CS 591 A1 Parallel Computing and Programming

Assignment 1 --- Matrix Multiplication

Members of group

Xueyan Xia	U 82450191	xueyanx@bu.edu
Ziqi Tan	U 88387934	ziqi1756@bu.edu

Requirements

1. Implement parallel matrix multiplication of square matrices of 2^{2d} random integers, using 2^k processes (i.e.., both matrix size and number of processes should be powers of 2 — d is the log2 of the number or rows/columns).
2. Compute the theoretical speedup with 1, 2, 4, and 8 processes, respectively (k = 0, 1, 2, and 3).
3. Test your program with various matrix sizes and numbers of workers and measure the actual running times.
4. Try to find the best performance (even if it is slower than the sequential version).
5. Compute speedup and efficiency for the best combination of matrix size and number of workers. Repeat this exercise for both threads (at the user level) and processes (at the kernel level).
6. Code must have complete docstrings, explaining the behavior of functions and the respective meanings of function parameters and return values. You must also use type hints on all symbols that allow them.

Details

1. Set your random seed to 1 so as to have a reproducible sequence of pseudo-random integers.
2. Your principal function should take k as one of its parameters.
3. You will need to work out some iteration logic to "chunk" your matrix appropriately into 2, 4, or 8 equal segments. (Look for tips in class.)
4. You will need to use both the threading and multiprocessing Python libraries for this exercise. Make separate versions, one using each of these libraries.
5. Use shared memory. For the process-based version you will find the multiprocessing. Array class useful for this purpose.
6. Use any Python construct for creating and destroying the threads and processes. You will likely find the lower-level Thread and Process classes the most useful for this assignment.
7. Test your program out on a CS Linux machine. If you do not yet have an account on one, please be sure to acquire one by registering here.

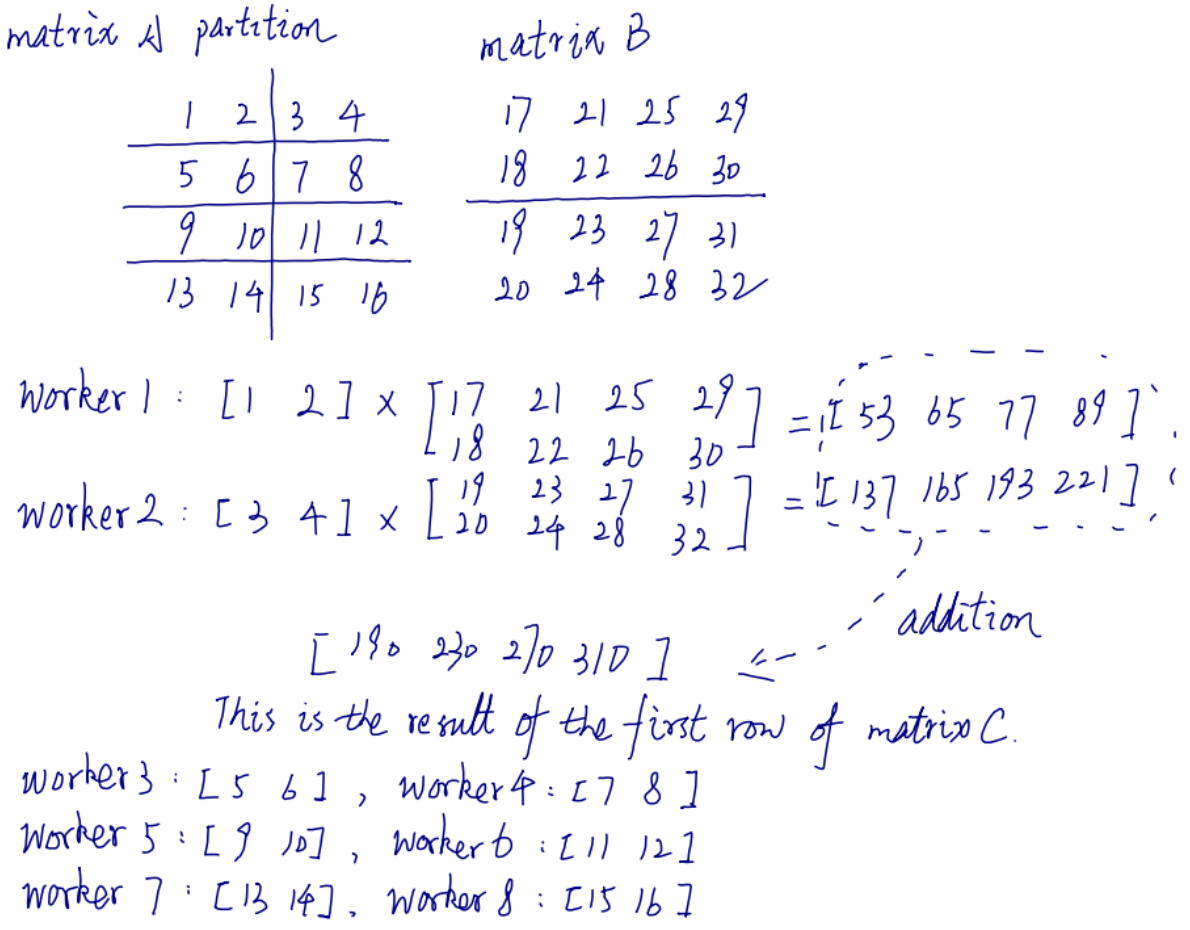
Methodology

Assume we have matrix A and B with size of N by N where N is powers of 2.
Matrix C will be the result of A x B.

1. Flatten A and B.
2. Partition matrix A.
3. Dispatch the partitions to different workers.
4. Merge results to matrix C.

Matrix A, B and C can be shared by all workers, because A and B are read-only and no one needs to modify them. Besides, in order to merge the results, we can just simply perform addition to matrix C. We only need to add a lock to matrix when we modify C.

Here is an example. Number of process = 8, matrix size = 4 * 4 = 16.



Result

Multiprocessing

Matrix size: 1
Brute force run time: 1.430e-05 s

Number of processors	Run time (seconds)
1	0.017744779586791992

Speed up: 1.0
Efficiency: 1.0

Matrix size: 2
Brute force run time: 1.812e-05 s

Number of processors	Run time (seconds)
1	0.004459381103515625
2	0.006894588470458984
4	0.008828401565551758

Speed up: [1.0, 0.646794384120617, 0.5051176105214832]
Efficiency: [1.0, 0.3233971920603085, 0.1262794026303708]

Matrix size: 4
Brute force run time: 3.386e-05 s

Number of processors	Run time (seconds)
1	0.0064792633056640625
2	0.005528926849365234
4	0.008667230606079102
8	0.011768817901611328

Speed up: [1.0, 1.171884432945235, 0.7475586609083157, 0.5505449536080386]
Efficiency: [1.0, 0.5859422164726175, 0.18688966522707892, 0.06881811920100482]

Matrix size: 16
Brute force run time: 8.380e-04 s

Number of processors	Run time (seconds)
1	0.021717548370361328
2	0.005528926849365234
4	0.015370607376098633
8	0.01767134666442871

Speed up: [1.0, 1.3764393000695094, 1.4129271432781647, 1.228969629919454]
Efficiency: [1.0, 0.6882196500347547, 0.3532317858195412, 0.15362120373993174]

Matrix size: 32

Brute force run time: 5.939e-03 s

Number of processors	Run time (seconds)
1	0.13787364959716797
2	0.08051490783691406
4	0.09130048751831055
8	0.20778179168701172

Speed up: [1.0, 1.7123990239973468, 1.510108580411655, 0.6635502008032128]

Efficiency: [1.0, 0.8561995119986734, 0.37752714510291374, 0.0829437751004016]

Matrix size: 64

Brute force run time: 4.589e-02 s

Number of processors	Run time (seconds)
1	1.0575041770935059
2	0.6256117820739746
4	0.641352653503418
8	2.2767128944396973

Speed up: [1.0, 1.690352079987622, 1.6488653649701788, 0.46448727886427654]

Efficiency: [1.0, 0.845176039993811, 0.4122163412425447, 0.05806090985803457]

Matrix size: 128

Brute force run time: 0.374 s

Number of processors	Run time (seconds)
1	8.01627516746521
2	4.656193733215332
4	4.890866279602051
8	19.20838475227356

Speed up: [1.0, 1.7216369478530216, 1.639029715635051, 0.41733208027897195]

Efficiency: [1.0, 0.8608184739265108, 0.40975742890876277, 0.052166510034871494]

Multithreading

Matrix size: 8

Brute force run time: 9.973e-04 s

Number of processors	Run time (seconds)
1	0.0009975
2	0.0009971
4	0.0009983
8	0.002992

Speed up: [1.000e+00, 1.000e+00, 9.992e-01, 3.334e-01]

Efficiency: [1.000e+00, 5.002e-01, 2.498e-01, 4.167e-02]

Matrix size: 16

Brute force run time: 9.975e-04 s

Number of processors	Run time (seconds)
1	0.00199
2	0.001995
4	0.002946
8	0.002992

Speed up: [1.000e+00, 9.975e-01, 6.755e-01, 6.651e-01]

Efficiency: [1.000e+00, 4.987e-01, 1.689e-01, 8.314e-02]

Matrix size: 32

Brute force run time: 4.986e-03 s

Number of processors	Run time (seconds)
1	0.01995
2	0.01396
4	0.02098
8	0.01499

Speed up: [1.000e+00, 1.429e+00, 9.509e-01, 1.331e+00]

Efficiency: [1.000e+00, 7.145e-01, 2.377e-01, 1.664e-01]

Matrix size: 64

Brute force run time: 3.587e-02 s

Number of processors	Run time (seconds)
1	0.1028
2	0.1337
4	0.1108
8	0.1057

Speed up: [1.000e+00, 7.689e-01, 9.278e-01, 9.726e-01]

Efficiency: [1.000e+00, 3.844e-01, 2.319e-01, 1.216e-01]

Matrix size: 128

Brute force run time: 2.698e-01 s

Number of processors	Run time (seconds)
1	1.434
2	0.9179
4	0.9261
8	0.9902

Speed up: [1.000e+00, 1.562e+00, 1.548e+00, 1.448e+00]

Efficiency: [1.000e+00, 7.811e-01, 3.871e-01, 1.810e-01]

Analysis and Conclusion

In both multiprocessing and multithreading:
1. The sequential brute force version always outperform the parallel versions.
2. The best performance occurs when we dispatch the job to **two workers**.
3. As the number of workers grow, the run time becomes longer and the speed up and efficiency go down. Obviously, the overhead of creating, destroying, switching process is larger compared to the benefits it brings.
4. Multithreading version is much faster than multiprocessing version since communication among threads is on user level but not kernel level. In this case, threads share space of process address so that they don't need to large overhead to communicate with each other.