Programmation Parallèle : Examen Ecrit (1h00)

Numéro de copie:

Affichage

Lister toutes les possibilités d'affichage du program suivant:

```
1  #pragma omp parallel num_threads(2)
2    {
3  #pragma omp critical
4      {
5  #pragma omp task
6        printf("A");
7  #pragma omp task
8        printf("B");
9      }
10   }
```

Réponse:

Debogage MPI

Dans cet exercice, chaque rang MPI fournit une valeur locale dans la fonction `findMaximum` et cette fonction est appelée depuis tous les rangs.

Le but de cette fonction est de trouver le minimum des `valLoc` s fournis par les processus, puis surécrire `valLoc` de chaque rang avec cette valeur minimum. Lister tous les bogues/problèmes dans ce morceau de code, eventuellement en faisant référence au numéro de ligne correspondant à chaque bogue.

```
1  void findMaximum(int valLoc)
2  {
3    int rank, size;
4    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5    MPI_Comm_size(MPI_COMM_WORLD, &size);
6    int valRecv;
7
8    if (rank == 0) {
9      MPI_Send(valLoc, N, MPI_INT 1, 0, MPI_COMM_WORLD);
10     MPI_Recv(valRecv, N, MPI_INT, size - 1, 0, MPI_COMM_WORLD);
11     valLoc = std::min(valLoc, valRecv);
12   } else {
13     MPI_Recv(valRecv, N, rank - 1, MPI_COMM_WORLD);
14     valLoc = std::min(valLoc, valRecv);
15     MPI_Send(valLoc, N, rank + 1, MPI_COMM_WORLD);
16   }
17 }
```

Réponse:

Appel d'un kernel GPU

Voici trois kernels de GPU qui multiplient tous les éléments d'un tableau donné par une constante et une fonction main qui multiplie un tableau par 2.0 sur le GPU.

```
1  #define K 8
2
3  __global__ mulTableau(float *x, float a, int N)
4  {
5    int idx = blockIdx.x;
6    if (idx < N)
7      x[i] *= a;
8  }
9
10 __global__ mulTableau2(float *x, float a, int N)
11 {
12   int idx = blockIdx.x * blockDim.x + threadIdx.x;
13   if (idx < N)
14     x[i] *= a;
15 }
16
17 __global__ mulTableau3(float *x, float a, int N)
18 {
19   int idxBeg = blockIdx.x * blockDim.x * K + threadIdx.x;
20   int idxEnd = idxBeg + blockDim.x * K;
21   for (int idx = idxBeg; idx < std::min(idxEnd, N); idx += K)
22     x[idxBeg] *= a;
23     idxBeg += K;
24   }
25 }
26
27 int main()
28 {
29   int N = 1000;
30   float *x = (float *) malloc(sizeof(float) * N);
31   float *dx; cudaMalloc(&dx, sizeof(float) * N);
32   cudaMemcpy(dx, x, sizeof(float) * N, CudaMemcpyHostToDevice);
33   int numBlocks = ?;
34   int blockSize = ?;
35   mulTableau?<<<numBlocks,blockSize>>>(x, 2.0, N);
36   cudaMemcpy(x, dx, sizeof(float) * N, CudaMemcpyDeviceToHost);
37   free(x); cudaFree(dx);
38   return 0;
39 }
```

Préciser les valeurs de `numBlocks` et `blockSize` qu'il faut utiliser afin d'effectuer le calcul correctement en générant le minimum nombre de threads possible. Ceci est à faire dans les trois scénarios suivants pour les lignes 34,35,36:

FIN D'EXAMEN