Computação de Alto Desempenho 2015/16
2nd Test – 9/6/2016
Duration: 2h00

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

1. Consider the following parallel computing standards/libraries and their programming models:
   i) OpenMP;
   ii) CUDA C;
   iii) MPI;

   a) Describe the difficulties a programmer may have on developing programs in CUDA C comparatively to OpenMP.
   b) Assume a parallel programming *hybrid approach* consisting of *OpenMP and MPI*. For which kind of applications would such an hybrid approach be better than using OpenMP or MPI separately? Justify your answer.
   And for which kind of applications would an *hybrid approach* consisting of *CUDA and MPI* be better than using *OpenMP and MPI*? Justify your anwer.

2. Describe the major characteristics of the *tiling technique* in CUDA and justify why it is useful for Matrix multiplication.

3. An application to *simulate a plague propagation in a pine tree forest* needs to be parallelised using *CUDA C* to improve its performance. The application contains *a function to simulate the plague propagation* in a square area where the pine trees are uniformly distributed. The inputs to this function are:
   - a matrix representing the pine trees in a square area of the forest;
   - the dimensions of the matrix, i.e. the number of pine trees per line (i.e. the width), and the number of pine trees per column (i.e. the height);
   - the probability of the plague spreading between all neighbouring trees;
   - the position of the pine tree where the plague begins, identified by the number of the line, *L*, and the number of the column, *C*, in the matrix.

   Initially, all pine trees in the forest are tagged with the value *zero*, i.e. the trees are *healthy*. The function simulates the plague propagation in a sequence of steps/iterations, and at each of these iterations, the function calculates the new state of all pine trees in the following way:
   - if a pine tree is already *dead* (its *value is 2*), the tree remains dead (its *value remains 2*);
   - if a tree is *ill* (its *value is 1*), the tree becomes dead (its *value becomes 2*);
   - if a tree is still *healthy* (its *value is 0*), then

- if *any of its neighbours* is already ill and therefore contagious (i.e. it has value 1), the tree becomes ill as well (i.e. its *value becomes 1*);
- else, a random value is generated and compared with the plague propagation probability received as input, and
  - if the generated value is bigger than this probability, the tree becomes ill (i.e. its *value becomes 1*);
  - else, the tree's remains healthy (i.e. its *value remains 0*).

The plague propagation function *terminates* the iterations when there is a *majority of dead trees* or the illness propagation ends by itself (e.g. the trees are resilient enough). The output of this function is one of two values:

- *zero*, if the plague propagation simulation has ended without a majority of dead trees (i.e. the plague propagation disappears and the number of healthy trees is equal or bigger than the number of dead trees);
- *an integer* identifying the *step/iteration number* of the simulation when the majority of trees became dead.

The prototype of the plague propagation function is:

```
int plague_propagation(char *pt_forest, int width, int height,
                float plague_prob, int line, int column);
```

For instance, this function may be invoked in the following way:

```
#define N  1024
unsigned char *forest = malloc(N*N);
niteration = plague_propagation(floresta, N, N, 0.55, 512, 512);
```

The goal of this question is to implement a parallel version of this application using ***Cuda C***, by filling in the following code skeleton in C with your *Cuda C* code.

```
#include <stdio.h>
#include <stdlib.h>
#define N  2048   // size of the side of the square matrix

/* Each forest's pine tree occupies a position in the matrix named
forest; if its value is 0, the tree is healthy; if it is 1, the tree is
ill; if it is 2, the tree is dead. Initially, all trees have to be
healthy.*/

unsigned char *forest = malloc(N*N);

// your kernel function

int main (int argc, char *argv[])
{
   FILE *f; int num_iteration;
   float plague_propagation = atof(argv[1]); // propagation probability
   int line = atoi(argv[2]); // line position of the first ill tree
   int column = atoi(argv[3]); // column position of the first ill tree

   // your CUDA C code
```

```
    // Output of results
    f= fopen("result.bin", "w");
    fwrite(forest, sizeof(unsigned char), N*N, f);
    fclose(f);
    printf("Number of the simulation's iteration: %d\n", num_iteration);
    return 0;
}
```

4. Consider a computational platform composed of ***N*** personal computers connected by an 1 Gbps local network where the *MPI* is installed. A program following the SPMD (*Single Program Multiple Data*) model is to be executed in this architecture by *N processes*. These processes use an *election model* to decide which one performs a particular function *f()*, at some point in time. Hence the processes have to periodically coordinate among themselves to select the process executing *f()*. This has to be preceded by the initialisation of the necessary data and the execution should end after the function has been invoked *MAX_CALL* times. Use the suitable *collective communication functions* to implement such an election model in order to satisfy the following conditions:

- **a)** Initially the ***rank 0 process*** generates *N random numbers* (e.g. between ***0*** and ***1***) filling a ***vector V*** with dimension N. This vector contains the seed values to be distributed to the N processes, one value per each process, so that each one may generate a sequence of pseudo-random numbers (i.e. the generated number sequence at each process can be reproduced e.g. for debug or validation purposes).
- **b)** All processes subsequently execute a cycle where, periodically, a (pseudo-)random number is generated, and this value is compared to all the (pseudo-)random numbers generated at the other processes. Namely, a process only executes the function *f()* if its local number is the ***bigger than the average*** of all the other generated numbers.
- **c)** In the end, all processes should have information about which processes invoked the function *f()* and how many times they did so. For this a ***vector K*** with dimension N will contain the number of times each processes invoked *f()* (e.g. for k[0]=3, k[1]=0, k[2]=1, etc, *rank 0* process invoked *f()* three times, *rank 2* process invoked *f()* one time, etc.).

5. Consider again the plague propagation problem in a pine tree forest that was described in question 3. Portugal has an extensive area of pine tree forest to be controlled against plagues and the simulation should use now a prediction function for the plague propagation to be applied to the whole area under observation. Namely, instead of using a fixed plague propagation probability among neighbouring trees, the *plague_propagation()* function uses a prediction function named *propagation_probability()*, which is available from a library.

The prototype of the *plague_propagation* function is now:

```
int plague_propagation(char *pt_forest, int width, int height,
                       int line, int column);
```

And the prototype of the *propagation_probability* function is:

```
float propagation_probability(int line, int column);
```

This function gets fresh values on the environment conditions (e.g. soil, humidity, etc) that have to be collected from sensors or weather services, and calculates and returns the plague propagation probability for the pine tree located at point (*line*, *column*). Depending on the area where a tree is located (e.g. its soil quality, local higher humidity near a river may facilitate propagation etc), the calculation time and returned values may vary from pine tree to pine tree.

Taking these conditions into account, the program should now be deployed in a cluster of 16 personal computers connected by a local network working at 1 Gbps, where the *OpenMP* and the *MPI* are installed.

a) Describe the application of the Foster methodology to the development of a parallel solution in this architecture justifying, for each phase, the options you make.

b) Write which MPI functions you would need to use to implement this code and how you would divide the workload considering this *hybrid OpenMP and MPI* architecture. Justify your answer.