Parallel Systems HW 3

December 2024

Introduction

For compilation, the makefile found in the initial directory was used, while the source code for each exercise is located under the corresponding directory **ask-X** Both programs are compiled with **make all** with the executable of each exercise being in its directory. The measurements were done automatically with python scripts in which, to calculate the time of each program, 5 executions were made and their average was taken. Their results are printed on the terminal in table format and saved in .csv files, which facilitated the process of measurements and production of graphical representations. In order to print the states of the tables in each program and not only the execution time, there are corresponding rules **make outX**, which produce executables that print these results to the terminal.

The measurements were made on the department's Ubuntu Linux machines, which have a CPU with 4 cores and an L1 cache-line of 64 bytes:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 94
model name      : Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
... skip ...
cpu cores       : 4
... skip ...
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual

$ gcc -v
... skip ...
Target: x86_64-linux-gnu
... skip ...
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.6 LTS (Focal Fossa)"
```

Flat Matrix

Both tasks in the assignment involve working with two-dimensional matrices, so it's important to note the method of allocation used. Instead of the classic approach of allocating memory iteratively in a for loop, we make a single malloc call at the beginning to allocate a large, contiguous block of memory of size `n * n * sizeof(int)`. Accessing the element at row `i` and column `j` is done using `A[i*n + j]`. This way, we perform only one allocation, and the matrix data is stored in consecutive memory locations, improving cache efficiency and making our lives a bit easier when it comes to MPI data distribution functions.

Exercise 3.1

Executions

To execute the parallel program of the exercise 3.1, after the compilation run the command:
`mpiexec -f ../machines -n <nodes> ./ask1/gol <grid_size> <generations>`

General Implementation Comments

We are now asked to solve the problem from the previous assignment in a distributed manner, where the main issue is that the matrix is no longer in shared memory and therefore we need to implement logic so that the processes now share parts of the matrix. We chose to do this in the simplest way, which is row-wise distribution. Specifically, each process is assigned a set of consecutive rows of the matrix, which are distributed using the `MPI_Scatterv()` call. Now, each process has its own submatrix and will proceed with computing the next generation. To generate the next state of a cell, it needs knowledge of all 8 of its neighboring cells, so there is an issue with the boundary cells of the first/last row, whose neighbors are located in a row belonging to another process. For this reason, before a process begins computing the next generation, it requests the neighboring rows and simultaneously sends its own boundary rows to its neighboring processes using the `MPI_Sendrecv()` call. Diagrammatically:

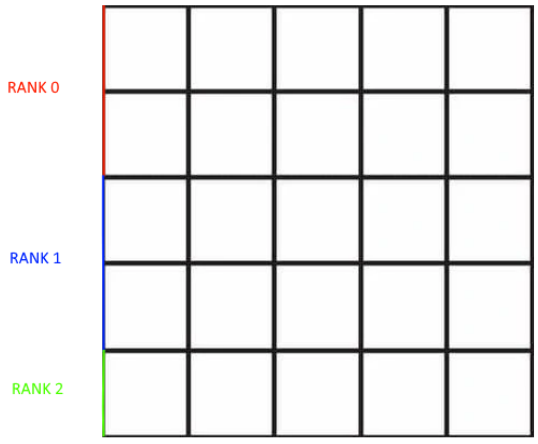


Figure 1: Work load distribution for N=5

In the local grid of each process, beyond the rows it manages, space is also allocated for the rows it receives from its neighbors. This also applies to the edge processes (rank 0 and rank n), since we applied the idea from the previous assignment: to avoid unnecessary checks, if a neighbor lies outside the boundaries of the matrix, the rows marked in blue are dummy rows that are always empty. The same logic is applied to the columns, as each row includes two extra empty cells, one at the beginning and one at the end, so they are always within bounds.

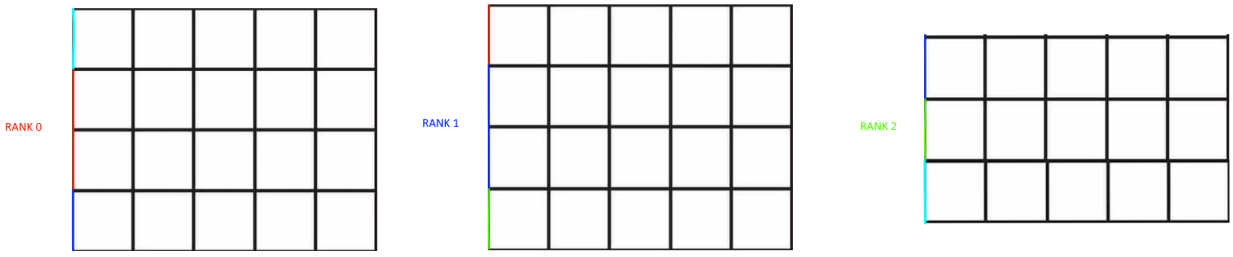
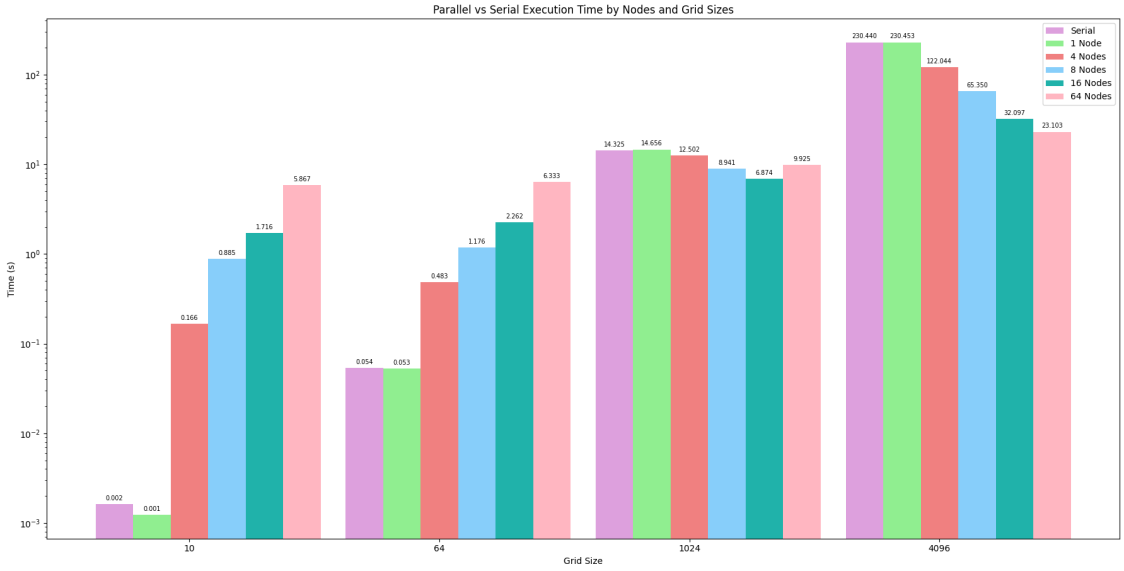


Figure 2: Local Grids

Once, after the exchanges, the process has the latest image of the matrix, it can proceed with the computation of the new generation in its local matrix. After computing all the generations, the root process performs `MPI_Gather()` to collect all the rows that were initially sent in order to print the final result.

Grid Size	Algorithm	Nodes	Scatter	Calc	Gather	Total Time
10	Serial	-	-	0.001617	-	0.0016
10	Parallel	1	0.000009	0.001218	0.000003	0.0012
10	Parallel	4	0.000013	0.165650	0.000004	0.1657
10	Parallel	8	0.002616	0.881797	0.000264	0.8847
10	Parallel	16	0.001182	1.715041	0.000111	1.7163
10	Parallel	64	0.001157	5.862638	0.003110	5.8669
64	Serial	-	-	0.053706	-	0.0537
64	Parallel	1	0.000011	0.053097	0.000005	0.0531
64	Parallel	4	0.000026	0.482623	0.000018	0.4827
64	Parallel	8	0.000706	1.175097	0.000215	1.176
64	Parallel	16	0.001564	2.258315	0.002229	2.2621
64	Parallel	64	0.012668	6.313858	0.006088	6.3326
1,024	Serial	-	-	14.324680	-	14.3247
1,024	Parallel	1	0.002521	14.326359	0.000285	14.3292
1,024	Parallel	4	0.052800	12.447588	0.001429	12.5018
1,024	Parallel	8	0.040952	8.876182	0.023797	8.9409
1,024	Parallel	16	0.042979	6.795991	0.035301	6.8743
1,024	Parallel	64	0.040419	9.653042	0.231488	9.9249
4,096	Serial	-	-	230.440023	-	230.44
4,096	Parallel	1	0.041610	230.406275	0.005524	230.4534
4,096	Parallel	4	0.089164	121.929285	0.025142	122.0436
4,096	Parallel	8	0.351118	64.608292	0.390935	65.3503
4,096	Parallel	16	0.493241	31.030203	0.573436	32.0969
4,096	Parallel	64	0.608353	21.834472	0.660267	23.1031



Conclusions

By observing and comparing the execution times of the serial program with those of the parallel version, we notice a clear performance improvement for large matrices when using the parallel algorithm. However, for small matrices (e.g., grid size = 10), increasing the number of nodes does not lead to a reduction in execution time. On the contrary, the time increases, likely due to the additional overhead introduced by the extra nodes. For example, for a matrix of size 10 and 1000 generations, the serial algorithm takes only 0.0016 seconds, while the parallel algorithm with 64 nodes takes 5.86 seconds. As previously mentioned, in cases of small workload, the cost of creating and managing multiple processes via MPI, as well as the load distribution, outweighs the benefits of parallel execution.

The performance improvement becomes evident for larger matrices, such as those with size n = 1024 or n = 4096. In these cases, we observe that the execution time of the program with 16 nodes and grid size = 4096 is reduced to less than one-seventh of the serial algorithm's time. For example, for n = 4096 and 1000 generations, the serial algorithm took about 230 seconds, while the parallel version with 16 nodes completed in just 32 seconds. It is also important to comment on the increase in time for scatter and gather operations with the increase in nodes for a fixed size. We see that to distribute and gather 4096 rows, there is a significant increase from 4 to 8 nodes (approximately from 0.05 to 0.35). This could be a simple measurement error, but we assume it has to do with how the nodes are distributed across the machines in the lab. The machines file specifies 4 processes per machine, so when we request 4, all 4 nodes may end up on the same machine, and thus the transfer is fast. However, when requesting 8 nodes, they may be spread across more machines, and communication occurs over the network instead of in-memory exchanges. Additionally, we notice that for the same number of nodes and the same grid size, the scatter and gather times are almost identical, with gather being slightly slower.

Finally, it becomes evident that the execution times of the serial program and the parallel program with 1 node are almost identical, with the parallel version being slightly slower. Specifically, we observe that additional time is spent on scatter and gather operations, even if nothing is actually sent. However, if we include the time required for the initialization of MPI in the total time of the parallel program, the serial program would appear slightly faster.

Below, we present the speedup for a matrix size of 4096:

comm_sz	T.Serial	T.Parallel	Speedup
1	230.44	230.453	0.999
4	230.44	122.043	1.888
8	230.44	65.350	3.526
16	230.44	32.096	7.179
64	230.44	23.103	9.974

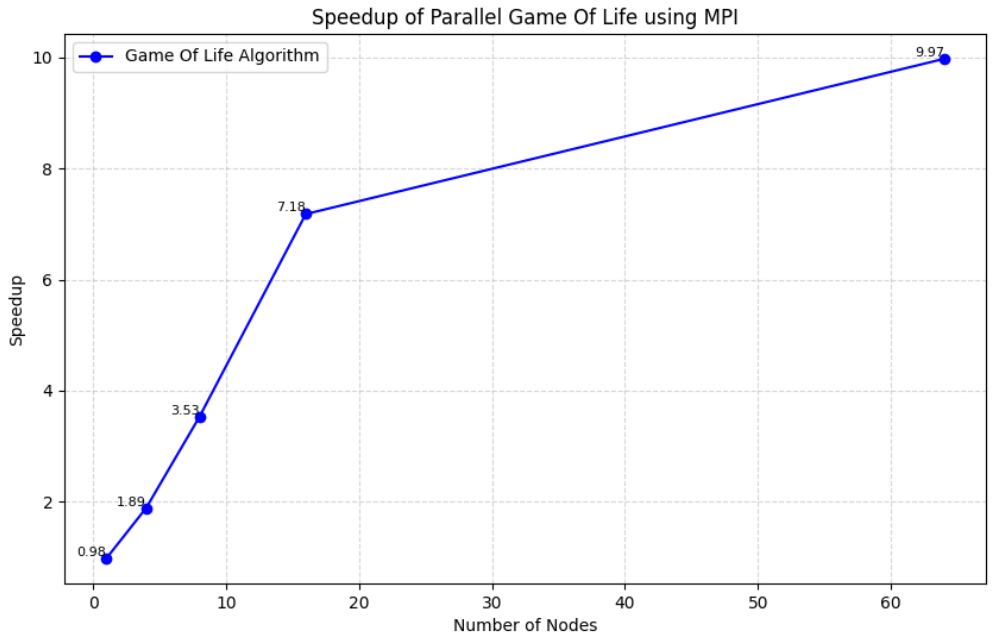


Figure 3: Speedup N=4096

By observing the speedup chart, we see that although it deviates from the ideal, it increases in a fairly linear manner. Specifically, we see that from 4 to 16 nodes, for approximately every doubling of the nodes, the speedup also doubles, but it always remains capped at half of the ideal (thus, we expect efficiency to be below 0.5, as we will see below). One explanation for this is that most of the computation time is spent on communication between the nodes, as every time we add a node, we also add two row exchanges per iteration, which seem to have significant impacts on performance. Also, for 64 nodes, even though a fairly large improvement is observed, it is not proportional to the improvement from the 16 nodes. We would expect at least a doubling of performance, but the increased communication seems to cost significantly, and that's why we see a drop in the speedup.

Efficiency:

comm_sz	10	64	1024	4096
1	1.31	1.011	0.977	0.98
4	0.0024	0.027	0.286	0.47
8	0.0002	0.005	0.20	0.44
16	0.00006	0.0014	0.13	0.45
64	0.000004	0.0001	0.022	0.15

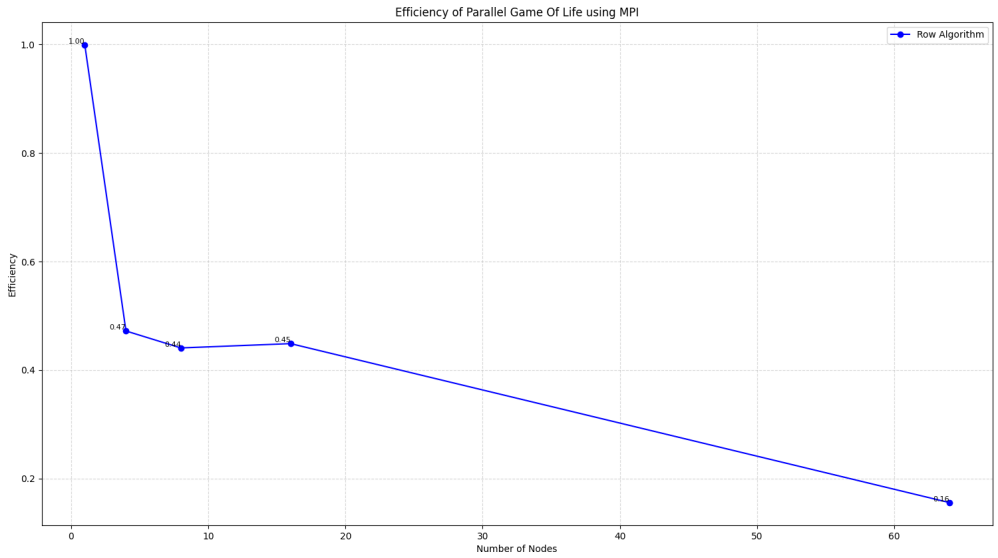


Figure 4: Efficiency N=4096

Initially, it should be noted that values above one are recorded for extremely small N, where the timing measurements may be inaccurate, but we see that for larger sizes, this is corrected. We observe that for small N, the performance is extremely low, as we would expect, since much more time is spent on communication than on computation. As the problem size increases, and thus the computations performed in parallel increase, the efficiency improves, which is logical.
By observing the efficiency chart for N=4096, we see that by increasing the nodes and keeping N fixed, after the initial drop from 1 node, the efficiency remains steady at around 0.45 for 4, 8, and 16 nodes. Therefore, we could perhaps characterize the problem/implementation as **strongly scalable**. However, as we described earlier, the efficiency seems to always be below half in all cases, and thus the communication between nodes outweighs the parallel computation.

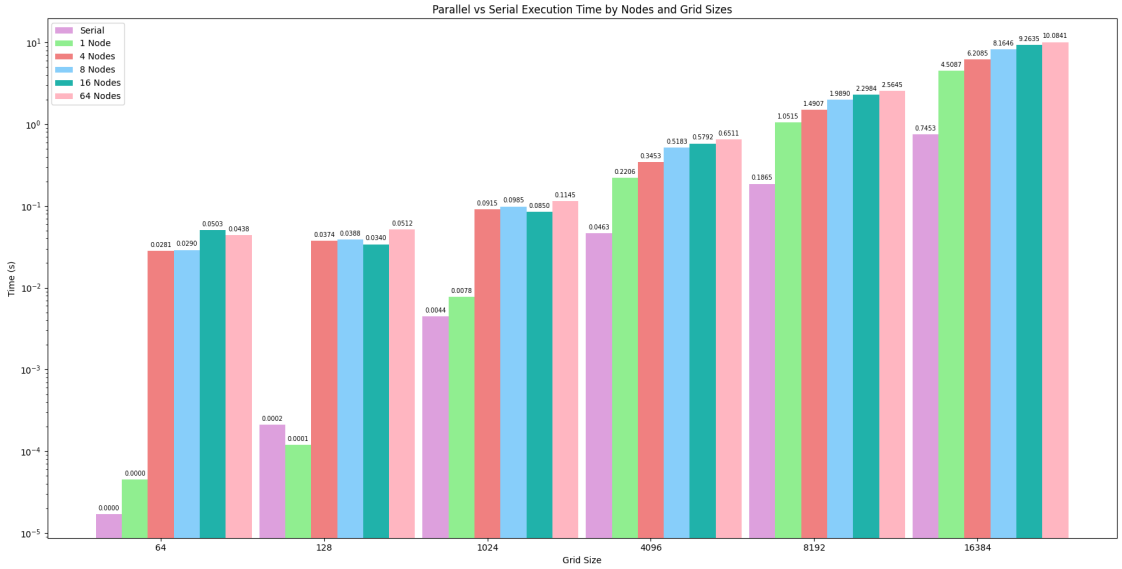
Exercise 3.2

Execution

For the execution of the parallel program of the exercise 3.2, after the compilation run the command:
`mpiexec -f ../machines -n <nodes> ./ask2/mat_vec <grid_size>` .

General Implementation Comments

A distributed version of the column-striped matrix-vector multiplication was implemented. The basic algorithm is to multiply the i-th column of the matrix by the i-th element of the vector, creating n new vectors, which are then summed together at the end to produce the resulting vector. For parallelizing this problem, we distributed the columns and the corresponding elements of the vector across the nodes. The vector elements are distributed with a simple `MPI_Scatter()`, but the columns, due to the flat representation of the matrix, were not as easy to send. Fortunately, MPI has taken care of such cases, so we defined a new data type `MPI_Type_vector()`, with the appropriate resizing to allow it to wrap around to the next column (We followed the instructions on page 12 of guide.). Therefore, each node receives an array with `n/comm_sz` columns and the corresponding elements of the vector. Locally, it multiplies one column by the corresponding element and adds the result to a local accumulator vector. Finally, the root process calls `MPI_Reduce()` to sum all the partial results from the nodes and produce the final result.
Running the program on the laboratory machines, we obtain the following results:



Grid Size	Algorithm	Nodes	Scatter	Calc	Reduce	Total Time
64	Serial	-	-	0.000017	-	0.00002
64	Parallel	1	0.000033	0.000012	0.000000	0.00005
64	Parallel	4	0.000045	0.015003	0.013006	0.02805
64	Parallel	8	0.000200	0.018983	0.009808	0.02899
64	Parallel	16	0.000296	0.029990	0.019991	0.05028
64	Parallel	64	0.001311	0.028504	0.013969	0.04378
128	Serial	-	-	0.000212	-	0.00021
128	Parallel	1	0.000068	0.000049	0.000002	0.00012
128	Parallel	4	0.000093	0.021912	0.015405	0.03741
128	Parallel	8	0.000500	0.028218	0.010061	0.03878
128	Parallel	16	0.000864	0.020995	0.012101	0.03396
128	Parallel	64	0.001046	0.025190	0.024971	0.05121
1,024	Serial	-	-	0.004442	-	0.00444
1,024	Parallel	1	0.004581	0.003176	0.000007	0.00776
1,024	Parallel	4	0.069122	0.011384	0.011019	0.09153
1,024	Parallel	8	0.062328	0.018111	0.018060	0.0985
1,024	Parallel	16	0.063495	0.015139	0.006332	0.08497
1,024	Parallel	64	0.076367	0.023570	0.014589	0.11453
4,096	Serial	-	-	0.046335	-	0.04634
4,096	Parallel	1	0.169681	0.050867	0.000008	0.22056
4,096	Parallel	4	0.314062	0.031138	0.000051	0.34525
4,096	Parallel	8	0.493417	0.023505	0.001331	0.51825
4,096	Parallel	16	0.566979	0.011632	0.000637	0.57925
4,096	Parallel	64	0.639614	0.009540	0.001968	0.65112
8,192	Serial	-	-	0.186517	-	0.18652
8,192	Parallel	1	0.848824	0.202626	0.000011	1.05146
8,192	Parallel	4	1.398207	0.092421	0.000082	1.49071
8,192	Parallel	8	1.916427	0.066652	0.005902	1.98898
8,192	Parallel	16	2.254256	0.035148	0.008949	2.29835
8,192	Parallel	64	2.531756	0.017724	0.015051	2.56453
16,384	Serial	-	-	0.745311	-	0.74531
16,384	Parallel	1	3.702011	0.806680	0.000012	4.5087
16,384	Parallel	4	5.852951	0.345502	0.010093	6.20855
16,384	Parallel	8	7.970814	0.192700	0.001116	8.16463
16,384	Parallel	16	9.162050	0.099634	0.001828	9.26351
16,384	Parallel	64	10.052782	0.027589	0.003712	10.08408

Conclusions

Initially, we used a serial version of the multiplication that traverses the matrix column-wise, with execution times shown in the table above. Observing the execution times, we noticed a rather interesting difference between the execution time of this serial program and that of the parallel one with a single node. We saw that the time for the 1-node version is significantly smaller compared to the serial version. One explanation for this is the way the data is stored and accessed in each algorithm. In the serial version, the matrix is traversed vertically through columns, so in each iteration, the next element is not adjacent in memory. In contrast, in the distributed form | even with just 1 node | we distribute the columns in such a way that each column is accessed as a contiguous array. Since memory accesses occur in consecutive locations, we get significantly better cache locality and, therefore, better performance, which becomes even more apparent as the matrix size increases. This large time difference initially led us to calculate efficiency values significantly

greater than one, which theoretically should not be possible.

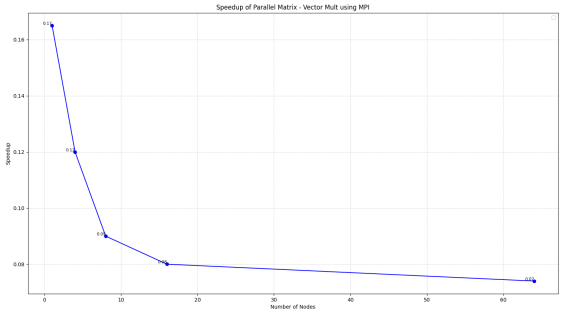
Grid Size	T.Serial (Col striped)	T.Serial (Row striped)
4096	0.391491	0.046335
8192	1.62599	0.186517
16384	6.95788	0.745311

However, by modifying the serial version to perform the multiplication row-wise, we observe in the table above that the execution time of this serial version improved significantly. This confirms our hypothesis and is now almost identical to the calculation time of the parallel 1-node version (as seen in the initial results table). Nevertheless, the delay caused by the vertical access pattern did not disappear in the parallel version. Looking at the time taken for the scatter operation, we see that this delay effectively shifted to the initial distribution of the columns. We notice that in all measurements, the largest portion of the execution time is devoted to the distribution of columns, a portion that seems to grow with the increase in the number of nodes. For N=16k, initially (for 4 nodes), the scatter takes 5.85 seconds and the computation 0.34 seconds, totaling 6.20 seconds. For 16 nodes, the scatter takes 10.05 seconds and the computation only 0.027 seconds.With the addition of more nodes, the pure computation time appears to decrease linearly, but the scatter time increases significantly. Therefore, parallelism introduces much more overhead than the performance gain it offers, making the tradeoff not worthwhile.

These observations are also confirmed by the speedup and efficiency metrics. The speedup for matrix dimension 16384 is as follows:

Speedup:

comm_sz	T.Serial	T.Parallel	Speedup
1	0.74531	4.5087	0.1653
4	0.74531	6.2085	0.12
8	0.74531	8.1646	0.091
16	0.74531	9.2635	0.08
64	0.74531	10.084	0.073



The speedup is well below one in all cases, which means that parallelization made the execution significantly slower than the serial version.

Efficiency:

comm_sz	64	128	1024	4096	8192	16384
1	0.377	1.766	0.572	0.210	0.177	0.165
4	0.0001	0.0014	0.0121	0.033	0.031	0.03
8	0.00007	0.00068	0.0056	0.011	0.011	0.01
16	0.00002	0.0004	0.0032	0.005	0.005	0.005
64	0.000006	0.00006	0.0006	0.0011	0.0011	0.0011

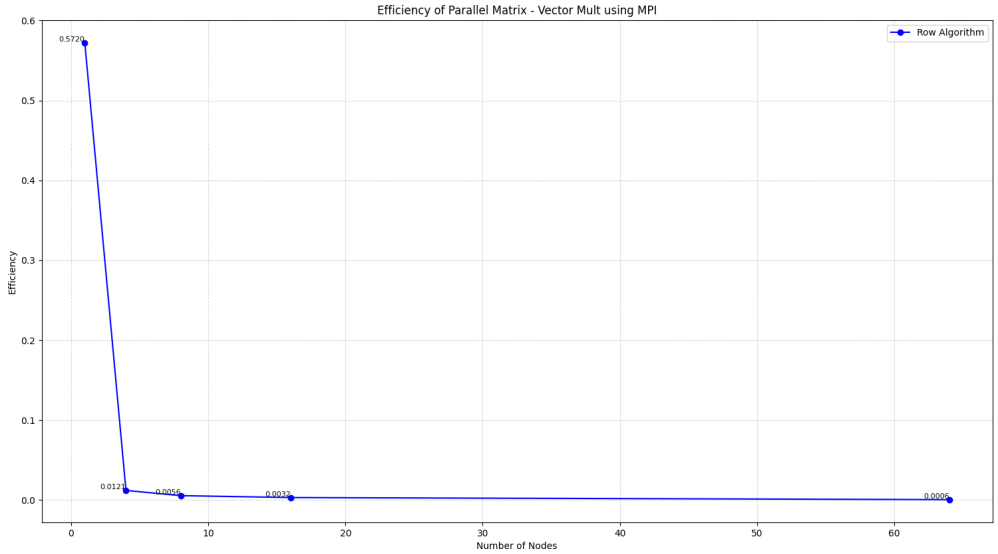


Figure 5: Efficiency N=1024

We did not expect good performance, and this is evident in the efficiency, which is close to zero in almost all cases | even for 1 node it's below half, since now it's being compared against the serial version that multiplies in a row-oriented manner and thus doesn't suffer from the vertical access pattern mentioned earlier. We observe that keeping the problem size fixed and increasing the number of nodes results in decreased efficiency. Additionally, keeping the number of nodes fixed and increasing the problem size, the efficiency remains relatively constant instead of increasing. Therefore, we can characterize the problem/implementation as **weakly scalable**, since allocating more resources does not lead to any performance improvement.