MPP: Parallel Implementation of Percolate

B159973

November 27, 2019

Contents

1 Introduction **1**

2 Code Description **1**
2.1 User Inputs . . . 1
2.2 Memory Allocation . . . 1
2.3 Two-dimensional domain de-composition . . . 1
2.4 Scatter & Gather . . . 2
2.5 Update Iterative Loop . . . 3
2.5.1 Non-Blocking Halo Exchange & Updating Neighbors . . . 3
2.5.2 Map Average . . . 3
2.5.3 Stopping Criteria . . . 3

3 Testing **4**
3.1 Test Composition . . . 4
3.2 Test Method . . . 4

4 Measuring Parallel Performance **5**
4.1 Timers . . . 5
4.2 Performance Scaling . . . 5
4.3 Number of Steps . . . 5

5 Scaling **5**
5.1 Weak Scaling . . . 5
5.2 Strong Scaling . . . 6
5.2.1 Speed Up . . . 6
5.2.2 Efficiency . . . 6

6 Result Analysis **8**
6.1 Overhead Functions . . . 8
6.2 Real Parallel Part . . . 8
6.3 Parallel Cost Estimation . . . 9
6.4 Strong Scaling Analysis . . . 9
6.5 Weak Scaling Analysis . . . 9

7 Conclusions **10**

A Cost Estimation **11**

Note

This is the coursework of Message Passing Programming, Msc High-Performance Computing with Data Science, University of Edinburgh, 2019. The course is under the supervision of Dr David Henty.

1 Introduction

The primary purpose of the paper is to demonstrate the use of Message Passing Interface (MPI) on a given program that can only run on one processor, percolate [1]. By employing message-passing parallel techniques, the program must not only run successfully on multiple processes, but the duration of execution must decrease in total runtime. The underline mechanisms of message-passing techniques that the parallel version applies are explained in detail. Functional correctness of the parallel version is proved through regression testing.

The performance when running in parallel is quantified, by showing the speedup and parallel efficiency. Furthermore, two types of scaling, weak and strong, are reviewed. In performance investigation, Amdahl's Law [2] and Gustafson's Law [3] are referred, for revealing the limits on scaling on the particular program in theory. In the end, conclusions cover up the reasons the specific implementation has these performance results.

2 Code Description

2.1 User Inputs

The user can run the parallel version of percolate by entering the parameters in a configuration file and run a script or explicitly via the command line. The inputs are checked for validation, and that do not exceed boundaries. The grid size must be a positive integer, up to 9 digits. Allocation of an enormous map may crush the system.The RHO must be a valid fraction between 0 and 1, seed must be a positive integer and max clusters must be a positive integer, not exceeding L^2.

2.2 Memory Allocation

Memory allocation is dynamic for all the grids inside the program. Therefore the program is memory efficient and allocates only the memory it requires for running the simulation. The generated grids on runtime are map, small-map, old small-map and new small-map. The first has dimensions of L^2, while the others have $M * N$. The main function used for allocation was $arralloc()$ which allocates memory contiguous and MPI vector datatypes can be regularly used.

2.3 Two-dimensional domain de-composition

For solving the problem, two-dimensional domain decomposition was used. A two-dimensional virtual topology on an MPI communicator helped with the optimization of communication.
$MPI\_Dims\_create(Processes, Dimensions)$ creates a tuple of dimensions based on the given processes. For example, using two dimensions and Processes = 6, the function produces $Dims1 = 3$ and $Dims2 = 2$. The process number 5 can be interpreted as (3,1).

$$MPI\_Dims\_create(P, 2) = (Dims1, Dims2)$$

The special function *MPI_Cart_create()* produces a new communicator using cartesian topology based on $Dims1$ and $Dims2$. Every process can be interpreted as a tuple in the generated topology. Each process is assigned a 2D small map, a subset of the map, which has height M and width N. For example, process no. 5 needs to receive the portion of the map height which will be about $\frac{L}{3}$ and width to be $\frac{L}{2}$. A function $f(P)$ calculates the correct dimensions for each process:

$$f(P) = \begin{cases} M = \frac{L}{Dims1}, N = \frac{L}{Dims2} & \text{if } P \geq 2 \\ L = M = N, & \text{if } P = 1 \end{cases}, \text{ where } L \geq M, N \text{ and } P \geq 1. \tag{1}$$

2.4 Scatter & Gather

Scatter is implemented using non-blocking synchronous *Issend() and Irecv()*. The map is sliced using *MPI_Dims_create* dimensions and each process gets a precise proportion of the map. The data type that is passed in the MPI communicator is a vector that can be represented as a single column. For each slice, N columns are distributed by the master process to the corresponding worker processes (not broadcasted). The column has a size of one in width and M in length, where M is the particular length of the small-map that the process holds. As stated before, each M and N can be different, so the vector that is sent has to be configured with the correct length. In comparison with sending all the columns of the small map, rather than fitting the whole in one message, it might have more communication overhead, but the messages are smaller in size. Moreover, greater control is provided through this method of sharing. The current implementation can handle the uneven division of the grid and processes. If a process has a smaller M than the other ones, the number of columns that are sent is decreased. Gather works the same as scatter, with the difference that each column that is sent from the workers to the master process has to be placed in the correct position of the map.

$$PWidth = P \ \% \ Dims2 \ , PHeight = floor(\frac{P}{Dims2}) \tag{2}$$
$$startWidth = ceil(PWidth * N) \ , endWidth = ceil(PWidth * N + 1) \tag{3}$$
$$startHeight = ceil(PHeight * M) \ , endHeight = ceil(PHeight * M + 1) \tag{4}$$
$$endHeight \leq L \ , endWidth \leq L \tag{5}$$

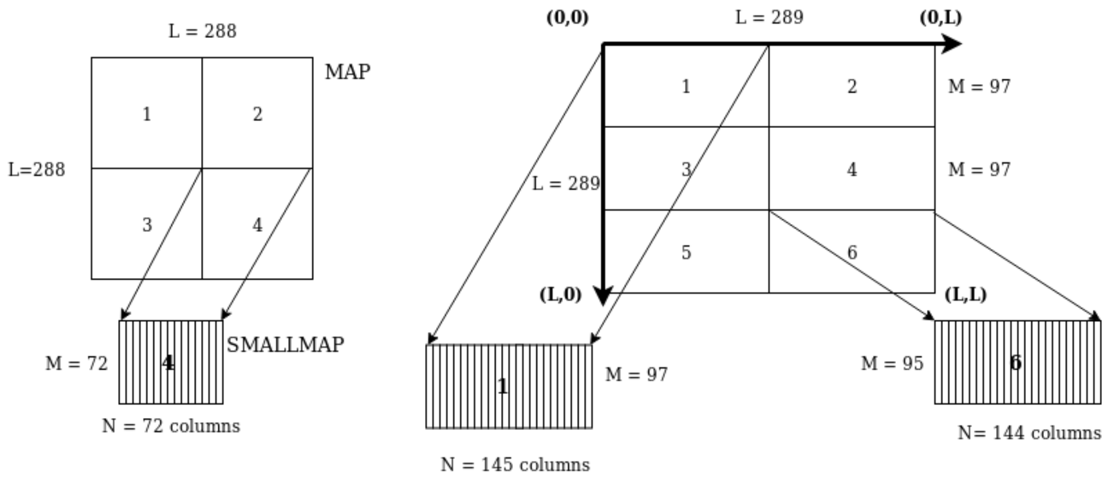

Figure 1: The 2D domain de-composition on a different number of processes. The particular slicing pattern enables the de-composition to be performed into any process domain. The height and width of each small map that a process handles are assigned dynamically by calculation based on L. The implementation convention is like a matrix where (0,0) is on the top left and (L,L) on the bottom right. For example, process 1 takes the portion of map starting from (0,0) until (97,145) while process 6 from (194,145) until (289,289). Process 6 should have M=97 but because it will exceed L, it is restricted to M = 95. Same happens with N. Each column has size $\frac{L}{Dims1}$ X 1.

2.5 Update Iterative Loop

2.5.1 Non-Blocking Halo Exchange & Updating Neighbors

Firstly, inside the update loop, the halo exchange takes place. Halos are shallow, which means that only two rows and two columns are passed to the neighbouring process. Halo exchange is done using non-blocking communication *Issend()and Irecv()*. When these are issued, there is no immediate waiting, but the inner cells (cells that are not in halos) are updated. This enables work to be done without waiting for halos to complete their exchange. Updating the neighbor is the same as in the serial version. Left, right, upper and bottom neighbor are found using *MPI_Cart_shift()*. If there aren't any neighbors, a *NULL* processor is returned, where send and receive complete without causing any trouble. After updating the inner cells, the waiting occurs using MPI $Waitall()$. After the exchange is done, the outer cells (halo cells) are updated. The updating follows periodicity on the first dimension.

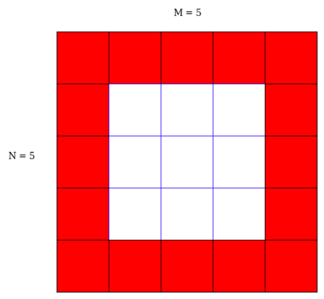

Figure 2: Shallow halo in red color of small-map 5x5. For each exchange two rows and columns are passed to the neighbor.

2.5.2 Map Average

After every neighbor is updated, the algorithm copies the new updated small-map into the old small-map. In that function, the sum of the small-map is calculated ($local\_sum$). Then, an *MPI_AllReduce* is issued so that every process calculates the average. The master process gathers all the local sums of each process, adds them into global sum ($global\_sum$) and finds the average by dividing (L^2) with . For every one hundred steps, the average is printed along with the number of changes.

2.5.3 Stopping Criteria

Additionally, the stopping criterion differs than the serial's version. In the serial version, a max step is declared and if only is exceeded the iterative loop stops, which it is inefficient and does not allow the executive time of a single step to be measured. Every process sends individually the number of the changes that happened on a single iteration *nchanges*. Individually, the process's update loop converges when the number of changes are zero. The primary concept is that if all the number of changes from every process are zero, then the entire map update converges. For proper terminating, *MPI_Allreduce()* is applied by sending reducing all the nchanges to another variable *stop*. If stop is equal to zero then the update stops for all the processes.

3 Testing

For proving functional correctness, the refactored parallel version can be executed with selected inputs on a controlled environment. The test assumes that the provided serial version outputs are correct and accurate.

3.1 Test Composition

The test is composed of a python unit test, *test_regression.py* which is found under /test folder. Furthermore, a helper library *regression.py* performs compilation and execution on of the above instances, along with output gathering and it is located under /lib folder. Finally, a batch script, *regression_test.sh* calls the unit test with the helper library. It is located in the root directory.

3.2 Test Method

The current testing method uses PYtest, a python [4] unit testing framework. Firstly, a batch script executes the parallel version with the configuration: L= 288, RHO = 0.411, Seed = 1564, Max clusters = 1. These are the same configurations which are hard-coded inside the serial version(except seed, which is inserted in the command line). The test assumes that those configurations cannot change in the serial version because essentially if they are changed, the serial version is modified and is no longer the original version given by the examiner. The only code that was added in the serial version was for implementing the periodic boundaries in the first dimension.

The batch script executes the parallel version 32 times, each time with a different number of processes (1,2,..,32). The specific interval (1 to 32) has prime numbers (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31) as well as evens and odds. Therefore the current test checks for the uneven decomposition of the map array during scattering and gathering on each process. After the execution, the outputs are saved in a folder. The helper library takes each parallel output and performs number filtering so that only numbers are collected. The same operation is done on the serial version.

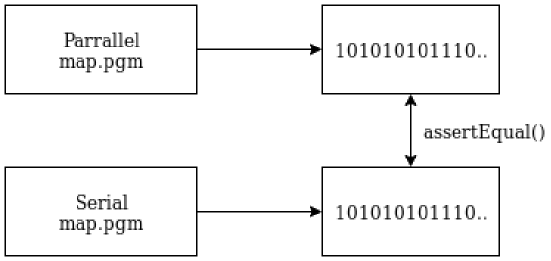

Figure 3: The two outputs are converted to a series of integers(0 or 1) into two different variables. After that, the function assertEqual() verifies the equality between them.

The output is a portable BitMap, where the main body of the file has only the values 0 and 1 (white & black). Next, the PYtest calls *assertEqual(a, b)* which checks that $a = b$, in this case $parallel\_map = serial\_map$. This comparison happens 32 times and the test framework generates an xml report that the test passed successfully. The current test ran 32 times (1-32 processes) on Cirrus[5], EPCC's supercomputer and successfully passed.

4 Measuring Parallel Performance

4.1 Timers

Two different timers measure times inside the program. The first is the TOTAL timer which measures the time of the entire program, and the second is UPDATE timer which measures only the iterative loop that finds the average and updates the map. Inside a static array, timers are being stored individually for each process. When timers are being dumped, MPI_reduce is called with a destination on the master rank to gather the elapsed times of all the processes. The approach above, calculates the average, minimum and maximum of UPDATE and TOTAL timers on every processes. As mentioned before, a step is a single iteration in the update loop. For measuring the time on each step, the number of steps is divided by the maximum process runtime, which is stored inside the UPDATE timer (max_update). Therefore the following statistics are exported from each run : avg_total, min_total, max_total, min_update, max_update, avg_update, time_step.

4.2 Performance Scaling

Two types of scaling are investigated using speedup: Strong scaling where the problem size is constant, and the number of processors is increasing and weak scaling where the problem size is growing in the same rate as the number of processors. It is essential to mention Amdahl's law and Gustafson's Law for correct performance profiling.

4.3 Number of Steps

It's very critical to mention that the number of steps is the same when running the parallel and the serial version. The number of steps is determined by grid size (L), the RHO and the seed. For example, with L=288, RHO=0.411, seed=1564 the problem demands about 1001 steps to solve. This indicates that 1001 times the nested for-loop will perform some computations on each cell in the grid of size L^2! Only this section of the program can be fully parallelized, as described later in the analysis. The number of steps will be the same, but the grid will be distributed into smaller chunks to the processes. Scalability is measured based on the duration of time of a single step. By proving that runtime does decrease in executing a single step on multiple processes, the entire program runtime execution reduces. For measuring scaling, the program was set to terminate after **100 steps** for not burning unnecessary CPU time. In the while loop of the update section, instead of trying to find if there are no local changes, the loop terminated when the number of steps reached one hundred.

5 Scaling

5.1 Weak Scaling

Weak scaling is described as measuring the runtimes of a program as the problem size (L) increases at the same rate as the number of processors. The ideal weak scaling is the runtime to stay constant as both variables increase. When conducting the experiment, the number of processes and the size of the grid was doubled every time for preserving a conventional pattern. A usual weak scaling behaviour is to exhibit an exponential inclination rather than being completely constant. This is commonly due to overhead functions and any other costs that the parallel implementation introduces.

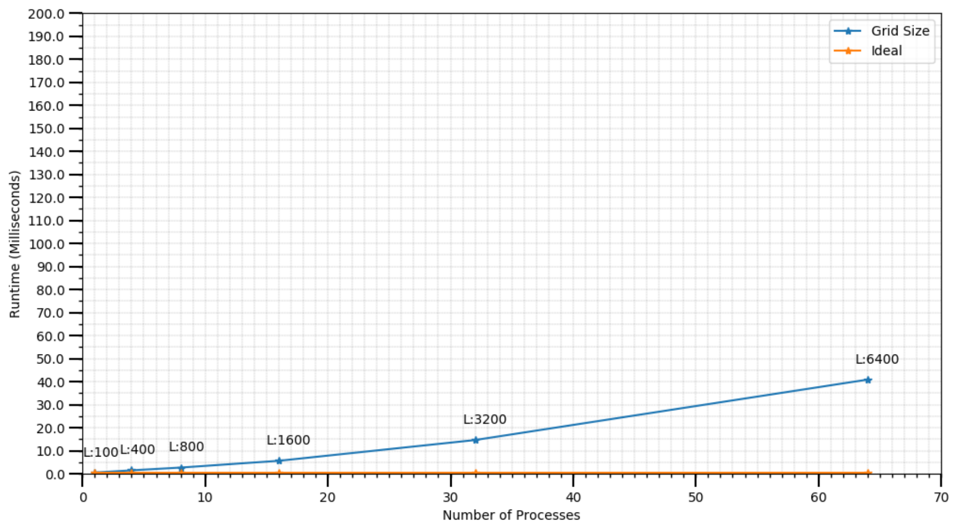

Figure 4: Runtimes in milliseconds for one hundred steps on L:100 & P:1, L:400 & P:4, L:800 & P:8, L:1600 & P:16, L:3200 & P:32, L:6400 & P:64, with RHO: 0.5. The problem input L and number of processors double every time. The figure reveals that indeed a sincere weak scaling is achieved. It worth to mention that for L:6400 and P:1 the execution took 1.21 seconds and about 40 milliseconds for P:64.

5.2 Strong Scaling

5.2.1 Speed Up

Strong scaling describes how the runtime of the program is affected when the problem size is fixed when executing on a different amount of processors. Normally it is more hard to achieve than weak scaling. The general equation for speed-up is dividing the serial total program runtime with the parallel version program runtime:

$$S(L,P) = \frac{Tserial(L,1)}{Tparallel(L,P)}$$

5.2.2 Efficiency

The efficiency estimates the portion of time for which a processor is utilized, and it is calculated by dividing speedup by the number of processors.

$$E = \frac{S(L,P)}{P}$$

All the runtimes can be seen in the folder under the name performance. Furthermore the python files that generated these graphs are included under the same folder with the batch scripts that collected the runtimes.

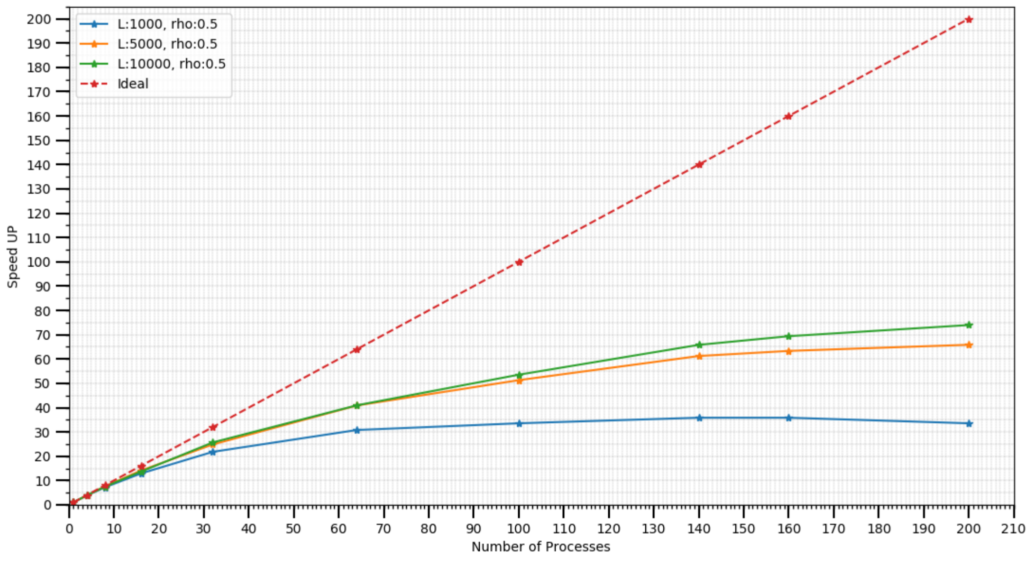

Figure 5: Speed Ups on three different grid sizes, L:1000, L:5000, L:10000 with constant RHO:0.5 based on one hundred steps. The number of processors is increasing but the problem size stays constant. The processes tested were: 1,4,8,16,32,64,100,140,160,200.

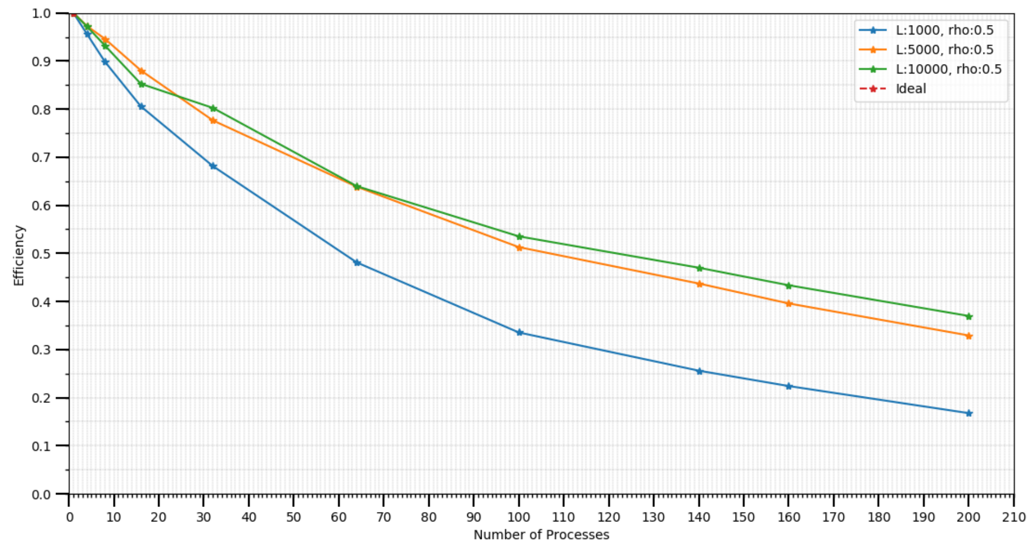

Figure 6: The efficiency of the parallel program on three different grid sizes, L:1000, L:5000, L:10000 with constant RHO:0.5 based on one hundred steps. The processes tested were: 1,4,8,16,32,64,100,140,160,200.

6 Result Analysis

6.1 Overhead Functions

For making a proper performance analysis, the abstract structure of the presented program must be noted:

```
Parallel Version
    main(){
        initialization()
        scatter()
        update_with_halos()
        gather()
        write_to_file()
    }
```

```
Serial Version
    main(){
        initialization()
        update()
        write_to_file()
    }
```

Scatter() and gather() are considered to be overhead functions because they are not included in the serial version. Furthermore update has an extra overhead, because for each step halos are being exchanged by each process. As specified earlier in the code description, the scatter() and gather() must send and receive $L * Dims1$ messages. Each column message has size $\frac{L}{Dims1}$ X 1. For each step two rows and two columns are being exchanged from each process the most, because some neighbors might not exist. Each row message has size 1 X $\frac{L}{Dims2}$. Therefore total messages for halo exchange might be about $2 * Steps * P$ where a message has size about 1 X $\frac{L}{Dims2}$ (row) and again $2 * Steps * P$ where a message is about $\frac{L}{Dims1}$ X 1 (column).

6.2 Real Parallel Part

Amdahl's law suggets that if the potential parallel part is fully parallelized then performance improvement is restricted by the propotion of the program which is serial:

$$T(L,P) = Tserial(L,1) + Tparallel(L,P) = aT(L,1) + \frac{(a-1)T(L,1)}{P} \tag{6}$$

$$S(L,P) = \frac{T(L,1)}{T(L,P)} = \frac{T(L,1)}{aT(L,1) + \frac{(a-1)T(L,1)}{P}} = \frac{P}{aP + (a-1)} \tag{7}$$

The fraction $a-1$ accounts for the parallel code of the program's runtime, which is the update loop. The structure of the update loop on each iteration is the following:

Algorithm 1 Update Loop

```
$M \leftarrow L/Dims[0]$
$N \leftarrow L/Dims[1]$
$sum \leftarrow 0, steps \leftarrow 0$
while $globalChange \neq 0$ do
    for $i \leftarrow 0, M$ do
        for $j \leftarrow 0, N$ do
            $sum \leftarrow sum + map[i][j]$
            $localChanges \leftarrow Update(upper, bottom, left, right)$
        end for
    end for
    $steps \leftarrow steps + 1$
    $globalChange \leftarrow calculateGlobal(localChanges)$
end while
```

The parallelizable segment of the program contains a nested for-loop. *Sum* is used to find the average and *Update(upper, bottom, left, right)* to update the cells. Each process is iterating from *0 to M* in the outer loop and from *0 to N* in the inner loop.

By using the (1) equation, the complexity is significantly decreased when running on more processes because it becomes:

$$\mathcal{O}(\frac{L^2}{P})$$

6.3 Parallel Cost Estimation

For calculating the cost of scatter and gather along with halo exchange, let's consider the following [6]:

$$Tparallel(L, P) = Tcomp(L, P) + Tcomm(L, P) \tag{8}$$

$$Tcomm(L, P) = tstartup + tbw \tag{9}$$

The parallel runtime can be written as the computation time plus the communication time. Scatter() and gather() along with halo exchange fall into $Tcomm(L, P)$. The $tstartup$ is the MPI latency which is considered to be half the time of a ping pong execution with message of size zero. It is the time the MPI needs to prepare the nodes to send a message. The tbw is the bandwidth of the message passing saturation, the division of the message size by the data rate communication of two MPI tasks.$Tstartup$ and tbw can be computed on the specific machine that the parallel code runs using ping pong tests.

$Tcomp(N, P)$ has the cost of $\mathcal{O}(\frac{L^2}{P})$. $Tcomm(N, P)$ has the cost of:

$$Tcomm(L, P) = TcommScatter(L, P) + TcommGather(L, P) + TcommHalos(L, P)$$

A rough abstract estimation is in the appendix [A].

6.4 Strong Scaling Analysis

Using the previous efficiency equation, the efficiency can be written as:

$$E = \frac{Tserial}{P * Tparallel}$$

If the problem size does not change and Tserial remains constant but each time more processors are added, Tparallel is increasing because the runtimes of overhead functions (*Tcomm*) tend to increase. The parallel part that can be fully parallelized is the update() segment. Furthermore, as the equations above mention, speed-up is limited by the serial part which in the aformented case is every segment except update() and during that time all the available extra processes must be idle. Therefore the exponential declination that goes towards zero on the efficiency figure is correct.

6.5 Weak Scaling Analysis

Gustafson's law supports that the the runtime of the serial section does not increase as opposed to Amdahl's law. The problem size increase but not the serial section.

Therefore the total runtime on running on a single processor is:

$$Tserial = T(1) \tag{10}$$
$$Tparallel = L * T(1) \tag{11}$$
$$Ttotal = a * Tserial + (1 - a) * Tparallel \tag{12}$$

and the running on multiple processors to be:

$$Ttotal = a * Tserial + \frac{((1 - a) * Tparallel)}{P} \tag{13}$$
$$\tag{14}$$

the speed up becomes:

$$SpeedUp(L) = \frac{a * Tserial + (1 - a) * Tparallel}{a * Tserial + \frac{((1-a)*Tparallel)}{P}} \tag{15}$$
$$\text{if } L = P \text{ then } L + (1 - L) * a \tag{16}$$

The runtimes on figure (4) follows this concept slightly different. As L and P increases and, the speed up should be linear. The ideal runtimes should be constant. In the figure the runtimes are being close to constant but they are slightly increasing exponentially.

7 Conclusions

It would be a utopia if always by increasing the parallel computation units the runtimes, decrease linearly. There are many factors that determine the efficiency of a parallel program, as discussed throughout the paper. Some of them are architecture-related, for instance, the number of CPUs, memory resources or even network components. The scalability of the application is very interesting. The algorithm that is used must be scalable to run on parallel machines. The I/O must be implemented in a sophisticated way and be as far as limited. The programmer must fully understand the MPI mechanisms, i.e. collective communications, non-blocking- communication e.t.c. For instance, the programmer must take advantage of non-blocking communication and make computations while waiting for the message to be received. Unfortunately, message passing programming introduces undesired latencies that derive from sending and receiving messages, as well as the bandwidth that is a mixture of both hardware and software factors. Through experiments, genuine weak and strong scaling is shown. Strong scaling is based on Amdahl's law which says that the serial segment of the program restricts the upper limit speedup. Therefore Amdahl's law assumes that given enhanced processing power, the speedup will remain the same at some point, while Gustafson argues that more parallel computing power while increasing the problem size the serial fraction becomes miniature, speedup is linear and the runtimes stay constant.

References

[1] Percolate Assignment
`https://www.learn.ed.ac.uk/webapps/blackboard/content/listContent`

[2] (1967) Gene M. Amdahl Validity of the single processor approach to achieving large scale computing capabilities
`https://dl.acm.org/citation.cfm?id=1465560`

[3] (1988) John L.Gustafson: Reevaluating Amdahl's Law:
`http://www.johngustafson.net/pubs/pub13/amdahl.pdf`

[4] (2019) Python: The pytest framework
`https://docs.pytest.org/en/latest/`

[5] EPCC Center(2019)
www.epcc.ed.ac.uk/facilities/demand-computing/cirrus Cirrus Facilities.

[6] (2014) Mary Thomas - Characterizing MPI Messaging and Communication Costs
https://edoras.sdsu.edu/ mthomas/docs/mpi/MPIComms.pdf.

A Cost Estimation

For the sake of simplicity in calculations, lets take only in account the processes that have principal square root (ex: 4,9,16..), the subsequent equality is occurring:

$$M = N = \frac{L}{\sqrt{P}}$$

Therefore the complexity on the fully parallelizable part of the program:

$$SizeInteger = 4 \tag{17}$$

$$ColumnSize = SizeInteger * \frac{L}{Dims1} \tag{18}$$

$$RowSize = SizeInteger * \frac{L}{Dims2} \tag{19}$$

$$HaloColsSize = Steps * ColumnSize * 2 * P \tag{20}$$

$$HaloRowwSize = Steps * RawSize * 2 * P \tag{21}$$

$$TcommScatter(L, P) = \mathcal{O}((L * \sqrt{P} * ColumnSize * tstartup) + \frac{L * \sqrt{P} * ColumnSize}{MPIBandwidth}) \tag{22}$$

$$TcommGather(L, P) = \mathcal{O}((L * \sqrt{P} * ColumnSize * tstartup) + \frac{L * \sqrt{P} * ColumnSize}{MPIBandwidth}) \tag{23}$$

$$TcommHalosRows(L, P) = \mathcal{O}((HaloRawSize * tstartup) + (\frac{HaloRowSize}{MPIBandwidth})) \tag{24}$$

$$TcommHalosCols(L, P) = \mathcal{O}((HaloColSize * tstartup) + (\frac{HaloColsSize}{MPIBandwidth})) \tag{25}$$

$$TcommHalos = TcommHalosCols(L, P) + TcommHalosRows(L, P) \tag{26}$$

$$Tcomp(L, P) = \mathcal{O}(\frac{L^2}{P}) \tag{27}$$

$$Tcomm(L, P) = TcommScatter(L, P) + TcommGather(L, P) + TcommHalos(L, P) \tag{28}$$

$$Tparallel(L, P) = Tcomp(L, P) + Tcomm(L, P) \tag{29}$$