Computação de Alto Desempenho 2015/16
Solution Draft of the 2nd Test 2013/14

1.

a) The GPUs show high performance improvements in massively parallel problems (i.e. without dependency restrictions among the tasks); the *data parallel problems*, e.g. following the *Single Program Multiple Data (SPMD)* model (i.e. based on a pattern of geometric data division where the same function is applied to all data elements like in the *map* pattern), also benefit from GPU execution.

b) Examples of the current GPU architectures' characteristics that restrict their performance potential *[1]* (3 characteristics would be enough):

-- The programmers need to know the details of a GPU's architecture in order to fully explore its capabilities. For instance, when moving a code to be executed on a GPU it is frequently needed to re-design its algorithm or its data organisation. Moreover, if more than one GPU is present, it is necessary to explicitly select a device where the code will be run.

-- There is separation between host memory and device memory, and the GPUs have limited memory. This forces an explicit management of the device's memory (explicit allocation and de-allocation of memory since GPUs do not have "*garbage collection"*), and explicit data transfer between host and device, and vice-versa. Therefore, a restrictive characteristic is a device's *compute intensity*, i.e. the ratio between the number of operations and the data amount that is necessary to move; in this case, the biggest limitation is the *bandwidth* between host and device and the data volume to be transferred between them. Moreover, the program's data needs to be analysed to identify which data needs to be placed on the device and to reserve this space, and also which data needs to be copied between *host* and *device* (and vice-versa) and when.

-- It is necessary to identify a code region's parallelism possibilities (e.g. identification of execution cycles) and to optimise the data access patterns, exploring the possible parallelism levels.

-- The code to be executed on a GPU has to be explicitly defined in a *kernel* that is initialised and transferred to the device, being necessary to subsequently wait for the end of its execution.

It is also necessary to prevent situations of "*thread divergence*" where only some threads execute an instruction; it may happen that one thread blocking (e.g. by invoking *__syncthreads()*) may lead to all the other threads in a block stalling as well.

2. See *q2.cu*.

3.
```
/* All processes invoke the following collective communication functions
in the same order. */
```
a)
```
MPI_Comm_size (comm,&comm_sz);
int local_sz = vsize/comm_sz;
MPI_Scatter (V, local_sz, vtype, recvbuf, local_sz, vtype, 0, comm )
```

```
// … the processing is performed at each node …

MPI_Gather ( sendbuf, local_sz, vtype, R, local_sz, vtype, 0, comm )
```

b)

```
MPI_Scatter (V, local_sz, vtype, recvbuf, local_sz, vtype, 0, comm )
float local_sum, global_sum;
// each node sums all its local elements in recvbuf into local_sum
MPI_Allreduce (&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, comm);
int global_count;
MPI_Allreduce (&local_sz, &global_count, 1, MPI_INT, MPI_SUM, comm);
float average = global_sum/global_size;
int local_above, global_above;
// local_above = calculation of how many individuals have a salary above average
MPI_Allreduce (&local_above, &global_above, 1, MPI_INT, MPI_SUM, comm);
```

c) *Scan* pattern and the *MPI_Scan* operation.

```
MPI_Scan ( V, V, vsize, MPI_INT, MPI_SUM, comm )
```

4. **a)** In case of a blocking *MPI_Send* (and *MPI_Recv*) implementation, the process communications in the figure lead to a *deadlock* situation. Even for a buffered *MPI_Send* implementation and in case of a high data volume to be sent, the *MPI_Send* will not return until the recipient starts receiving the data leading also to a deadlock.

b) The MPI_*Sendrecv* primitive can be used.
The functions *MPI_Isend* and *MPI_Irecv* can also be used but this requires additional code (e.g. using *MPI_Test()* and *MPI_Wait()*) in case it is necessary to guarantee that the data was in fact sent/received (before the execution may continue).

5. See code *q5.c*.

6. Please see section *6.1.9* of the book cited in [2] and the code which was made available online [3]. The following text is a *simplified description* of applying the 4 phases of the Foster's methodology:
- *Partitioning*: identification of all possible parallel tasks (for all the particles),
T1 – calculation of the forces over particle *q*;
T2 – calculation of the new position and velocity of *q*;
TI – *input task* – get the mass and initial position and velocity of all particles; scatter/distribute that information to all existing MPI processes;
TO – *output task* – in case it is necessary to produce output information in the current iteration (i.e. writing the current state of the simulation at the end of this iteration), the positions and velocities of all particles have to be gathered and printed.

- *Communication*: Tasks *T1i* (i:1..num_planets) depend on *T0*; each *T2i* depends on the corresponding *T1i* (same i); all *T1i* need to communicate among themselves (*allgather*), like all *T2i*; *TO* needs the results of all *T2i*.

- *Aggregation*: For the same *i*, *T1i* and *T2i* should be grouped in a task with bigger granularity; to reduce the communication, several of these bigger tasks should be processed in the same node.

- *Mapping*: division of the work among the nodes considering a static work distribution (since the workload is similar among the tasks in the *basic algorithm*); the total number of tasks is divided among the number of available MPI processes; at each node/process the tasks are distributed among the created threads (according to the architecture/number of cores in the nodes). The *rank 0* node executes the tasks *TI* and *TO* (it is assumed that the execution time of *TO* is small in comparison to the other tasks).

[1] More information available from, for instance,
NVIDIA, "CUDA C Best practices guide", March 2015;
PRACE, "Best Practice mini-guide accelerated clusters, using general purpose GPUs", May 2013.

[2] Peter Pacheco, "An Introduction to Parallel Programming", Elsevier, 2011.

[3] File *mpi_nbody_basic.c* in the archive named *exs-ch6-PeterPacheco.zip* in section "Textos de Apoio" no clip.