MPI Case Study

Fabio Affinito

April 24, 2012

In this case study you will (hopefully..) learn how to

- Use a master-slave model
- Perform a domain decomposition using ghost-zones
- Implementing a message passing form of the *Game of life*

Each of the above tasks relies on the previous one. Each one is self-contained and can be extended to reach the next objective. If you complete all the steps you should end up with a full working message passing implementation of the *Game of life*. If you don't manage to finish all the three steps, consider that the first part is the most difficult and important. It would be more important to succesfully finish the first part than the others. If you need some help, try to collaborate with your classmates. This is more useful than to watch the solutions. If you manage to finish all the case study, you will find at the end of this paper some suggestions of possible improvements.

1 Stage 1: the master-slave model

In this part you will:

- Create a master-slave model. The master will manage the output of data to a file
- Partition a 2-dimensional array between processors
- Generate a cartesian virtual topology
- Processor will assign a value to each array entry, depending on position
- Communicate back the data to the master processor which will write it on a file or on the standard output (screen)

We expect that the result of this stage will be a "chess-board" made of 1 or 0 values.

To start from scratch, you can take advantage from the following pseudo-code:

```
Initialize MPI

Find out how many processors there are
Check if the number of processors is appropriate for a grid
```

```
Create a two-dimensional periodic, cartesian grid

On processor 0:
Use a XSIZE by YSIZE integer array
Initialize elements to a 0 value (or, in generale to a MAXGREY value)
On other processors:
Use a XSIZE/nXprocs by YSIZE/nYprocs integer array
Initialize elements

Find the cartesian coordinates of the processor (x,y)
if(x+y+1)mod2 == 1 :
set elements in local array = 0
else
set elements in local array = MAXGREY = 1
Create derived datatype(s) to transfer local data back to processor 0

If processor 0:
do loop from 1 to number of processors-1
receive data using derived datatype into the appropriate part of
the array
else
processor sends data to processor 0

Finalize MPI
```

1.1 Step 1

Find the basic info: processor rank and number of processors used. This step also applies what you have learnt from virtual topologies. The virtual topology will allow you to map the data to the processors and ease the identification of the nearest neighbours.

Exploit an appropriate feature of the MPI standard to determine how many processors are for the X and Y dimension.

In this step you will make use of the following functions: MPI_COMM_SIZE, MPI_COMM_RANK, MPI_DIMS_CREATE, MPI_CART_CREATE.

1.2 Step 2

Initially set XSIZE=YSIZE=48 (don't exceed with this value if you want to print the whole matrix on the screen) and MAXGREY=1 (then you can change this value if you want more accurate visualization...). Note that for the above algorithm to work correctly the number of processors in each cartesian direction must divide exactly into XSIZE and YSIZE. We suggest to put a check on the code to enforce this condition.

Pay some attention on the way you use dynamic memory allocation.

1.3 Step 3

Once you have determined what portion of data space on a processor is responsible for, the data can be initialized. To do this, you must know where the processor lies in relation to the global data space. This can be derived from the cartesian coordinates (x,y) in the virtual topology in which the processor lies. If we assign 0 to the processors with (x+y+1) an even number, the desired chess board pattern will be obtained.

MPI routines required: MPI_CART_COORDS.

1.4 Step 4

The data from slave processors must be sent back to the master processor which will the reconfigure the global data space and output these numbers to a file (or to the screen). The way this process is done is dependent on the way the data is stored on the slave processors (and hence if you are using C or Fortran).

To transmit data from slaves to the master processor, you can use derived datatypes, i.e. a datatype with integer data alternate to blank spaces in a way that the receiving processor (the master) can accomodate the data in a suitable way.

To create derivate datatype for a data block, use the MPI_TYPE_VECTOR. Remember that in C rows are contigous in memory while in Fortran it is the columns that are contigous.

MPI Routines required: MPI_TYPE_VECTOR, MPI_TYPE_COMMIT, MPI_SEND, MPI_RECV.

If you are successful in this first part of the exercise, you should end up with a chess like board. Try first to use only 4 processors to debug your code. Then try to increase the number of processors putting care to avoid numbers that does not divide evenly the array dimensions. If you have completed this section you can proceed by moving to the next stage.

2 Stage 2: Boundary swaps

In this part of the exercise you will:

- Create a ghost region around each processor domain
- Perform ghost region swaps across processor domains.

In a lot of domain decomposition type problems it is often necessary to swap data at the boundaries between processor domains. This is done to minimize the subsequent communication between processors. The data imported from other processors is often referred as the ghost (or halo) region.

In this exercise we suggest you to allocate some vectors to contains the ghost regions. To copy regions of the local array onto these vectors, you can once again use derived datatypes.

To help you coding, you can follow this schema:

```
Create a row/column derived datatype
Find the nearest neighbours in x-direction
Find the nearest neighbours in y-direction
Swap boundaries with the processor above and below
Swap boundaries with the processor to the left and right
```

MPI routines required: MPI_TYPE_VECTOR, MPI_TYPE_COMMIT, MPI_CART_SHIFT, MPI_SENDRECV.

3 Stage 3: Building the Game of Life

You can now complete the case study by implementing the rules of the Game of Life using the static domain decomposition you have developed in the previous stages of this exercise.

If you get this far by the end of this exercise you will have implemented a complete application using MPI.

The next section will describe shortly the Game of Life if you are not familiar with it. If you already know this you may skip over it.

3.1 What is the Game of Life?

The Game of Life is a simple 2-dimensional cellular automata conceived by J.H.Conway in 1970. The model evolves a population of organisms in a 2-dimensional space and can exhibit very complex behavior from a very simple set of evolution rules.

The underlying evolution principle is very simple: cells can be alive or dead at anyone time step. The state of the system at the next time step is determined from the number of nearest neighbours each cell has at the present time.

The rules for evolving a system to the next time level are as follows:

- **dead** if the cell has less than two live neighbours
- **same state of previous time step** if the cell has exactly two live neighbours
- **newborn** if the cell has exactly three live neighbours
- **die** if the cell has more than three live neighbours (overcrowding)

3.2 Implementation

You can implement the rules of Game of Life using the domain decomposition that you have built in the previous stages. Each processor update the cells of his domain and, at each time step they swap the ghost regions. Remember that only the values inside the local array should be updated, while the ghost regions are necessary to calculate properly the number of nearest neighbours.

At each time step the global array is reconstructed on the master processor that is in charge of the output to a file. You can consider different possibilities for the visualization of the result. For example, you can write each time step to a different file and then reconstruct the dynamic behavior.

4 Supplementary exercises

If you reached the end of this exercise, you can consider to implement some modification on the code to make some improvements. The following are some suggestions:

- Try to never allocate completely the global array. In practise, the only need for a master processor is to perform input/output operation: there is no reason to allocate completely the global array on one processor. (note: very difficult without using MPI I/O functions).
- In the different stages we considered XSIZE=YSIZE. Is this necessary? (easy)
- If in the first stage you used send and receive, try to use collective functions. Or viceversa (medium)
- Release the constraint on the number of processors that divide exactly the global size of the array. (medium)

5 How to compile and run on CINECA PLX

For this case study, we will use the CINECA PLX cluster. So you will work remotely using a ssh connection.

If you want you can, alternatively, download an MPI implementation (for example *OpenMPI*, a compiler, and work on your own computer.

If you work on PLX, you can take advantage of the pre-installed modules. If you want to show all modules type:

```
module available
```

To compile your MPI application you need a compiler (C or Fortran) and an MPI implementation. For example, if you want to use GNU compilers and OpenMPI you can type:

```
module load gnu
module load openmpi/1.3.3--gnu--4.1.2
```

Alternatively, if you want to use the Intel compiler (strongly suggested for Fortran):

```
module load intel/co-2011.6.233--binary
module load openmpi/1.4.4--intel--co-2011.6.233--binary
```

Once you loaded the modules, a compiler wrapping the linking to the MPI libraries will be available from the command line. In particular, if you want to compile a C or a Fortran code, you will write, respectively:

```
mpicc mycode.c -o mycode.x
mpif90 mycode.f90 -o mycode.x
```

Since the resources requested in this exercise are poor, you can run your code on the login node (note that this is STRONGLY forbidden when you are running an ordinary application). You can do that typing:

```
mpirun -np N ./mycode.x
```

Where N is the number of MPI tasks, with a maximum value (on the login node) of 12. If you want to run on more processors, exploiting the backend nodes, please ask.