High Performance Scientific Computing
ME 766 : Homework 2

183079026 - Sandesh Goyal

March 20, 2021

1 Platform

Model Name: Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
CPU(s): 40
RAM: 32 GB
OS: CentOS Linux 7 (Core), 64-bit
GCC Version: 4.8.5

2 Matrix Multiplication and Upper Triangular Matrix transformation

Matrix Multiplication is performed using standard procedure of **dot-product** of one row of Matrix A with one column of Matrix B. In order to be **Memory storage/fetch efficient**, since the column of Matrix B is fetched every time and not row, Matrix B is initialized in column major wise so that its column data is stored in row for better efficiency during memory fetch. All this is done because in C language, for 2D array, **memory is addressed row wise and not column**.

Upper Triangular Matrix Transformation**(UTMT)** is performed using **Gauss Elimination method with row operations**.

Memory allocation is static for the Matrices and therefore the **matrix size(N) has to edited manually before compilation**. All the codes are compiled using **O3 flag** for compiler optimization, this helped in **saving the run-time**.

Correctness and Repeatability of the implementation is checked and verified for N = 5 for Integer values. Implementation uses floating point numbers for random initialization of the two matrices A and B. Result can be printed/displayed on the terminal directly by calling/uncommenting the **print_mat()** function. Slight discrepancy or small error(non zero value in lower triangle) can be expected in the final result because of the **truncation error in the storage and operation of floating point numbers**.

All the **timing analysis carried out and tabulated below is for the whole code** and not any section of the code. Real time of the whole code is reported.

2.1 OpenMP Implementation

For OpenMP based implementation, for both the operations, Matrix Multiplication and UTMT, parallelism is dome using **OpenMP pragma** for **for loop**, taking care of the **shared and private variables**. Matrix initialization for A and B with random floating point numbers is done serially.

For **Matrix multiplication, parallelism is done at the top-level** (outer most loop) because there is **no data dependency**. For **UTMT, parallelism is done at intermediate level**(inner-loop) so as to **avoid the data dependency**. Timing data for single thread execution can taken as of serial implementation.

Table 1: Timing Analysis for OpenMP

MATRIX SIZE (N) = 100					
Iterations/Threads	**1**	**2**	**4**	**6**	**8**
1	0.006	0.004	0.003	0.004	0.003
2	0.003	0.003	0.003	0.003	0.005
3	0.003	0.004	0.003	0.003	0.004
4	0.003	0.004	0.003	0.003	0.004
5	0.003	0.003	0.003	0.003	0.004
Average Run-Time(s)	**0.0036**	**0.0036**	**0.0030**	**0.0032**	**0.0040**
MATRIX SIZE (N) = 1000					
Iterations/Threads	**1**	**2**	**4**	**6**	**8**
1	1.312	0.776	0.473	0.320	0.260
2	1.005	0.782	0.430	0.323	0.262
3	1.019	0.774	0.432	0.324	0.262
4	1.004	0.574	0.469	0.325	0.261
5	1.019	0.541	0.421	0.318	0.261
Average Run-Time(s)	**1.0718**	**0.6894**	**0.4450**	**0.322**	**0.2612**
MATRIX SIZE (N) = 5000					
Iterations/Threads	**1**	**2**	**4**	**6**	**8**
1	144.473	72.097	37.171	27.500	22.544
2	144.556	72.740	37.644	27.406	22.292
3	144.167	71.640	37.825	27.571	22.532
4	144.184	71.788	37.371	28.077	22.358
5	144.803	72.009	37.489	27.993	22.642
Average Run-Time(s)	**144.4366**	**72.0548**	**37.5000**	**27.7094**	**22.4736**
MATRIX SIZE (N) = 10000					
Iterations/Threads	**1**	**2**	**4**	**6**	**8**
1	1163.286	589.766	323.114	238.194	195.779
2	1160.922	593.603	321.296	239.197	195.186
3	1163.632	592.206	318.684	239.673	194.971
4	1161.056	595.167	316.127	232.792	200.937
5	1163.288	594.375	316.093	236.757	193.666
Average Run-Time(s)	**1162.4368**	**593.0234**	**319.0628**	**237.3226**	**196.1078**

Table 2: Speedup using OpenMP

Threads	**Speedup (N = 100)**	**Speedup (N = 1000)**	**Speedup (N = 5000)**	**Speedup (N = 10000)**
1	1.00	1.00	1.00	1.00
2	1.00	1.55	2.00	1.96
4	1.20	2.41	3.85	3.64
6	1.13	3.35	5.21	4.90
8	0.90	4.10	6.43	5.93

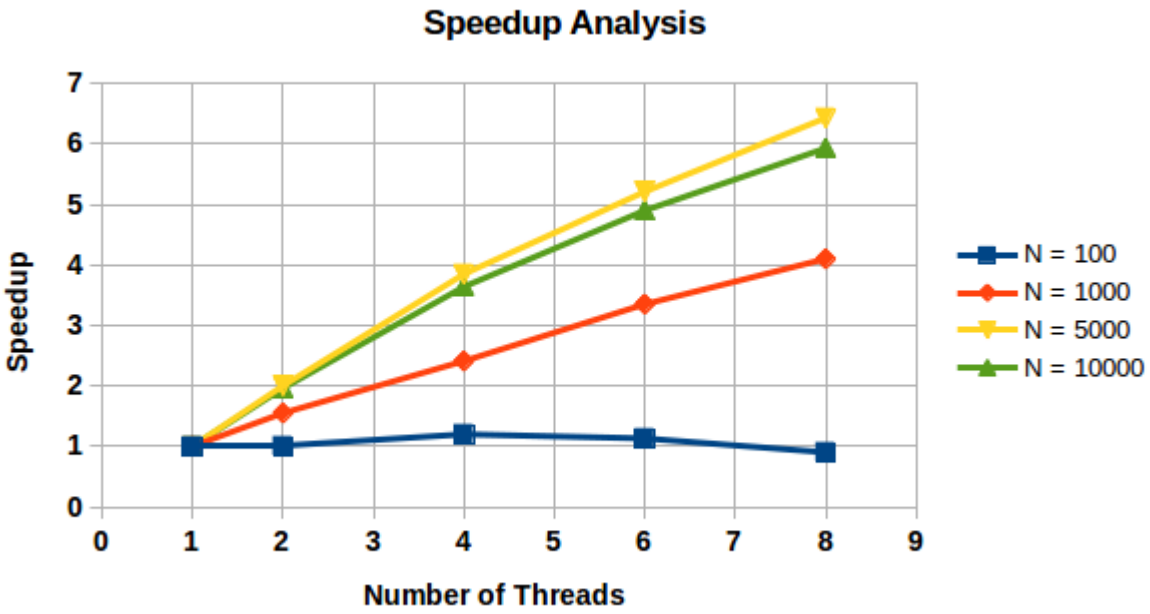

Figure 1: Number of Threads vs Speedup for OpenMP

OpenMP results: From the above data for timing analysis, it is very clear that parallelism using OpenMP is **not very effective for less number of operations or small matrix size because of more memory store/fetch and scheduling overheads**. But as the number of operations increases because of **increase in the matrix size, effect of parallelism can be clearly observed**. The speedup is as expected but not exactly equal to the number of threads because of the **memory and scheduling overheads** and serial matrix initialization with random numbers and also because of the other things happening the background.

2.2 MPI Implementation

For MPI based implementation, **Process 0 acts as ROOT**, it allots the work to the other processes, send data to them for processing/operations and then receives back after the operation is completed. Therefore **minimum required number of process is 2 and not 1**. If the number of process set is 5, then it means ROOT + 4 processes. **Matrix initialization for A and B is done by ROOT**. For task allocation to different number of processes, ROOT calculates the set size dynamically, therefore only number of processes are supposed to specified at the run-time and no other details.

For Matrix Multiplication, there is no data dependency, particular set of rows of Matrix A is divided by ROOT among the processes and Matrix B is broadcasted to all the processes, but **for UTMT, there is data dependency in every iteration** of making next column zero, therefore **next iteration is started by ROOT only after all the other processes complete the current iteration and update the data back to ROOT**.

There are **two versions** of the implementation:

Completely Parallel: In this, **both Matrix Multiplication and UTMT is parallelized** using process. **Results/speedup of only Matrix Multiplication is quite reasonable for multiple processes** but for UTMT, the speedup is less than 1, which means that the execution time increases instead of decreasing on increasing the number of processes. This is **because of very huge memory/data transfer overhead during UTMT**.

Partially Parallel: In this, **only Matrix Multiplication is parallelized** and **UTMT is serially computed by ROOT**. **Results/speedup is** not very great but **far better that the other version**. After Matrix Multiplication, all the processes except ROOT, stops and then ROOT performs UTMT

2.2.1 Completely Parallel

Table 3: Timing Analysis for MPI

MATRIX SIZE (N) = 1000					
Iterations/Process	**1**	**2**	**4**	**6**	**8**
1	2.371	2.886	2.931	3.074	3.861
2	2.138	2.747	2.938	3.030	3.886
3	2.157	2.707	2.970	3.048	3.748
4	2.138	2.759	2.941	2.976	3.850
5	2.132	2.762	2.972	3.019	3.902
Average Run-Time(s)	**2.1872**	**2.7722**	**2.9504**	**3.0294**	**3.8494**
MATRIX SIZE (N) = 5000					
Iterations/Process	**1**	**2**	**4**	**6**	**8**
1	311.304	342.226	398.318	445.528	484.823
2	316.527	339.745	371.293	424.394	485.038
3	317.115	338.348	390.969	435.272	498.816
4	310.587	339.610	385.665	444.987	470.054
5	311.085	339.641	378.763	435.477	481.000
Average Run-Time(s)	**313.3236**	**339.9140**	**385.0016**	**437.1316**	**483.9462**

Table 4: Speedup using MPI

Process	**Speedup (N = 1000)**	**Speedup (N = 5000)**
1	1.00	1.00
2	0.79	0.92
4	0.84	0.81
6	0.72	0.72
8	0.57	0.65

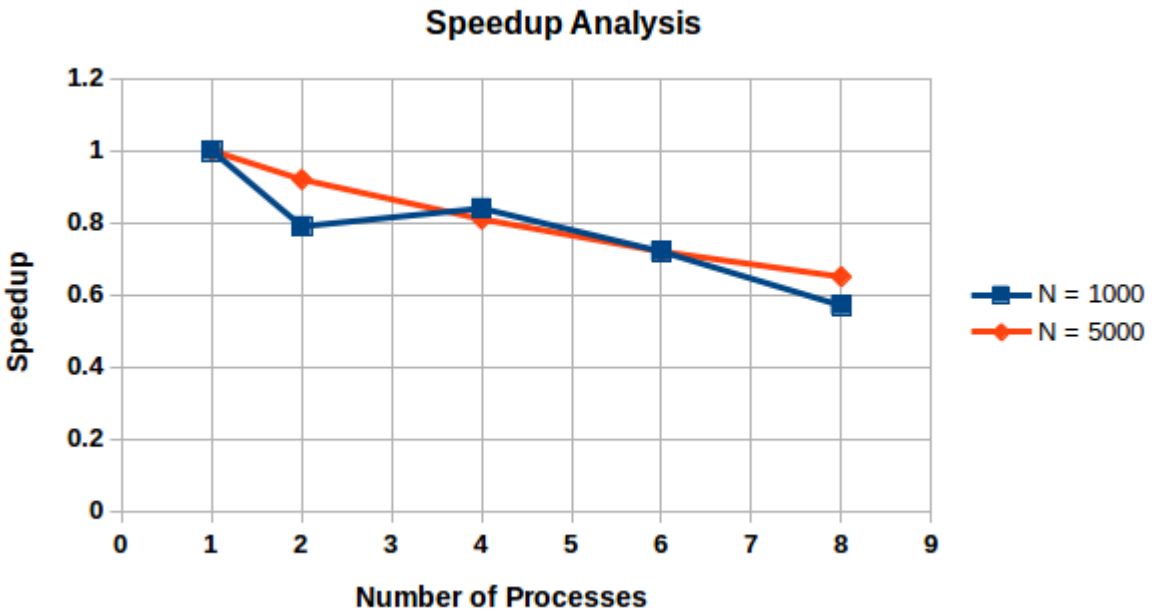

Figure 2: Number of Processes vs Speedup for MPI

2.2.2 Partially Parallel

For N = 100 and N = 1000, there is relatively large difference in Process 0(ROOT) time and real time because of system overhead, therefore, only for N = 100 and 1000, ROOT(terminates at the end) time is reported for rest, real time is used.

Table 5: Timing Analysis for MPI

MATRIX SIZE (N) = 100					
Iterations/Process	**1**	**2**	**4**	**6**	**8**
1	0.003	0.003	0.003	0.003	0.003
2	0.003	0.003	0.003	0.003	0.003
3	0.003	0.003	0.003	0.003	0.003
4	0.003	0.003	0.003	0.003	0.003
5	0.003	0.003	0.003	0.003	0.003
Average Run-Time(s)	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**
MATRIX SIZE (N) = 1000					
Iterations/Process	**1**	**2**	**4**	**6**	**8**
1	1.224	0.806	0.545	0.422	0.386
2	1.222	0.823	0.559	0.452	0.379
3	1.210	0.809	0.504	0.412	0.403
4	1.223	0.831	0.533	0.442	0.367
5	1.067	0.802	0.536	0.431	0.386
Average Run-Time(s)	**1.1892**	**0.8142**	**0.5354**	**0.4318**	**0.3842**
MATRIX SIZE (N) = 5000					
Iterations/Process	**1**	**2**	**4**	**6**	**8**
1	147.610	85.818	56.119	46.817	42.250
2	147.803	86.158	56.350	47.118	42.204
3	147.716	85.619	56.651	46.851	42.571
4	148.366	85.643	56.474	46.877	42.354
5	148.009	85.232	56.320	46.862	42.241
Average Run-Time(s)	**147.9008**	**85.6940**	**56.3828**	**46.9050**	**42.3240**
MATRIX SIZE (N) = 10000					
Iterations/Process	**1**	**2**	**4**	**6**	**8**
1	1184.252	690.212	451.777	376.179	337.646
2	1190.382	687.730	451.870	375.808	338.260
3	1190.284	687.567	451.681	374.902	338.628
4	1189.196	688.271	453.681	376.415	338.688
5	1189.235	682.094	452.066	375.008	337.693
Average Run-Time(s)	**1188.6698**	**687.1748**	**452.215**	**375.6624**	**338.1830**

Table 6: Speedup using MPI

Process	**Speedup (N = 100)**	**Speedup (N = 1000)**	**Speedup (N = 5000)**	**Speedup (N = 10000)**
1	1.00	1.00	1.00	1.00
2	1.00	1.46	1.73	1.73
4	1.00	2.22	2.62	2.63
6	1.00	2.75	3.15	3.16
8	1.00	3.10	3.50	3.51

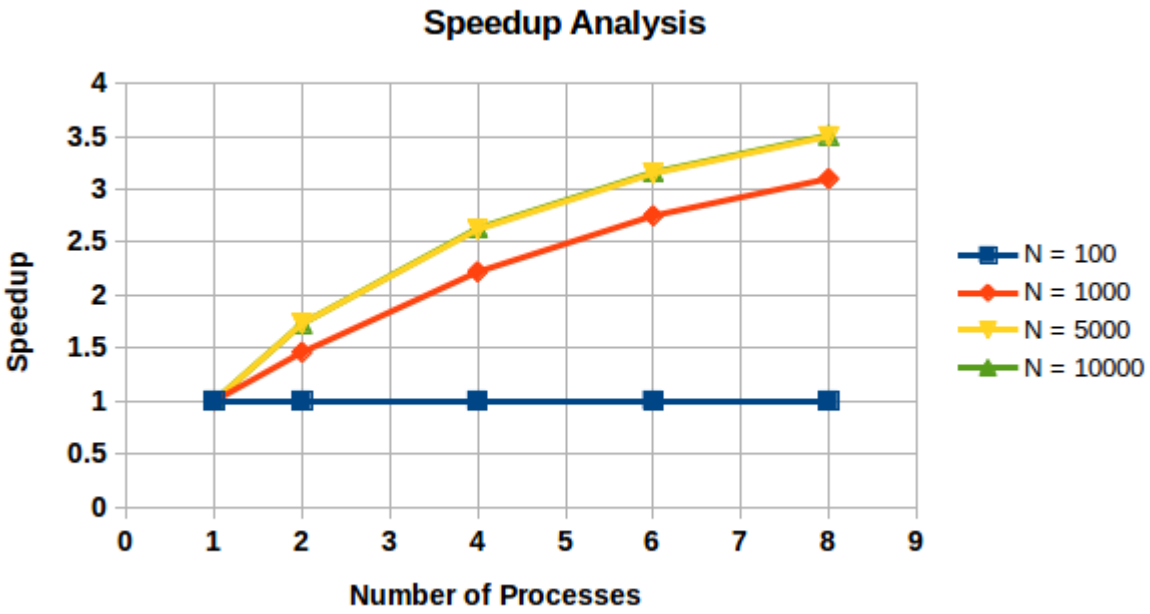

Figure 3: Number of Processes vs Speedup for MPI (N=5000 and N=10000 are overlapped)

3 Submission

- Sandesh_183079026_HW2.zip
  - mpi
    - completely_parallel
      - mpi_combine.c
      - mpi_combine.sh
      - N_1000_mpi.txt
      - N_5000_mpi.txt
    - partially_parallel
      - mpi_combine_s.c
      - mpi_combine_s.sh
      - N_100_mpi_s.txt
      - N_1000_mpi_s.txt
      - N_5000_mpi_s.txt
      - N_10000_mpi_s.txt
  - openmp
    - omp_combine.c
    - omp_combine.sh
    - N_100_omp.txt
    - N_1000_omp.txt
    - N_5000_omp.txt
    - N_10000_omp.txt
  - ME766_HW2_REPORT.pdf

The **script(.sh)** file can **compile and run the c code**. Value of **N(matrix size) has to be modified/set manually** before running the script. **".txt" files contain the timing data/values** obtained by running the script file on the platform mentioned in the first section.