Computação de Alto Desempenho 2013/14
2nd Test – 3/6/2014
Duration: 2h15

Closed book test; possible doubts about the contents should be solved by the student; please include the assumptions you made in your answers.

1. Nowadays it is very common to resort to GPUs to solve problems requiring intensive computing.

a) Identify in which classes of the above problems do the GPUs support significantly higher performance gains than conventional CPUs. Justify your answer.

b) Identify which characteristics of current GPU-based architectures prevent the applications from achieving the highest levels of performance potentially provided by the GPU hardware.

2. The goal of a program is to simulate the evolution along the time of a metal plate with NxN points (e.g. assume N=5000). Initially, the plate is at 20 Celsius degrees, and at the simulation's time *T=0* the plate's border is heated to 200ºC *in a constant way*. At time *T* of the simulation, the calculation of the new temperature of a plate's point *C[i,j]* is represented by the following C pseudo-code:

```
if ( i== 0) || (j== 0) || ( i==N-1) || (j==N-1) ) {
   a point's temperature remains the same (200°C)
}
else {
   a point's temperature is equal to the average temperature of its
   four neighbours [i-1,j], [i+1,i], [i,j+1], and [i,j-1], which is
   in turn multiplied by the constant CT=0.335
}
```

Implement the above algorithm using ***Cuda C***. Assume that at each simulation's step two matrices of *NxN* real numbers are used, and one contains the points' temperatures at time *t-1* and the other is updated with the new temperature values at time *t*. Fill in the following C code with your code in *Cuda C* that is necessary to implement the algorithm.

```
#include <stdio.h>
#include <stdlib.h>

#define END 20
#define N  5000
float t_1[N][N]; // matrix with the state of the plate at time t-1
float t[N][N];   // matrix with the state of the plate at time t

// define the kernel function

int main (int argc, char *argv[])
```

```
{
   FILE *f;

   // Implement the simulation code for 20 time units

    // Results' output
   f= fopen("simulacao.bin", "w");
   fwrite(t, sizeof(float), N*N, f);

   return 0;
 }
```

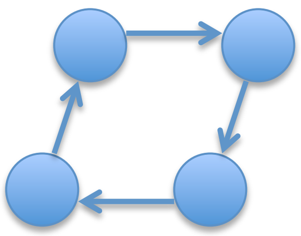
3. Consider a massively parallel program written in MPI to process a vector *V*, where only the node with *rank 0* has access to that vector. Explain how to use the collective communication primitives from MPI to distribute the data to be processed among all MPI nodes/processes and, subsequently, to collect the final result, for the following three problems:

a) the values in vector *V* are to be independently processed at each node (whatever that processing may be); in the end, only the node with *rank 0* prints the vector *R* with final result (this result vector *R* has the same dimension as vector *V*);

b) initially, the vector *V* contains a set of real numbers representing the salaries of some population, and the goal is to calculate how many individuals earn a salary above the average; the calculated number must be available at every node (and not only at the node with *rank 0*);

c) in this case, the initial values at vector V are ignored, but after the processing at all nodes, the vector V must contain, at each position, the factorial considering that position as argument; namely, in the end the contents of the vector V should be V[0] == 1, V[1] == 1, V[2] == 2, V[3] == 6, ... V[n] == n!. For this problem you just have to identify which is the name of the pattern that represents similar calculations and which is name of the collective communication primitive in MPI that allows this calculation.

4. Assume a *communicator* in MPI that contains a group of processes exchanging messages using *MPI_Send* and *MPI_Recv*, according to what is shown in the following figure:



a) Explain in which case a particular implementation of the *MPI_Send* and *MPI_Recv* functions may disrupt the communication of that group of processes.

b) Explain which MPI functions may be used alternatively so that the result is efficient, independently of which MPI's implementation is chosen.

5. Consider the following approximate method to calculate π:

```
#define TIMES 10000000
t = 0;
for( i = 0; i < TIMES; i++ ){
   x = randx( );  /* randx returns a random real number
                  between 0 e 1*/
   y = randx( );
   if( x*x+y*y<=1.0) t=t+1;
}
pi = 4.0*((double)t/(double)TIMES);
```

Using the MPI primitives, program a parallel version of this algorithm for a cluster of 16 personal computers connected by a local network working at 1 Gbps, where the MPI was installed.

6. Recall the *N-body* problem to simulate the positions and velocities of a set of planets described during the classes[1]. The pseudo-code of a sequential program to simulate the set's behaviour can be:

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

The *Annex A* presents the pseudo-code of the so called *basic algorithm*, to compute the total forces, and the positions and velocities in the code above. The *Annex A.a)* presents the pseudo-code to calculate the total forces over each planet (particle) *q*, and the *Annex A.b)* presents the pseudo-code to calculate the new positions and velocities of each planet (particle) *q* as a result of the calculated forces.

Use the Foster methodology to specify how to parallelise this *N-body* problem using a *hybrid programming strategy* with *MPI* and *OpenMP*.

[1] Description in "An Introduction to Parallel Programming", chapter 6, Peter Pacheco.

Annex

N-body solver to simulate the behaviour of a set of planets[1]

A. *Basic algorithm*

a) Calculation of the total of forces over the planets (particles) *q*:

```
for each particle q {
    for each particle k != q {
        x_diff = pos[q][X] − pos[k][X];
        y_diff = pos[q][Y] − pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] −= G*masses[q]*masses[k]/dist_cubed * x_diff;
        forces[q][Y] −= G*masses[q]*masses[k]/dist_cubed * y_diff;
    }
}
```

b) Calculation of the new positions and velocities of all planets (particles) *q*, as a result of the calculated forces:

```
for each particle q {
    pos[q][X] += delta_t*vel[q][X];
    pos[q][Y] += delta_t*vel[q][Y];
    vel[q][X] += delta_t/masses[q]*forces[q][X];
    vel[q][Y] += delta_t/masses[q]*forces[q][Y];
}
```