Tezan Sahu
170100035
tezansahu@iitb.ac.in

Homework 2
ME766: High Performance Scientific Computing

Due Date: 07/04/21

Problem Statement

Create two matrices, A and B, each of size $(N \times N)$. Initialise the matrices to random floating point numbers. Write parallel code for computing $C = AB$ and then transforming C into an upper triangular matrix using the following paradigms:

1. *Shared Memory Processing*, using **OpenMP** Code
2. *Message Passing*, using **MPI** Code (using a single desktop only)

Vary the size of the problem from $N = 100...10000$ Analyse the running time of the codes by varying the number of threads or number of MPI processes from 2 to 8.

1 OpenMP Version

Implementation Details

- The C++ code has been written in the `mat_mul_upper_openmp.cpp` file
- Both `N` & `num_threads` (No. of OpenMP threads to be used) are provided as input to the script. The default values are `N=1000` & `num_threads=1`.
- The pseudorandom number generator is seeded with 0 & used to populate matrices A & B
- The *matrix multiplication* & *Gaussian elimination* (for converting to upper triangular matrix) are implemented in separate functions & are performed using $O(N^3)$ algorithms (3 nested loops with indices `i`, `j` & `k`) such that the iterations indexed with `i` are parallelized across the threads, keeping the corresponding `j` & `k` private to each thread.
- The running times for multiplication & Gaussian elimination were noted separately using `chrono::system_clock`
- To prevent segmentation fault to occur at runtime due to insufficient stack size (while dealing with larger matrices), we use `ulimit -s unlimited` to increase the stack size before running the code
- The `timing_study.sh` script compiles `mat_mul_upper_openmp.cpp` using `g++` & performs the runtime analysis for `N` $\in \{100, 500, 1000, 2000, 5000\}$, each with `num_threads` $\in \{1, 2, 4, 6, 8\}$

  Note: *Since the algorithm is $O(N^3)$, and the running time for `N=1000` & `num_threads=1` was $\sim 7.5\ s$, the expected running time for `N=10000` was $>\sim 2\ hrs$. Thus, the analysis is limited only to `N=5000`*

Results & Discussion

Matrix Multiplication

	Time Taken (s) with various No. of Threads				
N	**1**	**2**	**4**	**6**	**8**
100	0.004	0.002	0.003	0.003	0.003
500	0.489	0.313	0.187	0.154	0.159
1000	4.964	3.169	2.125	1.797	2.032
2000	62.816	36.62	23.085	20.453	19.238
5000	1091.127	709.056	447.835	414.653	358.796

- The running time varies approximately $O(N^3)$
- For smaller values of N, increasing the number of threads speeds up the execution to some extent, but later causes a slight rise in the timing due to overheads of thread maintenance
- For larger values of N, increasing the number of threads necessarily reduces the running time, however, the speedup is not linear

Gaussian Elimination (Conversion to Upper Triangular Matrix)

	Time Taken (s) with various No. of Threads				
N	**1**	**2**	**4**	**6**	**8**
100	0.002	0.002	0.002	0.003	0.001
500	0.287	0.256	0.193	0.112	0.109
1000	2.574	2.15	1.23	0.931	0.853
2000	19.782	15.946	10.447	8.322	7.384
5000	315.414	244.193	178.637	141.089	125.348

- The running time varies approximately $O(N^3)$
- For all values of N, increasing the number of threads more or less reduces the execution time, however, the speedup is not linear

Overall Execution

	Time Taken (s) with various No. of Threads				
N	**1**	**2**	**4**	**6**	**8**
100	0.006	0.004	0.005	0.006	0.004
500	0.776	0.569	0.38	0.266	0.268
1000	7.538	5.319	3.355	2.728	2.885
2000	82.598	52.566	33.532	28.775	26.622
5000	1406.541	953.249	626.472	555.742	484.144

2 MPI Version

Implementation Details

- The C++ code has been written in the `mat_mul_upper_mpi.cpp` file
- Both `N` & `num_pes` (No. of MPI processes to be used) are specified at runtime. The default values are `N=1000` & `num_pes=1`.
- The pseudorandom number generator is seeded with 0 & used to populate matrices A & B
- For *matrix multiplication*:
  - The process with ID 0 splits matrix A into `num_pes` chunks, each with $\frac{N}{num\_pes}$ rows & N columns, & sends the respective A_{chunk}, full matrix B & the `offset` (to indicate the start of the chunk in the original matrix A) to all processes.
  - The last chunk takes care of rows if N is not divisible by `num_pes`
  - Each process receives the matrices from process 0, computes $C_{chunk} = A_{chunk}B$ & sends back C_{chunk} back to process 0.
  - Process 0 gathers these C_{chunk} received & constructs C appropriately.
- For *Gaussian elimination*:
  - In each iteration `i`, the pivot row of C, i.e., `C[i]` is broadcast to all processes
  - All the rows `j` below the i^{th} row are designated to the various processes & process 0 sends these rows to the designated processes
  - The Gaussian Elimination process is implemented between rows `i` & `j` by every process for all the rows that it receives from process 0
  - After computing these values, the modified rows are sent back to process 0 to reconstruct C post each iteration.
  - The pivot row `i` is incremented & the process continues until row `N-1` is reached
- The running times for multiplication & Gaussian elimination were noted separately using `chrono::system_clock`
- To prevent segmentation fault to occur at runtime due to insufficient stack size (while dealing with larger matrices), we use `ulimit -s unlimited` to increase the stack size before running the code
- The `timing_study.sh` script compiles `mat_mul_upper_mpi.cpp` using `mpic++` & performs the runtime analysis for `N` $\in \{100, 500, 1000, 2000\}$, each with `num_pes` $\in \{1, 2, 4, 6, 8\}$

  Note: *Since the algorithm is $O(N^3)$, and the running time for `N=1000` & `num_pes=1` was ~ 5 s, the expected running time for `N=10000` was ~ 1.5 hrs. Thus, the analysis is limited only to `N=5000`*

Results & Discussion

Matrix Multiplication

	Time Taken (s) with various No. of Processes				
N	**1**	**2**	**4**	**6**	**8**
100	0.003	0.002	0.002	0.001	0.001
500	0.456	0.25	0.206	0.156	0.146
1000	3.886	3.091	2.004	2.18	1.734
2000	57.567	34.244	22.229	19.516	19.7
5000	1207.127	736.837	522.730	449.497	509.166

- The running time varies approximately $O(N^3)$
- Number of messages passed is $O(num\_pes)$ [does not depend on N]
- For all values of N, increasing the number of threads more or less reduces the running time, however, the speedup is not linear
- For larger values of N, we observe stagnation or slight increase in execution time for `num_pes` ≥ 6

Gaussian Elimination (Conversion to Upper Triangular Matrix)

	Time Taken (s) with various No. of Processes				
N	**1**	**2**	**4**	**6**	**8**
100	0.001	0.001	0.002	0.003	0.003
500	0.111	0.124	0.132	0.173	0.174
1000	0.877	1.163	1.638	1.781	1.652
2000	7.556	11.303	16.037	17.962	19.491
5000	125.151	233.408	319.996	378.527	548.971

- The running time varies approximately $O(N^3)$
- Contrary to expectations, for all values of N, increasing the number of threads *increases* the execution time, which is possibly due to large number of messages being passed between processes for this algorithm [$O(N^2)$ messages]

Overall Execution

	Time Taken (s) with various No. of Processes				
N	**1**	**2**	**4**	**6**	**8**
100	0.004	0.003	0.004	0.004	0.004
500	0.567	0.374	0.338	0.329	0.32
1000	4.763	4.254	3.642	3.961	3.386
2000	65.123	45.547	38.266	37.478	39.191
5000	1332.278	970.245	842.727	828.024	1058.213

Since the decrease in execution time for the matrix multiplication is more significant than the increase in execution time for Gaussian elimination on increasing the number of processes, we actually get some speedup on increasing the number of MPI processes. However, for larger values of N like 2000 or 5000, we do notice that increasing `num_pes` to 6 or 8 actually starts slowing down the execution because message passing in Gaussian elimination starts taking much longer.

Machine Configuration & OS

Operating System	Ubuntu 18.04 *[Using Window Subsystem for Linux (WSL2)]*
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	8
Thread(s) per core	2
Core(s) per socket	4
Socket(s)	1
Model name	Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
CPU MHz	2304.006
g++ Version	7.5.0
OpenMP Version	4.5
MPI Version	3.3a2