Final Exam Module II: Parallelism **CPDS Q2-2019-2020** **5th June 2020**

SURNAME, NAME: .... DNI/NIF:..............................

The duration of the exam is **75 minutes**.
The **grades of the exam** will be published on the **12th June 2020**

Exercise 1 (0,5 point)

a) Many parallel languages have a barrier construct, e.g. MPI_Barrier in MPI or #pragma omp barrier in OpenMP. Define barrier.

b) How does one implement a global barrier in CUDA? Explain your answers. **Note**: This is a trick question. There are no barrier constructs in the programming language. The question asks, when do all parallel execution threads synchronize in CUDA ?

Exercise 2 (1 point)

Given the following C code with OpenMP pragmas:

```
#include #include
#define NUM_OF_COLUMNS 6
#define NUM_OF_ROWS (3*(NUM_OF_COLUMNS - 1))

int whichThread[NUM_OF_ROWS][NUM_OF_COLUMNS];

void fillColumn(int j) {
  int i;
#pragma omp for
  for (i = 0; i < NUM_OF_ROWS; i++)
    whichThread[i][j] = omp_get_thread_num(); }

int main() {
  int i, j;
  for (i = 0; i < NUM_OF_ROWS; i++) // initialize the array
    for (j= 0; j < NUM_OF_COLUMNS; j++) whichThread[i][j] = -1;

#pragma omp parallel num_threads(NUM_OF_COLUMNS - 1)
  fillColumn(0);

#pragma omp parallel for num_threads(NUM_OF_COLUMNS - 1)
  for (j = 1; j < NUM_OF_COLUMNS; j++) fillColumn(j);

  for (i = 0; i < NUM_OF_ROWS; i++) // print out the results
    for (j= 0; j < NUM_OF_COLUMNS; j++) printf(" %2d ", whichThread[i][j]);

  printf("\n"); } return 0;
}
```

SURNAME, NAME: .... DNI/NIF:.................................

a) What is the output of this program if it is compiled **without** the -fopenmp compiler flag? Briefly explain why.

b) What is the output of this program if it is compiled **with** the -fopenmp compiler flag? Explain, very carefully, why the output differs from the output of the serial version of the program.

Exercise 3 (1,5 points)

a) Parallelize the following code using OpenMP pragmas. Be sure to explicitly specify the "schedule" options that should be used for better performance.

```
for (i=1; i<N; i++) {
  for (j=1; j<i; j++) {

    C[i] *= A[i][j] + B[i][j];

  }
}
```

b) Let be Tc the cost of executing one iteration of the j-loop, Let be T the number of threads to be used. Let be N the number of iterations in the i-loop. Let be *Fi(th)* and *Li(th)* the first and last iteration assigned to thread *th*. Let be *Ni(th)* the total number of iterations assigned to thread *th*. Let be *W(th)* the total amount of work performed by thread *th* if the i-loop were executed in parallel under a **STATIC** scheduling. Assuming T divides N, give an expression to model all the previous defined variables:

```
Ni(th) =

Fi(th) =

Li(th) =

W(th) =
```

SURNAME, NAME: .... DNI/NIF:..............................

Exercise 4 (0,5 point)

Fill the table with any of the two options (device or host) in each case for the function prototype:

Keyword	Executed on the:	Only callable from the:
`__device__ void Function()`		
`__global__ void Function()`		
`__host__ void Function()`		

Exercise 5 (0,5 points)

Given a thread organization in the form of a ·**2D grid of 1D blocks** of threads, use the CUDA built-in variables (gridDim, BlockDim, BlockIdx, ThreadIdx) to compute the global thread ID:

Exercise 6 (1,5 points)

The following code snippet corresponds to a code skeleton for reduction operations in CUDA. Assume *SharedData* is a vector allocated in shared memory and stores the data for the reduction.

```
for (unsigned int j=blockDim.x >> 1; j>0; j>>=1)
{
  if (tid < j)
    SharedData[tid] += SharedData[tid+j];
  __syncthreads();
}
```

One possible optimization is making a specialized code version for an specific value of blockDim.x =256:

```
if (tid < 128) SharedData[tid] += SharedData[tid+128]; __syncthreads();
if (tid < 64) SharedData[tid] += SharedData[tid+64]; __syncthreads();
if (tid < 32) {
  SharedData[tid] += SharedData[tid+32];
  SharedData[tid] += SharedData[tid+16];
  SharedData[tid] += SharedData[tid+8];
  SharedData[tid] += SharedData[tid+4];
  SharedData[tid] += SharedData[tid+2];
  SharedData[tid] += SharedData[tid+1];
}
```

This loop unrolling eliminates the call to __syncthreads() when there are fewer than 32 active threads in a thread block.

a) Why is __syncthreads() necessary in general, e.g. in the loop prior to unrolling? What purpose does it serve?

SURNAME, NAME: .... DNI/NIF:..............................

b) Why is it safe to eliminate __syncthreads() in the unrolled loop for fewer than 32 threads? What CUDA principle does this demonstrate?

Exercise 7 (2,5 point)

Sketch a CUDA program for adding the elements of a two vectors, A and B and store the result in a vector C, of size N. Your code should include all key CUDA API calls and the usage of shared memory, if necessary.

```
__global__ void Addition(int *a_d, int *b_d, int *c_d, int N)
{
/* GPU code */




}

int main(void)
{
  const int n = "very large number";
  int a[n], b[n], c[n];


  int *a_dev, *b_dev, *c_dev;


/* CPU code */

}
```

SURNAME, NAME: .... DNI/NIF:.............................

Exercise 8 (1,5 points)

Sketch an MPI program for adding the elements of a vector, V, of size N. Your code should include all key MPI API calls. Assume P processors. Assume the content of the V vector initially is stored in MPI process 0.

```
int main (int argc, char *argv[])
{
  int *a, *b, *c;
  int total_proc; // total nuber of processes
  int rank; // rank of each process
  long long int n_per_proc;   // elements per process
  long long int i, n;
  unsigned int MASTER=0;

  MPI_Status status;

  // Initialization of MPI environment
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &total_proc);
  MPI_Comm_rank (MPI_COMM_WORLD,&rank);

  int *ap, *bp, int *cp;

  if (rank == MASTER) {
    a = (int *) malloc(sizeof(int)*n);
    b = (int *) malloc(sizeof(int)*n);
    c = (int *) malloc(sizeof(int)*n);

    MPI_Bcast (&n, ______________________________________________________);
    n_per_proc = n/total_proc;
    MPI_Bcast (__________________________________________________________);

    MPI_Scatter(a, ______________________________________________________);

    MPI_Scatter(_________________________________________________________);


    for(i=0;i<n_per_proc;i++) cp[i] = ap[i]+bp[i];

    MPI_Gather(__________________________________________________________);
  } else { // Non-master tasks
    MPI_Bcast(___________________________________________________________);

    MPI_Bcast(___________________________________________________________);

    ap = ______________________________________;

    bp = ______________________________________;

    cp = ______________________________________;

    MPI_Scatter(_________________________________________________________);

    MPI_Scatter(_________________________________________________________);

    for(i=0;i<n_per_proc;i++) cp[i] = ap[i]+bp[i];

    MPI_Gather(__________________________________________________________);

  }
  MPI_Finalize();
  return 0;
}
```

SURNAME, NAME: .... DNI/NIF:..............................

Exercise 9 (0,5 point)

a) In MPI, what is a process rank?

b) In MPI you set the number of processes when you write the source code. **True** or **False**:

c) Explain if the following MPI code segment is correct or not, and why:

Process 0 executes:

```
MPI_Recv( &yourdata, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD, &status);
MPI_Send( &mydata, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
```

Process 1 executes:

```
MPI_Recv( &yourdata, 1, MPI_FLOAT, 0, tag,MPI_COMM_WORLD, &status);
MPI_Send( &mydata, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
```