurs tâches.
- #pragma omp parallel for schedule(static) : Diviser les itérations en paquets de taille donnée (nb itérations/nb threads) et l'attribuer cyclement à chaque thread.
- #pragma omp parallel for schedule(dynamic,...) : Division des itérations selon nous.
- #pragma omp parallel schedule(runtime) : division à l'aide du variable d'environnement
- #pragma omp parallel for ordered: pour une exécution ordonnée.
 - #pragma omp ordered : Où appliquer l'ordre.
- #pragma omp for reduction (+: s): Calcul partiel dans chaque thread puis synchronisation et calcul final du résultat.
- #pragma omp sections nowait : Partie exécutée par 1 seul thread.
 - #pragma omp section
- #pragma omp single : Exécuté par 1 seul thread (n'importe quel).
 - #pragma omp single copyprivate(x) : Enregistrer la valeur de x durant la partie parallèle.
- #pragma omp master: Exécuté par 1 seul thread (thread 0).
- #pragma omp barriere : Chaque thread attend tous les autres threads de terminer pour continuer ensemble.
- #pragma omp atomic: Protéger 1 variable, (accès protégé par 1 seul thread).
- #pragma omp critical : Protéger un ensemble des variables.

omp_get_thread_num();
omp_get_num_threads();
omp_get_wtime();

Pour random: #include<time.h>

```
time_t t;
srand((unsigned) time(&t));
rand()...;
```

1 Environnement

1.1 Initialisation de l'environnement MPI

```
int MPI_Init(int * argc, char ***argv)
```

1.2 Rang du processus

```
int MPI_Comm_rank(MPI_Comm comm, int * rang)
```

1.3 Nombre de processus

```
int MPI_Comm_size(MPI_Comm comm, int * nb_procs)
```

1.4 Désactivation de l'environnement MPI

```
int MPI_Finalize(void)
```

1.5 Arrêt d'un programme MPI

```
int MPI_Abort(MPI_Comm comm, int error)
```

1.6 Prise de temps

```
double MPI_Wtime(void)
```

2 Communications point à point

2.1 Envoi de message

```
int MPI_Send(
    const void * message,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette,
    MPI_Comm comm)
```

2.2 Envoi non bloquant de message

```
int MPI_Isend(
    const void * message,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette,
    MPI_Comm comm,
    MPI_Request * requete)
```

2.3 Réception de message

```
int MPI_Recv(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    int etiquette,
    MPI_Comm comm,
    MPI_Status * statut)
```

2.4 Réception non bloquant de message

```
int MPI_Irecv(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    int etiquette,
    MPI_Comm comm,
    MPI_Request * requete)
```

2.5 Envoi et réception de message

```
int MPI_Sendrecv(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    int rang_dest,
    int etiquette_message_emis,
    void * message_recu,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_source,
    int etiquette_message_recu,
    MPI_Comm comm,
    MPI_Status * statut)
```

```
int MPI_Sendrecv_replace(
    void * message_emis_recu,
    int longueur,
    MPI_Datatype type,
    int rang_dest,
    int etiquette_message_emis,
    int rang_source,
    int etiquette_message_recu,
    MPI_Comm comm,
    MPI_Status * statut)
```

2.6 Atteinte de la fin d'une communication non bloquante

```
int MPI_Wait(MPI_Request * requete, MPI_Status * statut)
```

2.7 Test de la fin d'une communication non bloquante

```
int MPI_Test(
    MPI_Request * requete,
    int * drapeau,
    MPI_Status * statut)
```

3 Communications collectives

3.1 Diffusion générale

```
int MPI_Bcast(
    void * message,
    int longueur,
    MPI_Datatype type,
    int rang_source,
    MPI_Comm comm)
```

3.2 Diffusion de message

```
int MPI_Scatter(
    const void * message_a_repartir,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_recu,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_source,
    MPI_Comm comm)
```

3.3 Collecte

```
int MPI_Gather(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    int rang_dest,
    MPI_Comm comm)
```

```
int MPI_Allgather(
    const void * message_emis,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    MPI_Comm comm)
```

3.4 Collecte et diffusion

```
int MPI_Gatherv(
    const void * message_a_repartir,
    int longueur_message_emis,
    MPI_Datatype type_message_emis,
    void * message_collecte,
    int longueur_message_recu,
    MPI_Datatype type_message_recu,
    MPI_Comm comm)
```