Foundations of high performance computing: final assignment

Davide Sancin

April 15, 2023

Exercise 1

1.1 Introduction

The aim of this exercise is to implement a parallel version of Conway's Game of Life, a cellular automation which plays on an infinite 2D grid. Two different versions of the game evolution are considered:

- static evolution: the system is stopped at a state and then all the cell are evolved simultaneously to the next state.
- ordered evolution: cells are evolved one at a time, in row major order.

Only the static evolution can be parallelized, the ordered evolution is serial by definition, since changing a cell state impacts the state of adjacent cells.

1.2 Methodology

The exercise was solved using a hybrid MPI/OpenMP program written in C language.

1.2.1 Parallelism methodology

The data has to be divided between MPI processes and OpenMP threads. The grid will be divided into chunks of rows, so that each process will work on a given sub-grid; if the groups of rows is not equally divisible by the number of processes, the last process will take care of the remaining rows. Each process then spawns a number of threads, each of which will work on single cells to compute the operations needed to perform the evolution; in particular each thread will be assigned a portion of the grid and will then calculate the number of alive neighbours of all the cells in said portion.

1.2.2 MPI methodology

To calculate the number of alive neighbours, a thread needs information on the state of all eight neighbouring cells, but cells on the border of the sub-grids don't have a full neighbourhood. This issue is solved modelling the grid as a torus

that wraps around in the top, the bottom and the sides; to do so ghost rows and columns are created, they are copies of the rows and columns on the opposite side of the edges of the grid. The computation of ghost rows is complicated by the fact that the opposite rows are stored in processors on other nodes/sockets, so it is necessary to exchange messages.

To transmit the rows the MPI_Send and MPI_Receive commands are used. The first row is sent to the upper process and the lower ghost row is received from the lower process, moreover the last row is sent to the lower process and the upper ghost row is received from the upper process. To compute the ghost columns, it is first necessary to exchange ghost rows, since they are needed to determine the value of the corners; then the right column will be the left ghost column, and the left column will be the right ghost column.

1.3 Implementation

In this section the most important parts of the code are presented.

1.3.1 Domain decomposition

The process 0 (master process) get the number of rows and columns of the grid from the initial PGM image, and it sends these values to the other processes then all processes then calculate the number of rows and columns on which they will work (figure 1.1). Finally the master process read the PGM image in an array and then send the chunks of rows to the other processes, keeping one for itself (figure 1.2).

```
100   int rows;
101   int cols;
102
103   //process 0 get the number of rows and columns of image and sends them to other processes
104   if (rank==0){
105     rows=read_rows(file_name);
106     cols=read_cols(file_name);
107     //cycle to send
108     for (int i=1; i<size; i++){
109       MPI_Send(&rows, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
110       MPI_Send(&cols, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
111     }
112   }else{      //other processes receive
113     MPI_Recv(&rows, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
114     MPI_Recv(&cols, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
115   }
116
117   int grid_size=rows*cols;    //dimension of the grid
118   int local_rows=rows/size;   //number of rows on which each process will work
119   int local_cols=cols;        //number of rows on which each process will work
120
121   //check if the number of rows is divisible by number of processes, if not the last process will work on remaining rows
122   int offset=rows%size;
123   if (rank==size-1){
124     local_rows=local_rows+offset;
125   }
126
127   int local_grid_size=local_rows*local_cols;      //size of the local grid of each process
128
129   //sizes including ghost rows and columns
130   int local_rows_ghost=local_rows+2;
131   int local_cols_ghost=local_cols+2;
132   int local_grid_size_ghost=local_rows_ghost*local_cols_ghost;
```

Figure 1.1: Domain decomposition first part

```
134    //get the processes that have neighbouring rows of current process
135    int upper_rank=(rank==0)? size-1 : rank-1;
136    int lower_rank=(rank==size-1)? 0 : rank+1;
137
138    //memory allocations
139    int *local_grid_current=(int *) malloc(local_grid_size*sizeof(int));        //local grid
140    int *local_grid_ghost=(int *) malloc(local_grid_size_ghost*sizeof(int));    //local grid whith ghosts
141    int *local_grid_next=(int *) malloc(local_grid_size_ghost*sizeof(int));      //local grid after evolution
142
143    //process 0 reads image, send local grids to other processes and keep one for itself
144    if (rank==0){
145      int *full_grid_current=(int*) malloc(grid_size*sizeof(int));
146      readin_array(full_grid_current, file_name, rows, cols);
147
148      for (int i=0; i<local_grid_size; i++){
149        local_grid_current[i]=full_grid_current[i];
150      }
151
152    //send local grids to other processes
153      for (int i=1; i<size; i++){
154        if (i<size-1){
155          MPI_Send(&full_grid_current[i*local_grid_size], local_grid_size, MPI_INT, i, 0, MPI_COMM_WORLD);
156        }else{
157          MPI_Send(&full_grid_current[i*local_grid_size], (local_rows+offset)*local_cols, MPI_INT, i, 0, MPI_COMM_WORLD);
158        }
159      }
160
161      free(full_grid_current);
162    }
163    //receive the local grids from process 0
164    if (rank !=0){
165      MPI_Recv(&local_grid_current[0], local_grid_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
166    }
167
168    //copy the local_grid into the local grid with ghost rows/cols
169    for (int i=0; i<local_rows; i++){
170      for (int j=0; j<local_cols; j++){
171        local_grid_ghost[(i+1)*local_cols_ghost+(j+1)]=local_grid_current[i*local_cols+j];
172      }
173    }
174
```

Figure 1.2: Domain decomposition first part

1.3.2 Message passing

After the domain decomposition processes exchange the ghost rows and compute the ghost columns and rows (figure 1.3)

```
10 void exchange_ghost_rows(int *local_grid_ghost, int local_rows_ghost, int local_cols_ghost, int upper_rank, int lower_rank){
11   MPI_Request request1, request2;
12   MPI_Isend(&local_grid_ghost[local_cols_ghost], local_cols_ghost, MPI_INT, upper_rank, 0, MPI_COMM_WORLD, &request1);
13   MPI_Irecv(&local_grid_ghost[(local_rows_ghost-1)*local_cols_ghost], local_cols_ghost, MPI_INT, lower_rank, 0, MPI_COMM_WORLD, &request1);
14   MPI_Isend(&local_grid_ghost[(local_rows_ghost-2)*local_cols_ghost], local_cols_ghost, MPI_INT, lower_rank, 1, MPI_COMM_WORLD, &request2);
15   MPI_Irecv(&local_grid_ghost[0], local_cols_ghost, MPI_INT, upper_rank, 1, MPI_COMM_WORLD, &request2);
16   MPI_Wait(&request1, MPI_STATUS_IGNORE);
17   MPI_Wait(&request2, MPI_STATUS_IGNORE);
18 }
19
20 //compute ghost columns
21 void compute_ghost_cols(int *local_grid_ghost, int local_rows_ghost, int local_cols_ghost){
22   for (int i=0; i<local_rows_ghost; i++){
23     local_grid_ghost[i*local_cols_ghost]=local_grid_ghost[(i+1)*local_cols_ghost-2];
24     local_grid_ghost[(i+1)*local_cols_ghost-1]=local_grid_ghost[i*local_cols_ghost+1];
25   }
26 }
27
28 //compute ghost rows
29 void compute_ghost_rows(int *grid, int rows, int cols,  int rows_ghost, int cols_ghost){
30   for (int i=1; i<=cols; i++){
31     grid[i]=grid[cols_ghost*rows+i];
32     grid[cols_ghost*(rows_ghost-1)+i]=grid[cols_ghost+i];
33   }
34 }
```

Figure 1.3: Message passing

1.3.3 Evolution

Finally the evolution, in this case static, is performed by counting the number of alive neighbour in each cell, the next state of a cell is found by applying the rules of the game and it is saved in another grid to preserve the staticness. The

for loop in which the rules are applied is managed by OpenmMP, so multiple threads works simultaneously on different sections of the loop (figure 1.4).

```c
//count the number of alive neighbours
int alive_neigh(int *grid, int i, int j, int cols){
  //this compute the sum of the elements of all neighbouring cells
  int neighbours=grid[(i-1)*cols+(j-1)] + grid[(i-1)*cols+j] + grid[(i-1)*cols+(j+1)]+
                 grid[i*cols+(j-1)]                          + grid[i*cols+(j+1)]+
                 grid[(i+1)*cols+(j-1)] + grid[(i+1)*cols+j] + grid[(i+1)*cols+(j+1)];
                       //i,j row/cols index  cols is length of row



  return (8-neighbours/255);   //since dead=255 and there are 8 neigbours, the number of alive neighbours will be 8-neighbours/255
}

//apply static evolution algorithm
void static_evo(int *grid, int *grid_next, int rows, int cols){
  #pragma omp parallel for schedule(static)
  for (int i=1; i<rows-1; i++){
    for (int j=1; j<cols-1; j++){
      int count=alive_neigh(grid, i, j, cols);
      if (grid[i*cols+j]==ALIVE && (count==2 || count==3)){
        grid_next[i*cols+j]=ALIVE;
      }else if(grid[i*cols+j]==DEAD && count==3){
        grid_next[i*cols+j]=ALIVE;
      }else{
        grid_next[i*cols+j]=DEAD;
      }
    }
  }
}
```

Figure 1.4: Evolution

1.4 Results and discussion

For the static evolution, some tests were performed to evaluate the performance of the code using different OpenMP/MPI options and using different sizes of the grid. In particular the measurement were done to assess various scalings:

- OpenMP scalability.
- MPI weak scalability.
- MPI strong scalability.

The metrics used to compare the performances are the execution time and the speedup, which is defined as the ratio of the execution times using different numbers of workers.

1.4.1 OpenMP scalability

To study the OpenMP scalability a single task was allocated on a node, and the number of threads was increased at each run. The test was done using the OpenMP option OMP_PROC_BIND=close and the mpirun option –map-by socket. Two different grid sizes (5000 and 10000) were used, both evolution iterate 100 times.

In figure 1.5 it is possible to see the result of the test. up to 32 threads the speedup scale well, then it slowly increase up to 64 threads. As expected the speedup is bigger for the bigger grid size.

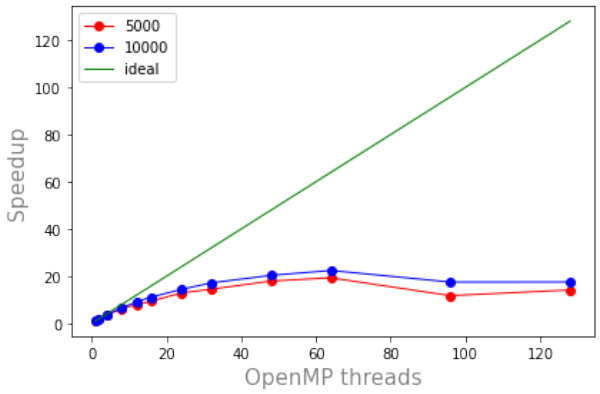

Figure 1.5: OpenMP scalability

1.4.2 MPI weak scalability

To study the MPI weak scalability the workload is maintained constant for each task, so starting from an initial grid size the number of cells has to be doubled for each task used; starting with one task and with a grid size of 5000x5000, the dimension are listed in table 1.1. The test was ran using 64 threads (for each

MPI tasks	Grid size
5000	1
7071	2
8660	3
10000	4
11180	5
12247	6

Table 1.1: Grid size for each number of MPI task

process) and then using just one thread. In the ideal case the run time would remain constant, but as it is possible observing in the figures 1.6 and 1.7, in both cases the run-time slightly increase with the number tasks, this is probably due to the fact that also the overhead increase with the number of tasks.

1.4.3 MPI strong scalability

To study the MPI strong scalability the size of the grid is maintained constant and the number of tasks is increased at each run. The test was run on various grid sizes, using 64 threads and using just one thread. Using 64 threads the mpirun option –map-by socket was used, while when using one thread the mpirun option –map-by core was used. For the one thread case, in figure 1.8 it is possible to see that the speedup scale well up to 128 tasks since only one node is used, with more than 128 tasks the other nodes are also used hence the

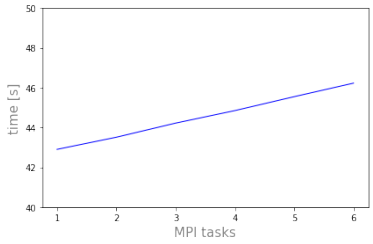

Figure 1.6: MPI weak scaling using 1 thread

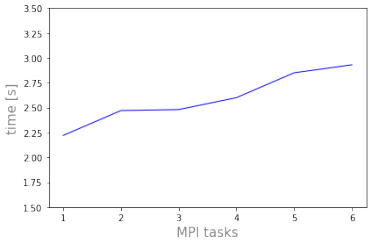

Figure 1.7: MPI weak scaling using 64 threads

behaviour of the speedup; for the 64 threads it is possible to see in figure 1.9 that the speedup scale well, also when using more than one node. In both cases there is no increase in the speedup with the increase of the size of the grids, this is probably due to an increase in message overhead.

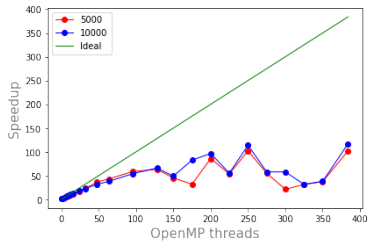

Figure 1.8: MPI strong scaling using 1 thread

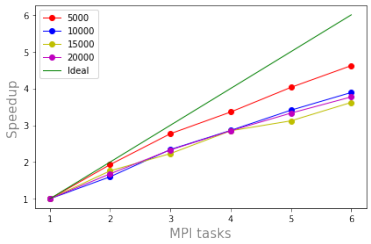

Figure 1.9: MPI strong scaling using 64 threads

1.5 Conclusions

In conclusion, it is possible to say that both MPI and OpenMP scale well enough, but only with OpenMP the expected increase in speedup for the increased grid size is observed, this probably means that increasing the tasks the message overhead and other non-computational overheads increase too much. This problem could be solved optimizing the communication and synchronization between processes. Finally it could also be possible to improve the algorithm used for the static evolution, this implementation, as it can be seen in figure 1.4, uses three if with two conditions each, this slows down the flow of the program, implementing the rules in other ways it could be possible to reduce the number of if conditions.

Exercise 2

The aim of this exercise is to compare the performances of the math libraries OpenBlas and MKL on the epyc node. In particular the size scaling and the core scaling are analyzed; for the study of the size scaling the number of floating point operations per second is considered, while for the study of cores scaling the speedup is considered.

2.1 Size scaling

The size of the square matrix was increased from 2000x2000 to 20000x20000 by steps of 1000. The scaling was tested using both single and double precision, and changing various arguments of the places (cores, sockets, threads) and bind (close, spread) OMP parameters.

2.1.1 Single precision

For the single precision the expected peak performance on an EPYC node is $5324.8 GFlops$, found considering 64 cores each performing 32 single precision operations per cycle with a maximum frequency of $2.6 GHz$.

Places=sockets

In the figures 2.1 and 2.2 it is possible to observe the results obtained using the option OMP_PLACES=sockets. The performances are similar for both libraries, and there are no significant differences between the two bindings; it is possible to observe some irregularities as the size increases.

Places=cores

In the figures 2.3 and 2.4 it is possible to observe the results obtained using the option OMP_PLACES=cores. Using the option bind=close (figure 2.3) both libraries behave similarly for smaller matrices, but as the size increases the OBLAS library performs significantly better. Using the option bind=spread, for small matrices the MKL library outperforms the OBLAS one by more than 1000 GFlops, for bigger matrices both libraries performs similarly, with the OBLAS one being slightly better.

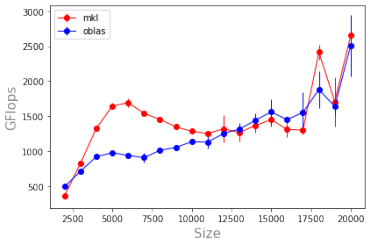

Figure 2.1: places=sockets and bind=close

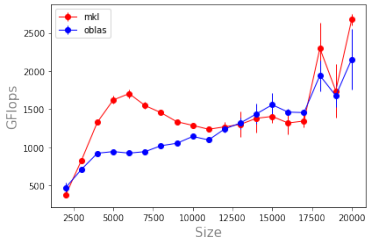

Figure 2.2: places=sockets and bind=spread

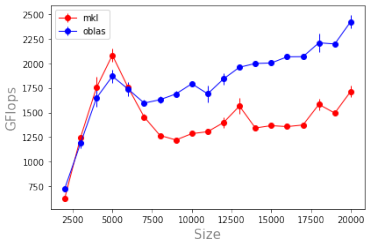

Figure 2.3: places=cores and bind=close

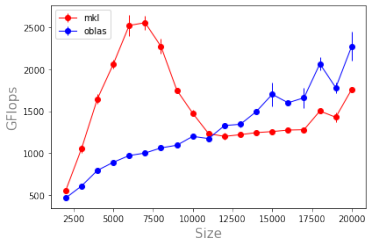

Figure 2.4: places=cores and bind=spread

Places=threads

In the figures 2.5 and 2.6 it is possible to observe the results obtained using the option OMP_PLACES=threads. The results are compatible withe the ones observed in the places=cores case, this is due to the fact that on the node where the tests were run the simultaneous multithreading is not enabled.

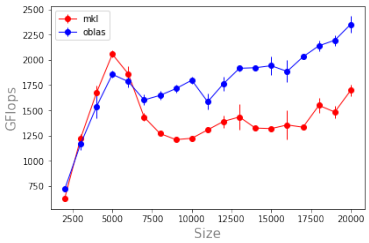

Figure 2.5: places=threads and bind=close

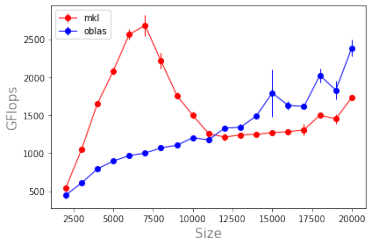

Figure 2.6: places=threads and bind=spread

2.1.2 double precision

For the double precision the expected peak performance on an EPYC node is 2662.4$GFlops$, found considering 64 cores each performing 16 double precision operations per cycle with a maximum frequency of 2.6GHz.

Places=sockets

In the figures 2.7 and 2.8 it is possible to observe the results obtained using the option OMP_PLACES=sockets. Analogously to the single precision case the binding option does not affect the performances; as expected the observed peak performance is lower than the single precision case.

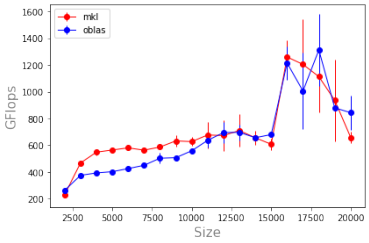

Figure 2.7: places=sockets and bind=close

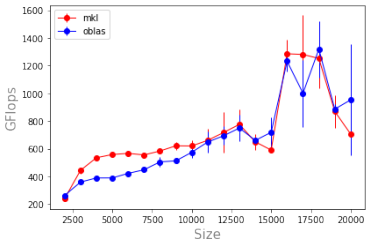

Figure 2.8: places=sockets and bind=spread

Places=cores and Places=threads

In the figures 2.8 and 2.9 it is possible to observe the results obtained using the option OMP_PLACES=cores, and in the figures 2.10 and 2.11 it is possible to observe the results obtained using the option OMP_PLACES=threads. Analogously to the single precision case the places option does not affect the performances, the OBLAS library has a higher peak performance for both binding options

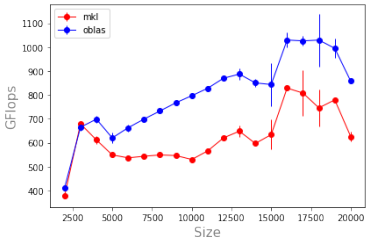

Figure 2.9: places=cores and bind=close

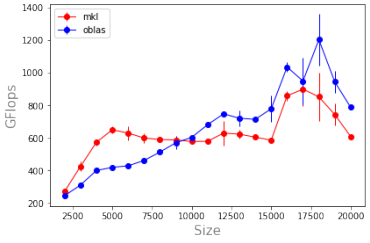

Figure 2.10: places=cores and bind=spread

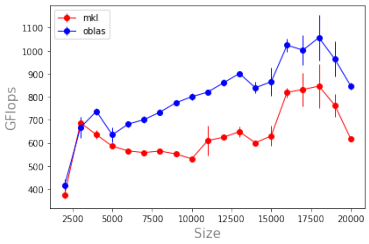

Figure 2.11: places=threads and bind=close

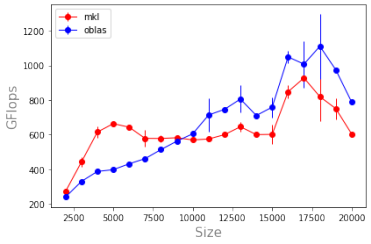

Figure 2.12: places=threads and bind=spread

2.2 Cores scaling

The size of the matrix was fixed at 15000x15000, while the number of OMP thread was gradually increased at each execution. The test were ran analogously to the study of size scaling, but the places=threads option was ignored since, as previously seen, is compatible with the places=cores options.

2.2.1 Single

In figure 2.13 the results for the bind=close option are presented, while in figure 2.14 the ones using the bind=spread can be seen. In the close case both libraries scale well up to 24 cores, then the OBLAS library with the place=sockets option has a lower speedup than all other cases, which have similar speedup. In the spread case both libraries scale well up to 24 cores and there are no significant differences between the places options, then the MKL library has a slightly greater speedup.

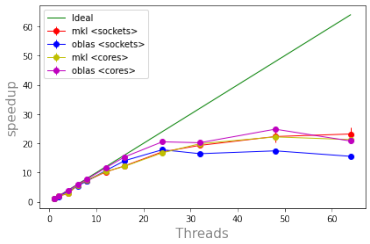

Figure 2.13: bind=close

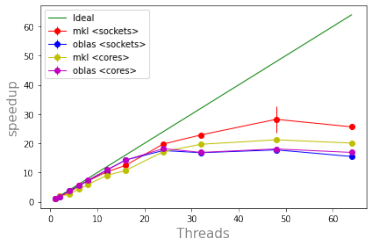

Figure 2.14: bind=spread

2.2.2 Double

In figure 2.14 the results for the bind=close option are presented, while in figure 2.15 the ones using the bind=spread can be seen. In both binding cases both

libraries scale almost ideally up to 16 cores, then the OBLAS library has a greater speedup.

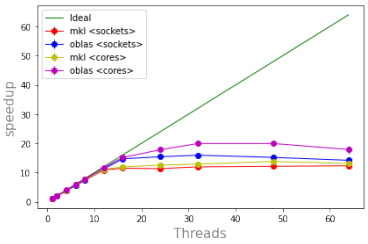

Figure 2.15: bind=close

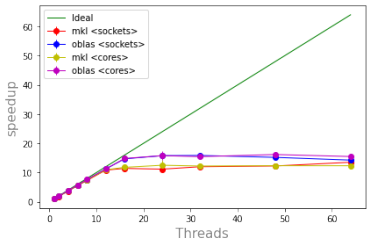

Figure 2.16: bind=spread