HPC Coursework 2: **Using Open MPI to Optimize The Lattice Boltzmann Program**

1.0 Introduction

This report looks into the optimizations implemented on a program that implements the **Lattice Boltzmann** scheme using the Open MPI libraries. The times mentioned in the report pertain to testing on the 128x128 parameters and obstacles files unless and otherwise mentioned.

2.0 Compiler Optimizations

The program without any optimizer flags runs to a time of **238.92s**. Since, there isn't a *–fast* flag on the *mpicc* compiler, we will use the *–O3* optimizer flag instead. When *–O*, *–O2*, and *–O3* were tested they all improved the time to a similar value, averaging to **105.62s**. We will try to further optimize the program serially before we approach parallel optimizations using Open MPI.

3.0 Serial Optimizations

3.1 Combining functions

An approach that I had not explored was the combining of functions. The *for* loops inside *rebound* and *collision* could be combined into function called *rebound_collision*. This however increased the run time of the program by one second. However when the program was run without any optimizer flags, the run time came to **227.31s**. Therefore, we have stumbled upon one of the many optimizations that *–O3* performs for us.

3.2 Converting Division into Multiplication

We could definitely optimize *Collision* by calculating the inverse of *c*_sq before the iteration and calculate the inverse of *local_density* before we calculate the velocity components (We wouldn't have to divide by local density twice). The original and the new equations are shown below:

$$\mathrm{d}_{\mathrm{equ[n]}} = \mathrm{w1} \times \mathrm{local}_{\mathrm{density}} \times \left(1.0 + \left(\frac{\mathrm{u[n]}}{\mathrm{c_{sq}}}\right) + \left(\frac{\mathrm{u[n]} * \mathrm{u[n]}}{2.0 * \mathrm{c\_sq} * \mathrm{c\_sq}}\right) - \left(\frac{\mathrm{u\_sq}}{2.0 * \mathrm{c\_sq}}\right)\right)$$

$$\mathrm{d}_{\mathrm{equ[n]}} = \mathrm{w1} \times \mathrm{local}_{\mathrm{density}} \times \left(1.0 + (\mathrm{u[n]} * \mathrm{inv\_c\_sq}) + \left(\frac{\mathrm{u[n]} * \mathrm{u[n]} * \mathrm{inv\_c\_sq} * \mathrm{inv\_c\_sq}}{2.0}\right) - \left(\frac{\mathrm{u\_sq} * \mathrm{inv\_c\_sq}}{2.0}\right)\right)$$

This could also be applied to *av_velocity* as it is very similar computationally to *collision*. After this was implemented the program has a run time of **45.73 s**. Now we can explore some of the other optimizations offered on the Intel compiler.

3.2 Using Vectorization with the Intel compiler

To invoke the highest level of Vectorization, the *–xHost* flag must be put as an option. After doing so we saw a very minute speedup to **45.58s**. Therefore, the code must be given more opportunities to vectorise. When a for loop was implemented in the calculation of $\mathrm{d}_{\mathrm{equ[n]}}$, no difference was made to the time. Therefore, it can be concluded that there are no other ways to take advantage of the vectorization capabilities of the intel compiler.

4.0 Structure of the Program with *MPI*

4.1 The *MPI* Overarching Structure with halo exchange in *main*

The basic structure of the MPI implementation of the Lattice Boltzmann program involves initializing MPI with *MPI_Init* right after *initialise* is called. This ensures that all the functions are run in all the processors available. The number of processors being used are **64** (**4** nodes) to ensure the fastest performance. The structure of the program has been largely changed to accommodate *MPI* and optimize its performance. Each processor available will compute all the functions on a particular number of rows from the *cells* structure. This division into rows happens before the multitude of iterations after which the chunks from the various ranks are put back together for calculations of the Reynolds number. The number of rows that each rank will accommodate is calculated by finding out the celling (the *ceil* function of *math.h* in *C*) of the division of the number of rows by the number of processors available. However the *obstacles* file has also been split into the rows that pertain to the rows of *cells* that belong to each rank. Note that this works for all the cases besides the **700 x 500** case, which will be discussed later. All of the smaller arrays still hold to the **1D** structure followed by the arrays within this program for simplicity. ***Figure 2*** shows which speeds are updated in respect to the green cell which is undergoing *propagate*. Since the program only needs rows above and below the particular row it is performing calculations on during *propagate*, we need a halo exchange right after *propagate* ever iteration. Two buffer rows have been created (orange rows in **figure 1**) above and below the rows contained by the rank to store the results of changes in speeds above the last row and below the first row. The speeds of **7**, **4** and **8** from are stored in a send buffer and sent to the left (the rank that has its rows to the left of this particular rank and therefore contains the rows below this one) with MPI_Sendrecv. A similar exchange is performed on **6**, **2** and **5** with the set of rows above. The rest of this section talks about how the different functions have been modified to accommodate for the changes that have been implemented to optimize with MPI.

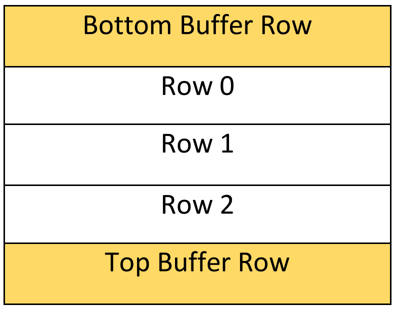


Figure 1 (tmp_cells)

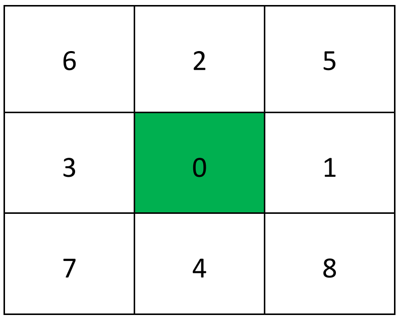


Figure 2 (Cells)

4.2 Changes in the functions with *MPI and the new data structure*

accelerate_flow deals only the second last row if the entirety of *cells* grid. This means that only the last rank is affected by this function. This is therefore one parallel optimization that is benefited by using MPI. Most of *propagate* has been discussed earlier since halo exchange is performed right after this function. However the calculation of the cell above (*north*) and below (*south*) have been changed not to involve *modulo* calculations as to accommodate the new data structure that contains extra buffer rows. There have been no significant changes in *rebound* or *collision* besides changing the boundaries of the variables that are incremented in the *for* loops.

4.3 *av_velocity* and the second instance of message passing

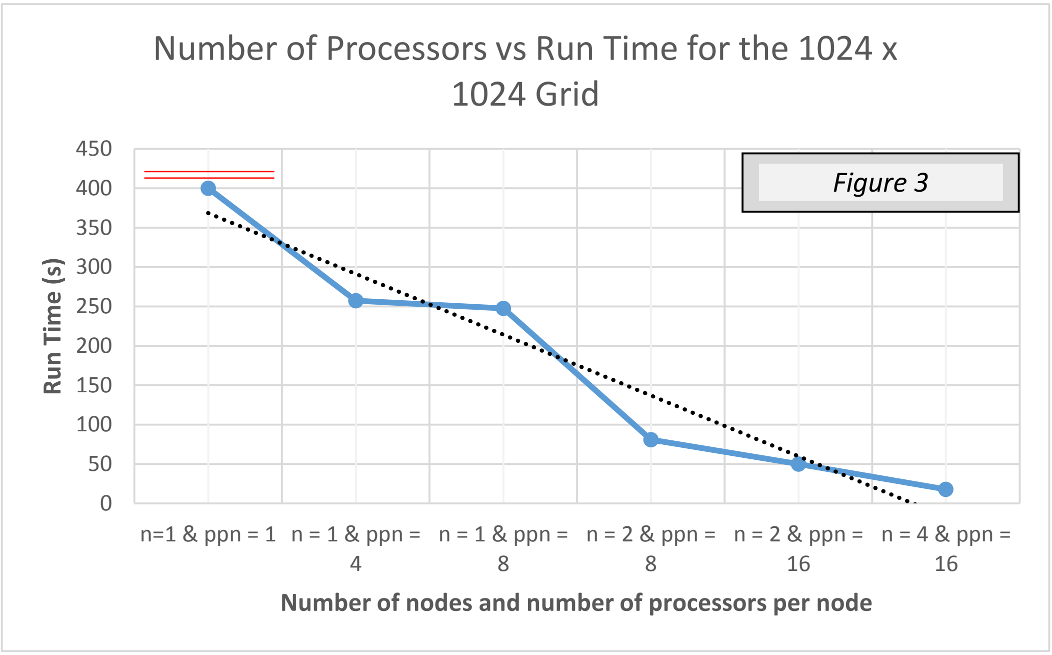
av_velocity mostly retained the calculations its serial counterpart has. However *tot_u* which was calculated and then divided by *tot_cells* was left as just *tot_u*. After this functions returns the value to *main*, this is sent to the master rank where it is added onto all the other values of *tot_u* from all the other ranks. *tot_cells* is calculated at the very start and therefore the value of av_vels for the particular iteration is calculated at the end of the respective iteration. MPI_Send and MPI_Recv have been used for the communication of the *double* between the ranks. The time it currently takes for this program to run the **1024x1024** grid is **17.96s**.

4.4 Tackling the 700 x 500 grid

Of the various input grids the program can process, 700x500 is a special case since the number of rows isn't divisible by 4. Therefore special changes are needed to be made to program to tackle this grid. The floor of the division of the number of rows in the cells grid (500) by the number of processors available is the value of number of rows for the smaller part of cells that is distributed to each rank. This is the case for all ranks besides the master rank. The master rank receives all the leftover rows which are exactly $500\ \%\ no.of\ processors$ (an optimal solution will be implemented when this division algorithm works). However this method does not work on the current implementation of the program and therefore optimizing the division of the leftover rows into the ranks is insignificant.

4.4 Scaling of the program according to the number of processors available

Figure 3 shows how the run time of the program with its current optimizations run with a different number of processors:



As it can be seen as the number of the processors changes the run time of the program significantly. Note the fourth data point is not the only answer tested for sixteen processors. When tested with sixteen processors on node instead of eight each on two there was a speedup of one second to **80.06s**. This shows that having more than one node induces latency and must not be the case unless a node is full. The first result crossed the **10:00** minutes time restriction on the job and therefore has been shown with a red bar as this would lead to a disproportionate portrayal of the other values. A black dotted trend line has been plotted on the graph showing that the doubling of the number of processors halves the amount of run time. Thus run time is ***inversely proportional*** to the number of processors being used by the program.

5.0 Parallel Optimizations

5.1 Investigating different types of message passing

Here we compare if using *MPI_Sendrecv* is faster than including individual *MPI_Send* and *MPI_Recv*. After modifying halo exchange to include *MPI_Send* and *MPI_Recv* the time was **17.96s**. Therefore there seems to be no difference between these two methods of message passing and therefore *Sendrecv* must constitute of an *MPI_Send* and *MPI_Recv* statement. And since the halo exchange is a recurring pattern the current implementation is a lot safer and more efficient than other point-to-point message passing options such as *MPI_Bsend* and *MPI_Isend*.

5.2 Including reduction into the program

MPI_Reduce can be used at the second instance of message passing where the value that returns at the *av_velocity* from each rank is sent to the master rank. After this was implemented we saw the run time actually decrease to **19.37s**. Since there wasn't a speedup when tree based load balancing was implemented into the program, we can conclude that ***bcp3*** has a pretty low latency value. *scatter_gather* could also be implemented into this stage. However, through empirical data reduction seems to be the most efficient method of message passing for this part of the program.

6.0 Conclusion

Through my investigations it is very clear that while serial optimizations are very important for the program, compilers and their optimizers will do most of these optimizations for the user. On the other hand there are many possibilities with a language such as Open *MPI* to parallelize the loops and further optimize the program. However, through empirical investigation, I can say that *MPI* is far more effective and useful for parallel optimizations. While *OpenMP* is very good at using the resources available, the inclusion of more than one node had no benefit to the efficiency (and actually hiders crashes the program as this is beyond the scope of this particular language). *MPI* however on the other hand made very effective use of the resources and has shown that the run time is inversely proportional to the number of processors used. There are many ways in which this program can be further optimized by *MPI*. For example, the *MPI-2* libraries contain many functions such as **RMA** (remote memory access) and the ability to partition the communicator. These optimizations add further credit to *MPI* over *OpenMP* and serial optimizations. My final time on the *1024 x 1024* case turned out to be **17.96s**.