Assignment #2: Porting the lattice Boltzmann code to MPI

Name: *Christos Mourouzi,* ***User Id:*** *cm16663,* ***Canditate No.:*** *33747*

Introduction

The aim of this project is to parallelize the lattice Boltzmann, D2Q9 (BGK method) algorithm, by taking advantage of the distributed memory model of MPI (Message-Passaging-Interface). MPI is very popular in the HPC community as it is portable, has a lot of in-built routines that increases functionality and by the time of years has been declared as a standard message passing library[1].

The report begins with a brief description of some extra serial optimization applied to the code. Then, the parallel implementation of the code is explained and the effectiveness of other optimizations is analyzed. In the end, the runtime measurements of the serial code and the MPI code, executed with a several number of cores, are compared.

Serial Code Optimizations

- The double type variables are transformed into float type variables to minimize the computation of double precision complex equations, especially in the *rebound_collision()* function.

- The *tot_cells* value is calculated after the initialization of the *obstacles[]* array as it is a standard number which represents the number of cells that are not obstacles in the whole grid. It is no longer needed to re-compute the *tot_cells* value for each time step. This also helps the MPI implementation of the code as it reduces the need of another call of *MPI_Reduce()* routine (which is a collective-blocking communication call).

- The *propagate()* function is combined with the *rebound_collision()* function. Also, instead of writing speeds' values to each *tmp_cells[]* neighbour, the values of the appropriate speeds are written into a temp float array with a pull method procedure (speeds "come" to a cell, not "leaving" it). This modification improves the cache management (less cache hits as the code does not visit the memory as often as before). After the propagation step, rebound and collision step take place. At the end of each time step (at the end line of the main loop of the whole procedure) the pointer of the *cells[]* array is being swapped with the *tmp_cells[]* one. The cells array now points at the new *tmp_cells[]* data, and the *tmp_cells[]* array points at the cells array (which was not changed in the collision procedure). This swap happens in every iteration.

Compiler flags

The Intel Compiler 16-u2 (mpiicc) was chosen for the purposes of this project as it was faster than the GNU one (mpicc). Also by using -xAVX flag vectorization possibilities are enabled that makes the code run a little faster.

MPI Implementation

The main advantage of MPI is that it has a distributed memory model. That means that each MPI process does not share memory with the other processes (each core has its own shared memory and can not "see" any of the other cores' memories). Then, processes while working in parallel, communicate in a network by sending or receiving messages (data) with each other, in order to complete a task.

[1] https://computing.llnl.gov/tutorials/mpi/

For the giving code of D2Q9-BGK algorithm the procedure of the MPI Implementation is described as below:

1. First the *cells[]*, *tmp_cells[]* and *obstacles[]* arrays are initialized and the *tot_cells* value is being calculated.

2. Then the number of the columns of the initialized arrays, that each process will be responsible for, is calculated. This happens, by dividing the columns of the whole grid (*params.nx*) with the number of the processes (*size*). If there is a remainder from the division, then the processes with a rank less than the remainder get another column of the grid. Afterwards, each process copies its corresponding chunk from the initialized arrays to a local grid. The local array has two extra columns (one to the left and one to the right) as it is shown in Figure 1.

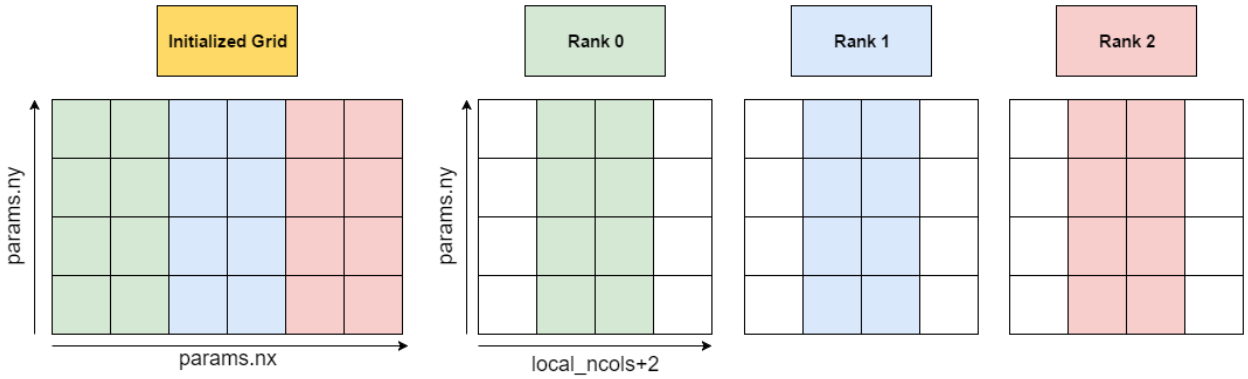

Figure 1. Simple example of the copy procedure. In a 4x6 grid, rank 0 gets the first two columns, rank 1 takes the next two columns and rank 2 gets the last two columns.

3. After the local grids are stored in each core's memory, *accelerate_flow()* and *propagate_rebound_collision()* are called for a number of time steps. Each process uses its own local array to make the computations needed for the algorithm. The main problem is that in the propagation step, speed values of the adjacent cells must be taken into consideration. Before propagation step takes place, each process sends its penultimate column cells' speeds 1 ,5 ,8 to the right process (rank+1) and its second column cells' speeds 3, 6 ,7 to the left process (rank-1). Then these are stored in the first and last column cells of the right and left neighbor process' local array respectively. Boundary conditions are considered for the MASTER and the last process only. This can be displayed graphically in Figure 2. Then, rebound and collision step take place as described in the Serial Code Optimization section.

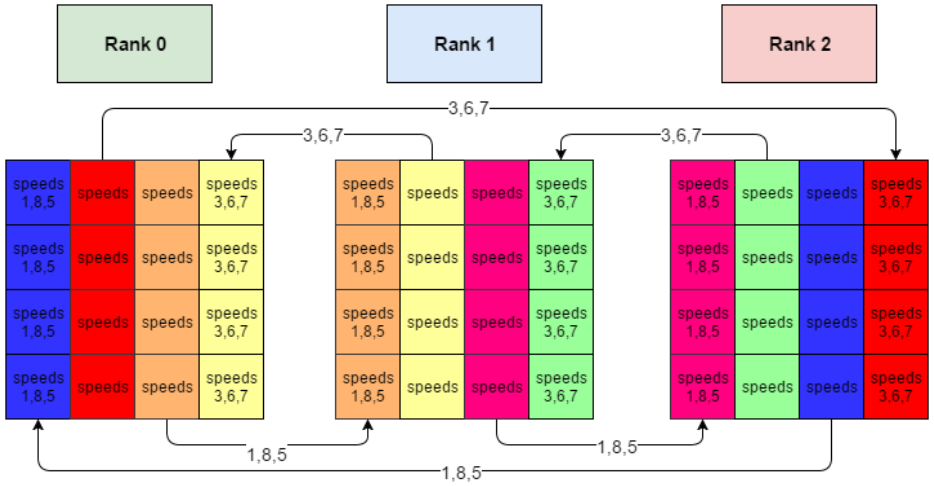

Figure 2. For the same examples as in Figure 1, in each time step, before propagation step begins each process send and receive the appropriate speed values to and from their neighbors.

4. When the whole procedure ends, each process sends its *local_cells[]* array to the MASTER (rank 0) process in order to be stored in the initial *cells[]* array to the appropriate position. Finally, the MASTER process takes responsibility of writing the result values to a file and printing out the execution time and the Reynolds number.

Parallel Code Optimizations

Exchange data before propagation step

First, an attempt with a halo-exchange method by using *MPI_Sendrecv()* routine was used for sending and receiving the appropriate data prior to the propagation step. *MPI_Sendrecv()* is a blocking communication mode and is just an extension of point-to-point communications. It has an advantage over the *MPI_Send()* and *MPI_Recv()* routines as it uses the same buffer for the send and for the receive, so that the message sent is replaced by the message received.[2] The message that was going to be exchanged was a float array with a size of *3*params.ny*sizeof(float)* containing the appropriate speeds (speeds 1,5,8 of each cell to the right process and speeds 3,6,7 of each cell to the left process). The results were great but another approach of non-blocking communication put into to test to compare which method is the fastest solution.

Grid	MPI_Sendrecv()	MPI_Isend() MPI_Irecv()
128x128	2.18 (s)	2.15 (s)
128x256	2.78 (s)	2.72 (s)
256x256	7.44 (s)	5.51 (s)
1024x1024	15.76 (s)	13.80 (s)

Table 1. Execution times for 64 processes

Instead of using *MPI_Sendrecv()*, an approach with *MPI_Isend()* and *MPI_Irecv()* routines was implemented. These routines are asynchronous and they return almost immediately. A non-blocking send call indicates that the system may start copying data out of the send buffer[3]. So, data (second and penultimate columns) should not be modified after an Isend operation is called, until the send completes. In order to achieve that, after *MPI_Isend()* and *MPI_Irecv()* were called, computations in all columns of each local grid took place, except in the second and in the penultimate column. Subsequently, *MPI_Wait()* was called to ensure that the messages were sent and received. Then, computations happened in the second and penultimate column separately. This approach was even faster (look at Table 1) as there were no blocking-communication delays. Finally, this method was chosen to be the appropriate one.

Computation of the average velocity in every time step

At the end of the *propagate_rebound_collision()* function, average velocity in each time step must be calculated. This assumes that *tot_u* value must be collected by all processes and then be summed. By using *MPI_Reduce()* all processes send their *tot_u* value to the MASTER which is summed and stored in the *total_u* variable of the MASTER process. In the end, only the MASTER is responsible to calculate the average velocity by dividing the *total_u* value with the already known *tot_cells* value and save it into the *av_vels[]* array.

Write the final cells array and print out the results

After all the time steps are finished, each process sends its final version of the *local_cells[]* array to the MASTER process by calling *MPI_Ssend()* (in order the call to be safe and no data are lost for any reason). Finally, the MASTER receives each chunk from each process, copies it to the appropriate position of the whole grid *cells[]* array and writes and prints out the final results.

[2] http://www.math.tu-cottbus.de/~kd/parallel/mpi/mpi-course.book_133.html
[3] https://www.open-mpi.org/doc/v1.8/man3/MPI_Isend.3.php

Experiments and results

The MPI implementation got into test for a several number of processors on nodes of BlueCrystal super-computer. Execution time of the program and correctness of the results were considered. All the results past the check scripts. The optimized serial code was executed by one processor for all four test cases (128x128, 128x256, 256x256, 1024x1024 grids). Then these times were divided by the times taken from the MPI runtimes on a number of cores, averaged over some number of runs. The division is equal to the speedup (xTimes) of each MPI runtime. The scaling graph below shows the speedup achieved for each test case in compare with the ideal one.

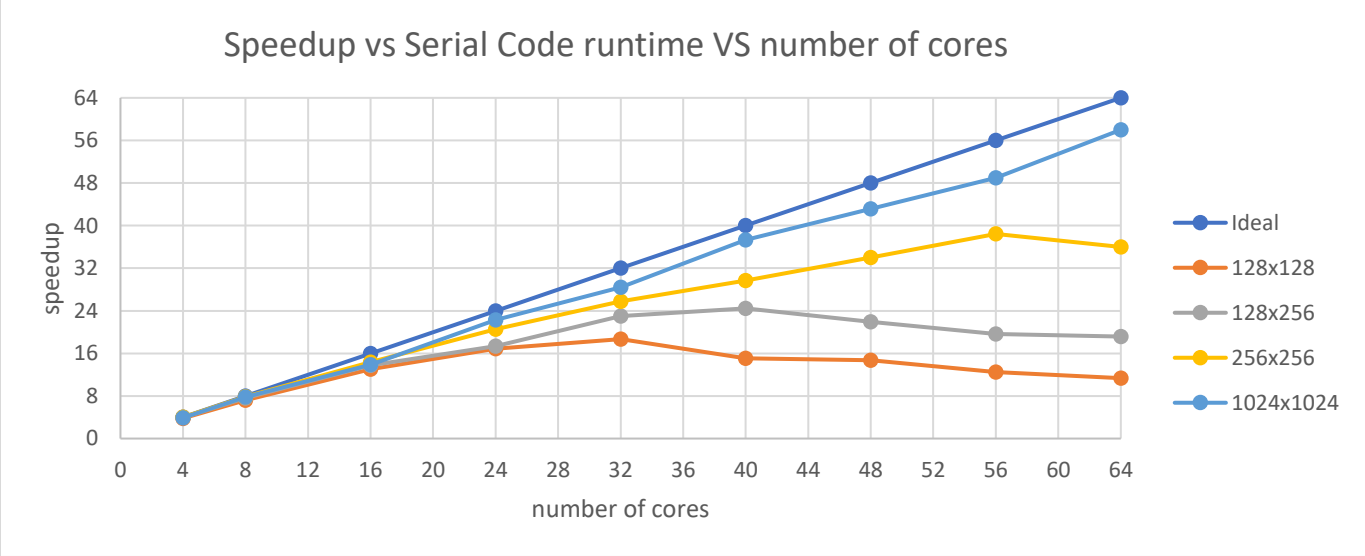

Figure 3. Compare of speedup vs serial code time VS number of cores used

As the table above suggests, great scaling was achieved in the big problem of 1024x1024 grid. In the cases of 128x128 and 128x256 grids, the speedup decreases after using more than 32 cores and 40 cores respectively. That happens due to synchronization and network latency issues. Those grids are divided in many small chunks. Every core has to do a small amount of job until receiving a message from its neighbours, and that causes some waiting until the neighbours send that message. Also, the number of nodes increases and that costs some extra network latency. In the case of the 256x256 grid, speedup decreases slightly, only in the case of using 64 cores. The best four execution times for all four test cases in compare with the serial code runtime are listed in the table below (in MPI Implementation row, the number of cores that achieved that time is mentioned in the brackets) .

	128x128	128x256	256x256	1024x1024
Serial Code	24.1 s	49.88 s	199.1	795.1 s
MPI Implementation	1.29 s (32 cores)	2.04 s (40 cores)	5.18 s (56 cores)	13.8 s (64 cores)

Table 2. Best execution times for all four test cases

Conclusion

The final results show how more effective the parallelism optimization of complex mathematic algorithms can be by using the MPI distributed memory model. MPI is a powerful tool for achieving this kind of results by taking advantages of the combined compute power of modern multi-core computer processors.