Report

Computing C = A×B & Transforming C to upper Δ

6th April 2021

OVERVIEW

Creating two matrices, A and B, each of size (N × N). Initialising the matrices to random floating point numbers.
Writing an **OpenMP and MPI** code for computing C = A×B and then transforming C into an upper triangular matrix. Reporting the times taken for the codes by varying the size of the problem from N = 100 . . . 10000. Varying the number for MPI process and OpenMP threads from 2 to 8 on a single desktop

OpenMP Parallelisation

Algorithm

- Allocate dynamic memory using malloc() with the size (N × N ×size of float) for A (mat1) , B (mat2) and C (mul).
- Initialize random floating numbers in each parallel thread with seed which is a combination of current time from the current thread number.
- Multiply matrices using three loops i.e O(n^3) and parallelised with **#pragma omp parallel shared (matrices) private (loop variables)** command. Thus, splitting for-loops independently amongst threads. Using ikj loop sequence over usual ijk because of caching advantages.
- Using row transformation technique to transform a matrix to upper triangle where three for loops are spawned. Outermost loop represents the number of selected columns which are transformed to zero. Penultimate loop iterates the rows and ultimate for-loop iterates the column and thus each element.

- Basic operation is like, $R_i \rightarrow R_i + \lambda \times R_j$, where i and j are as per row transformation for triangularisation
- Outmost loop can't be parallelised. So, dividing the penultimate loop of rows, thus sections of rows and corresponding column elements are solved on threads parallely using **#pragma omp parallel for private(row, column, λ)**.
- Display matrices and results for N<10 to check for logic correctness.

Observations

The results of time taken to compute C = A×B and transforming C to upper triangle matrix for OpenMP parallelisation are reported in the table below. The time reported is "real time". The actual real-time spent in running the process from start to finish as if it was measured by a human with a stopwatch by its definition.

OpenMP (Table-1)					
N	**Thread =1**	**Thread =2**	**Thread =4**	**Thread =6**	**Thread =8**
	Time (sec)	**Time (sec)**	**Time (sec)**	**Time (sec)**	**Time (sec)**
100	0.09938	0.11026	0.09982	0.09978	0.10936
200	0.15818	0.16956	0.17814	0.17434	0.20816
500	0.85174	0.76246	0.847	0.7875	0.87522
1000	5.8562	5.5835	4.562	3.374767	3.8674
2000	45.9952	35.67375	23.4076	17.6229	18.00035
5000	715.3215	447.5954	282.0079	198.9748	181.5451
10000	5718.499	1998.4527	1787.7983	1373.6564	1302.8688

Conclusions

- As the dimension of matrices increases, the time to compute increases as shown in the table-1 and figure 1.
- As the dimension of matrices increases, the speedup increases as shown in figure 2. Also, the parallelisation overheads can be observed for N<500 since there isn't significant speedup observed.
- Ikj looping is faster as can be compared from Table-1 and Table-3 (At the last of report).

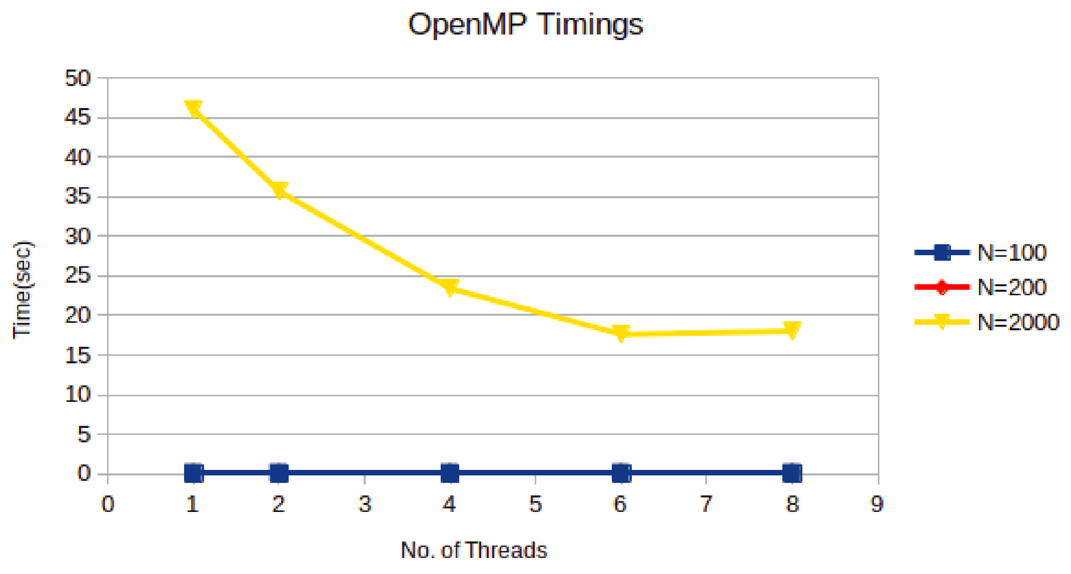

Figure 1

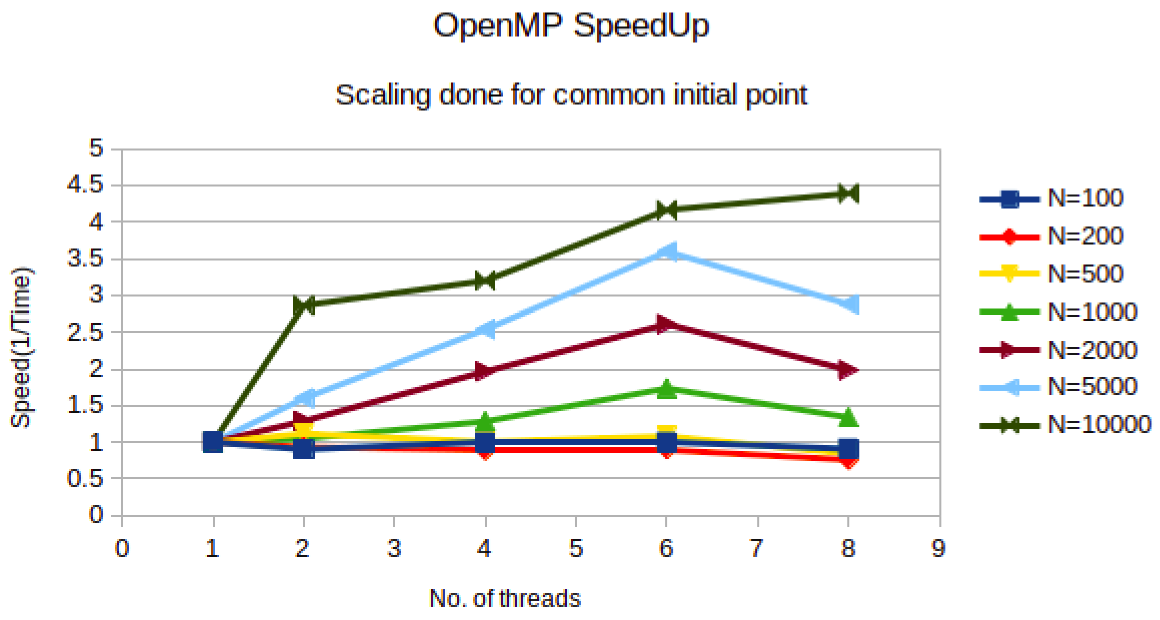

Figure 2

MPI Parallelisation

Algorithm

- Defining *stripsize* as *N/number of nodes* which corresponds to dividing matrices row wise into strips for parallel computation.
- Allocate dynamic memory using malloc() with the size (N × N × size of float) for A (mat1) , B (mat2) and C (mul) in the parent node.
- For rank = 0 i.e. parent node will have N ×N i.e. full A and C memory allocations while other processes will have N/nodes rows and N columns of matrices A and C. All processes will have N rows and N columns of B i.e. full B will be shared.
- All other nodes will have *stripsize*× N memory allocation except for parent node i.e. rank = 0 and last node whose size is decided by remaining rows of matrix (eg if N=5, nodes =2, then division is like 2 rows for node with rank = 0 and 3 rows for node with rank =1).
- Allocate memory for the iteratively shared row which will make column entries zero for triangularisation.
- At parent node i.e. rank =0 initialise A and B matrices with floating random numbers in range 0 to 1.
- Now send the strips with *stripsize* as per step-1 of matrix A to other nodes while broadcast matrix B to others. However, other nodes will receive strips of A.
- Multiply matrices using three loops i.e O(n^3) at all the nodes parallely. Thus, splitting for-loops independently amongst nodes/processors. Using ikj loop sequence over usual ijk because of caching advantages.
- Parent receives C i.e. multiplication results from all the nodes while other nodes send their parts (*stripsize*) of computed results to parent.
- Using row transformation technique to transform a matrix to upper triangle where three for loops are spawned. Outermost loop represents the number of selected columns which are transformed to zero. Penultimate loop iterates the rows and ultimate for-loop iterates the column and thus each element.
- Outmost loop can't be parallelised. So, dividing the penultimate loop of rows. Thus sections of rows and corresponding column elements are solved on nodes parallely.
- Inside the outermost loop, if node is parent then send the row (called *share_row* in code) which is currently responsible to make corresponding columns zero at that iteration to all the other nodes.
- Perform the operation like, $\mathbf{R}_i \rightarrow \mathbf{R}_i + \boldsymbol{\lambda} \times \mathbf{R}_j$, where j is the *shared_row* are as per row transformation for triangularisation at that node on matrix C (parent) or strips of C(other nodes).

- Parent receives C i.e. triangularisation results from all the nodes while other nodes send their parts (*stripsize*) of computed results to parent as partial result C.
- Repeat the above two steps N-1 times to finally receive the triangularisation matrix C.
- Display matrices and results for N<10 to check for logic correctness.

Observation

The time taken to compute with MPI Parallelisation is reported in the table below.

MPI (Table-2)					
N	Node =1	Nodes =2	Nodes =4	Nodes =6	Nodes =8
	Time (sec)	Time (sec)	Time (sec)	Time (sec)	Time (sec)
100	0.9342	0.9472	0.94178	0.98472	0.9858
200	1.01936	1.00222	0.9789	1.0104	1.00942
500	1.7036	1.40856	1.24472	1.16442	1.2093
1000	6.9214	4.4523	3.1207	2.60253	2.43526
2000	47.691	29.3142	19.17315	15.01925	13.7007
5000	728.8851	441.8399	275.7005	228.9042	202.9322
10000	5839.5993	3516.7059	2183.3539	1786.2638	1585.7281

Conclusions

- As the dimension of matrices increases, the time to compute increases as shown in the table and figure 3.
- As the dimension of matrices increases, the speedup increases as shown in figure 4. Also, the parallelisation overheads can be observed for N<500 since there isn't significant speedup observed.
- From Table-1 &2 it can be observed that timings for MPI and OpenMP are almost similar. However, OpenMP parallelisation for the given problem statement and solution developed is little better than MPI because of too much data/message passing between nodes in triangularisation algorithm.
- Ikj looping is faster as can be compared from Table-2 and Table-4(At the last of report).

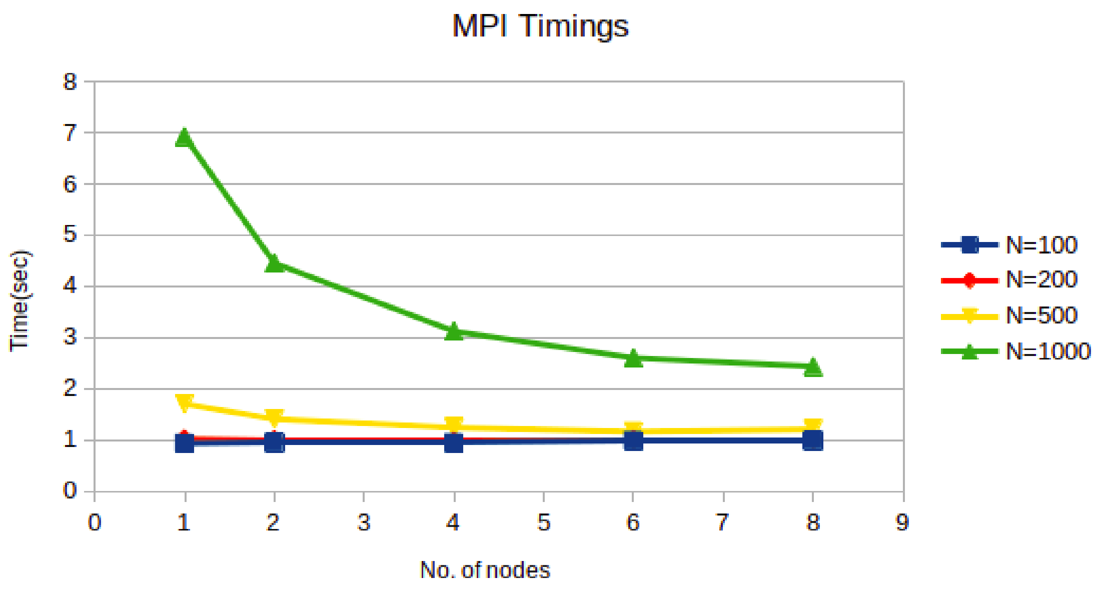

Figure 3

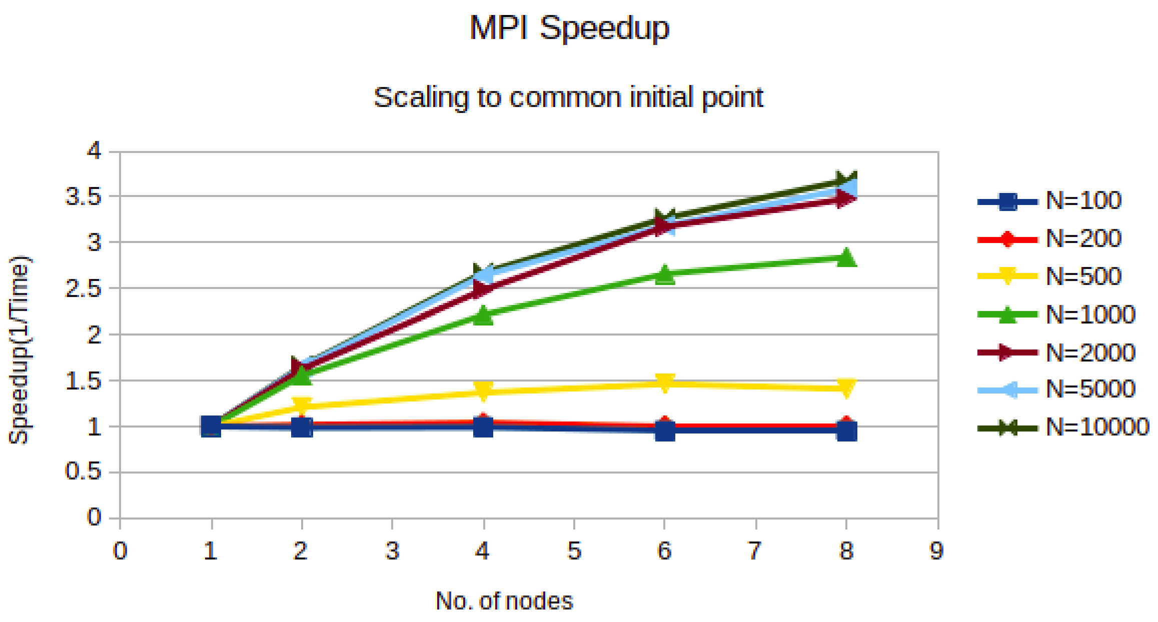

Figure 4

CPU Info and running attached code

CPU Info:

No. of processors : 16

vendor_id : GenuineIntel

model name : Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz

OS: Ubuntu 20 & using Open MPI and openMP

N--> Dimension of matrices

Simply run $ sudo ./HW2openMP.sh for logic correctness and timing analysis

------------------------------------or---

To run code for OpenMP
Set no. of threads: $ export OMP_NUM_THREADS=<No. of threads>
Compile: $ gcc -fopenmp HW2_openMP.c -o openMP.out
Output: $ time ./openMP.out <N>
Example:

```
$ export OMP_NUM_THREADS=4
$ gcc -fopenmp HW2_openMP.c -o openMP.out
$ time ./openMP.out 2000
```

Simply run $ sudo ./HW2MPI.sh for logic correctness and timing analysis

-----------------------------------or--

To run code for MPI
Compile: $ mpicc HW2_MPI.c -o MPI.out
Output: $ time mpirun -np <nodes> ./MPI.out <N>
Example:

```
$ mpicc HW2_MPI.c -o MPI.out
$ time mpirun -np 8 ./MPI.out 1000
```

For ijk looping in Multiplication

OpenMP (Table-3)					
N	Thread =1	Thread =2	Thread =4	Thread =6	Thread =8
	Time (sec)	Time (sec)	Time (sec)	Time (sec)	Time (sec)
100	0.017	0.023	0.031	0.032	0.041
200	0.076	0.156	0.105	0.111	0.134
500	0.905	1.114	0.663	0.675	0.822
1000	7.226	6.275	3.618	3.259	3.725
2000	73.049	47.745	25.512	23.032	21.385
5000	1740.732	1014.201	506.71	371.997	327.154

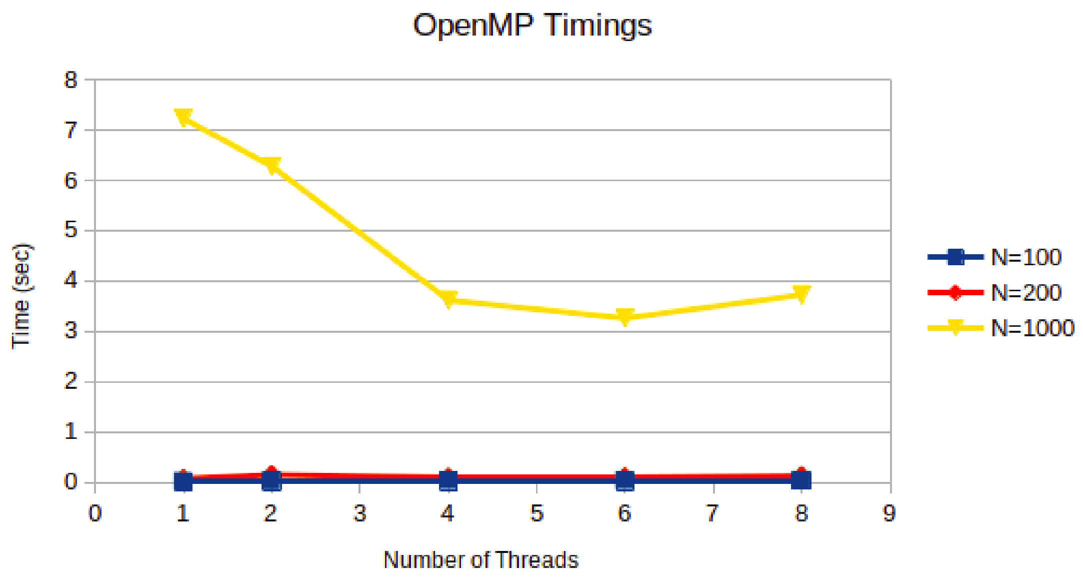

Figure 5

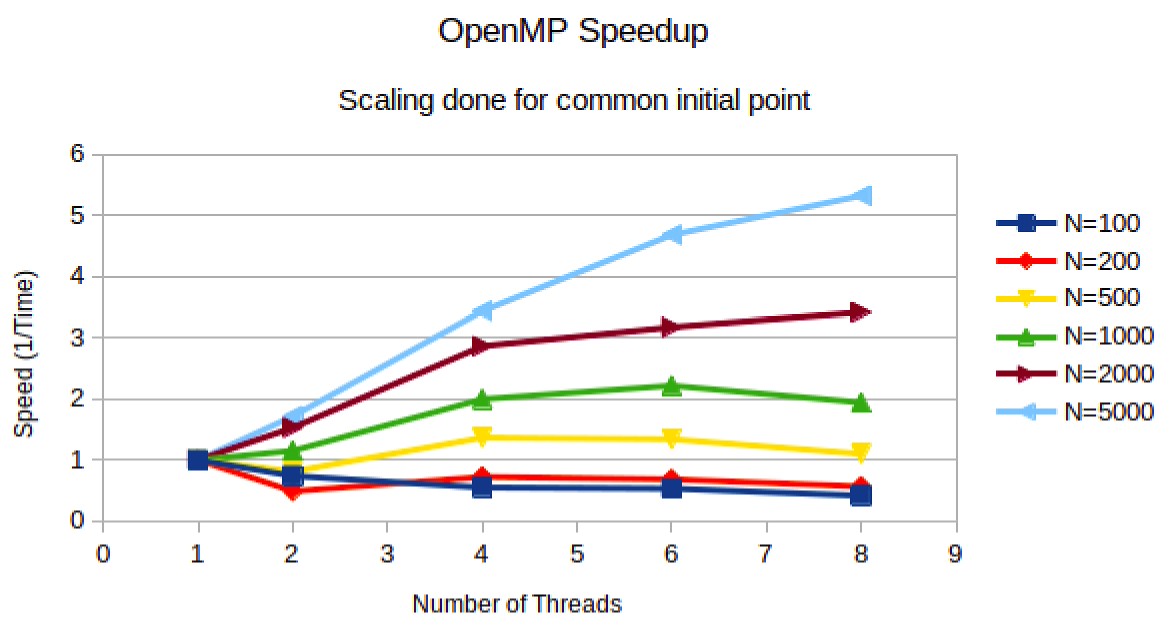

Figure 6

For ijk looping in Multiplication

The time taken to compute with MPI Parallelisation is reported in the table below.

MPI (Table-4)					
N	Node =1	Nodes =2	Nodes =4	Nodes =6	Nodes =8
	Time (sec)	Time (sec)	Time (sec)	Time (sec)	Time (sec)
100	0.344	0.345	0.354	0.361	0.372
200	0.404	0.384	0.372	0.383	0.393
500	1.322	0.951	0.729	0.646	0.631
1000	7.938	4.682	3.094	2.591	2.294
2000	80.953	47.432	28.025	21.652	18.01
5000	1867.566	1020.782	616.483	453.533	360.226

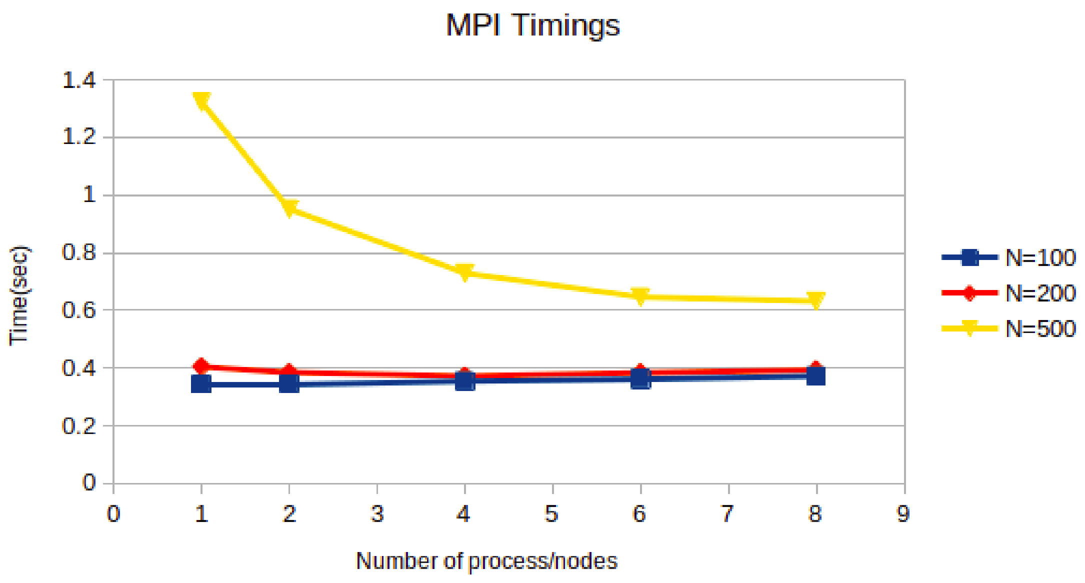

Figure 7

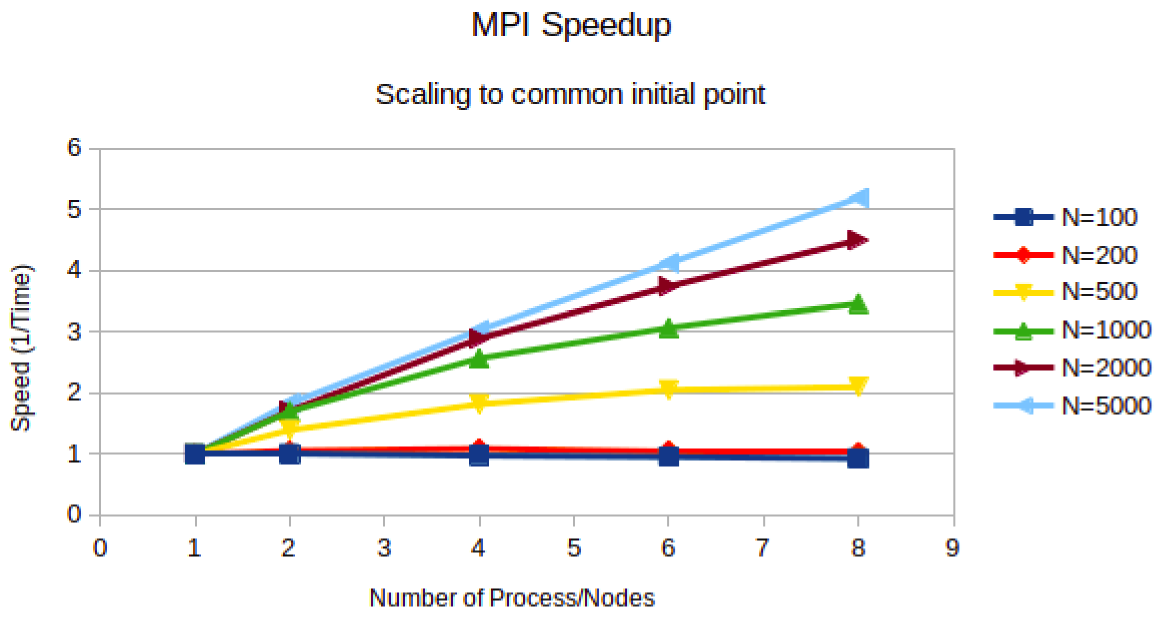

Figure 8