Ecole nationale Supérieure d'Informatique **Mars 2021**

Examen final HPC
2CS-SIQ

Durée : 2h **Documents interdits**

Exercice 1 [4 points]

Soit le code séquentiel suivant qui réalise le produit scalaire de deux vecteurs a et b. Donner la version parallèle de ce code en utilisant la bibliothèque Pthreads pour un nombre de threads égal à 4.

```
#include
#define SIZE 256

int main ()
{
   double sum , a[SIZE], b[SIZE];
   sum = 0.;

   for (size_t i = 0; i < SIZE; i++)
   {
      a[i] = i * 0.5;
      b[i] = i * 2.0;
   }

   for (size_t i = 0; i < SIZE; i++)
      sum = sum + a[i]*b[i];

   printf("sum = %g\n", sum);
   return 0;
}
```

Exercice 2 [5 points]

Proposer un algorithme qui calcule le nombre de premiers entre 2 et N, puis donner son implémentation parallèle en utilisant les fonctions et directives OpenMP.

Exercice 3 [5 points]

On se propose de calculer un produit matrice-vecteur $A \times X = Y$, où A est une matrice de taille 4x4, X un vecteur de taille 4 et Y le vecteur résultat. Donner le code MPI implémentant le produit matrice-vecteur selon les contraintes suivantes :

1- Allouer et initialiser la matrice A et le vecteur Y par le processus 0.
2- Chaque processus doit calculer un élément du vecteur Y.
3- Regrouper par le processus 0 tous les éléments calculés dans le vecteur Y.

Exercice 4 [2 points]

Analyser le code ci-dessous (OpenMP/MPI) et répondre aux questions suivantes :

1- Quelle est la tâche réalisée par le programme ?
2- Déterminer le parallélisme réalisé sur la tâche identifiée à la question 1 en expliquant la tâche réalisée par chaque processus ainsi que par chaque thread.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h>
#include "mpi.h"
#define CHUNKSIZE 10
#define N 100

void openmp_code(){
   int nthreads, tid, i, chunk;
   float a[N], b[N], c[N];
   for (i=0; i < N; i++)
      a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel private(i,tid)
   {
      tid = omp_get_thread_num();
      if (tid == 0){
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
      printf("Thread %d starting...\n",tid);
      #pragma omp for schedule(static,chunk)
      for (i=0; i<N; i++){
         c[i] = a[i] + b[i];
         printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
      }
   }
}
main(int argc, char **argv ) {
   char message[20];
   int i, rank, size, tag=99;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if(rank == 0) {
     strcpy(message, "Hello, world");
     for (i=1; i<size; i++)
        MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
   }
   else
     MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
     openmp_code();
     printf("Message from process =%d : %.13s\n", rank,message);
     MPI_Finalize();
}
```

Exercice 5 [4 points]

Ecrire un programme CUDA C qui permet à un GPU d'inverser un vecteur de taille 2048x2048. Le but est donc d'obtenir en sortie du GPU un vecteur symétrique du vecteur d'entrée.

Ecole nationale Supérieure d'Informatique **Mars 2021**

Examen final HPC
2CS-SIQ

Aide-mémoire sur quelques fonctions Pthreads et MPI

```
/* Obtenir l'identifiant unique du thread courant */
pthread_t pthread_self(void);

/* Créer un thread */
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void*(*start_routine)(void*), void *arg);

/* Terminer l'exécution du thread */
void pthread_exit(void *value_ptr);

/* Attendre la fin de l'exécution du thread thread */
int pthread_join(pthread_t thread,void **value_ptr);

/* Détacher le thread thread du thread courant */
int pthread_detach(pthread_t thread);

/* Créer une structure d'attributs de thread */
int pthread_attr_init(pthread_attr_t *attr);

/* Détruire une structure d'attributs de thread */
int pthread_attr_destroy(pthread_attr_t *attr);

/* Définir le statut détaché de l'attribut */
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

/* Créer un mutex */
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);

/* Détruire un mutex */
int pthread_mutex_destroy(pthread_mutex_t *mutex);

/* Acquérir un mutex (bloquant) */
int pthread_mutex_lock(pthread_mutex_t *mutex);

/* Libérer un mutex */
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* Créer une variable de condition */
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

/* Détruire une variable de condition */
int pthread_cond_destroy(pthread_cond_t *cond);

/* Attendre sur une condition */
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

/* Signaler un thread en attente sur une condition */
int pthread_cond_signal(pthread_cond_t *cond);

/* Initialiser l'environnement d'exécution MPI */
int MPI_Init(int *argc, char ***argv);

/* Terminer l'environnement d'exécution MPI */
int MPI_Finalize();
```

```
/* Retourner la taille du communicateur */
int MPI_Comm_size(MPI_Comm comm, int *size);

/* Renvoyer le rang du processus appelant */
int MPI_Comm_rank(MPI_Comm comm, int *rank);


/* Envoi bloquant */
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm);

/* Réception bloquante */
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Status *status);

/* Renvoyer le nombre de valeurs reçues */
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count);

/* Envoi non-bloquant */
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request);
```

```
/* Réception non-bloquante */
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Request *request);

/ Attendre la réalisation d'une communication non-bloquante */
int MPI_Wait(MPI_Request *request, MPI_Status *status);

/* Synchronisation globale */
int MPI_Barrier(MPI_Comm comm);

/* Diffusion globale */
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,int root, MPI_Comm comm);

/* Diffusion sélective */
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

/* Collecte de données réparties */
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

/* Réduction */
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm);

/* Partitionnement d'un communicateur */
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```