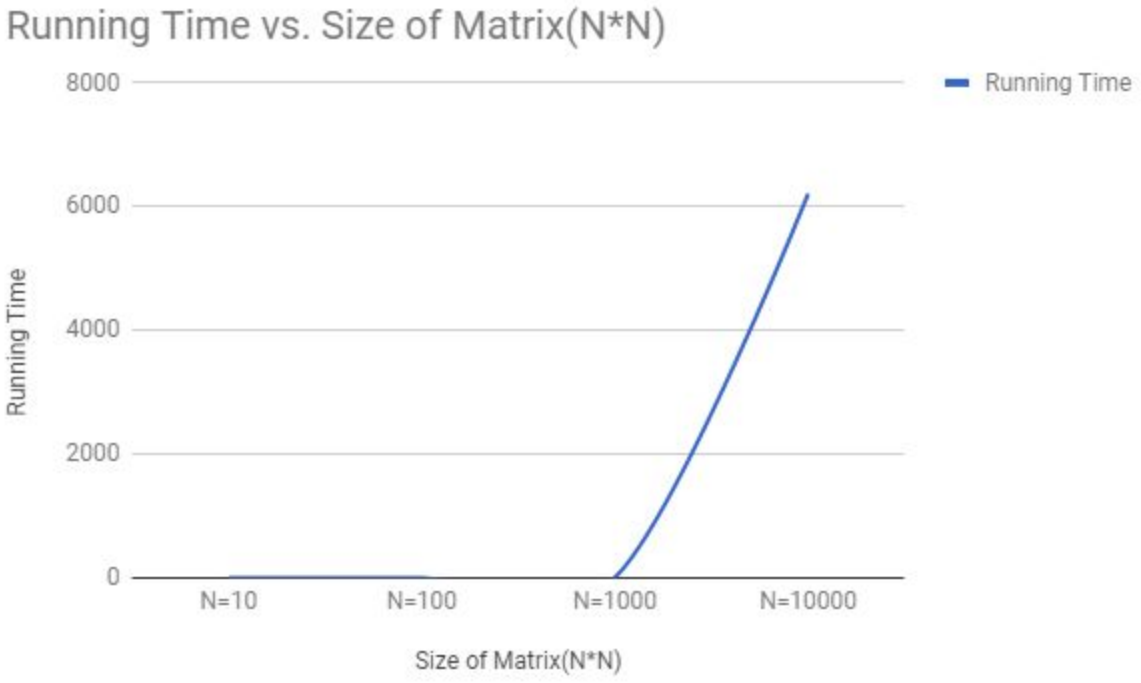
Author: Ca11me1ce
Task: Matrix-Matrix Multiplication

Serial Version

NOTE: The serial-version code is combined with the OpenMP version in file - matrix_mul.c
Compile Command: $gcc matrix_mul.c -o matrix_mul.out -fopenmp
Running Command: $./matrix_mul.out <NUM_THREADS> <N>

Serial-Version Running Time

Size of Matrix(N*N)	N=10	N=100	N=1000	N=10000	...
Running Time	0.000007	0.005457	5.445931	6211.239023	...



In the serial version of matrix-matrix multiplication, when the value of N is increased, the running time is increasing. I also tested parallel version of N=10 and N=100, and the serial version is always faster than or equal to the parallel version when N is less than 100. When N is increasing 10 times, the running time is increasing 1000 times. Since there are three nested for-loop in the serial version, the time complexity is $O(n^3)$. Therefore, the time complexity is

always increasing 1000 times when N is increasing 10 times. Then the serial version can run any size of the matrix, but the running time is always increasing to a very big number. The serial version is easy to implement, and the code is easy to understand. So if you want to handle small size matrix multiplication, the serial version is the best method to use in your design.

Parallel Version

NUM_THREADS=10

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	986.634529	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.012232	0.021212	0.032132	0.052313	0.632320
Summa MPI	0.016518	0.023892	0.032531	0.043931	0.063212

NUM_THREADS=20

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	712.362738	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.022093	0.032301	0.042321	0.055232	0.092023
Summa MPI	0.021882	0.029039	0.042039	0.059032	0.086232

NUM_THREADS=30

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	482.23092	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.032021	0.042102	0.070892	0.082032	0.092638
Summa MPI	0.033342	0.046393	0.065343	0.073829	0.095938

NUM_THREADS=40

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	613.256213	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.039092	0.055902	0.076239	0.098231	0.121943
Summa MPI	0.040231	0.056291	0.076372	0.097382	0.156723

NUM_THREADS=50

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	634.728492	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.000320	0.000382	0.000442	0.000558	0.000732
Summa MPI	0.000267	0.000312	0.000421	0.000542	0.000621

NUM_THREADS=60

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	764.203802	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.000238	0.000329	0.000347	0.000549	0.000628
Summa MPI	0.000197	0.000292	0.000331	0.000462	0.000565

NUM_THREADS=80

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	Almost same with NUM_THREADS=60	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.00125	0.000214	0.000319	0.000394	0.000492
Summa MPI	0.000107	0.000182	0.000262	0.000360	0.000415

NUM_THREADS=100

	N=10^4	N=10^5	N=10^6	N=10^7	N=10^8
OpenMP	Almost same with NUM_THREADS=60	Increasing 1000 time from previous running time	Seg fault	Seg fault	Seg fault
Cannon MPI	0.000102	0.000129	0.000231	0.000329	0.000519
Summa MPI	0.000097	0.000149	0.000172	0.000269	0.000322

OpenMP:

Compile Command: Please see the compile command of the serial version. They are in the combined c source file.

According to the record and the graph chart of OpenMP, we can know that the running time is increasing when N is increasing. When N is increasing 10 times, the running time is increasing 1000 times. Then when I tested N=10^6 and above, it returns the segmentation fault because of OpenMP's restriction of the single processor. I also tested the number between 10^5 and 10^6, therefore I can know that the maximum size of the matrix is 176679 that the OpenMP version can handle. Then the running time is decreasing when NUM_THREADS is increasing. But when N>=40, the running time is increasing or steady from the running time of N=30. I also tested NUM_THREADS from 30 to 40, and I get the maximum of NUM_THREADS is 32 that is the best design for the design of OpenMP version. Then the NUM_THREADS from 32 to 100 that is not very useful for the OpenMP version.

Cannon MPI:

Compile command: $module load intel

$module load impi/5.1.3
$mpicc cannon_mpi.c -o cannon_mpi.out

Running command: $mpirun -np <NUM_THREADS> ./cannon_mpi.out

According to the record and the graph chart of Cannon MPI, we can know that the running time is increasing when N is increasing. The cannon algorithm is very fast. The cannon MPI version can handle the matrix which size is greater than 108 *108, and it can handle very big size matrix multiplication. Then when NUM_THREADS is between 10 and 50, the running time is increasing or steady. By NUM_THREADS>=50, the running time is decreasing, and more threads would decrease more running time. That means it will have very low running time when NUM_THREADS=100. And It can handle any large size matrix multiplication. Then this algorithm only can handle the square matrix that means the number of rows is equal to the number of columns in each matrix.

Summa MPI:

Compile command: $module load intel
$module load impi/5.1.3
$mpicc summa_mpi.c -o summa_mpi.out

Running command: $mpirun -np <NUM_THREADS> ./summa_mpi.out

According to the record and the graph chart of Summa MPI, we can know that the running time is increasing when N is increasing. The summa algorithm is very similar with the cannon algorithm and also very fast. The summa MPI version can handle the matrix which size is greater than 108 *108, and it can handle very big size matrix multiplication. Then when NUM_THREADS is between 10 and 50, the running time is increasing or steady. By NUM_THREADS>=50, the running time is decreasing, and more threads would decrease more running time. That means it will have very low running time when NUM_THREADS=100. And It can handle any large size matrix multiplication. This algorithm has an advantage, that is: when calculating matrix A*B=C, then A's size is m*I and B's size is I*n, then m, n, and I can be three different numbers. This advantage does not apply the cannon algorithm.

Conclusion

According to the record and analysis, we can conclude that the serial-version implementation is the best method if you just handle the very small size of matrices. Then if you want to the middle size of matrices, you can use OpenMP-version implementation. Also, the serial version and OpenMP version are very easy to implement, and you do not need much time to design these two codes. The implementation of MPI is very hard to design, and the cannon algorithm and the summa algorithm are both difficult to understand; they will spend your much time. However, these two MPI implementations are super fast and super fast, and their running times are very similar. The running time is very low. It is very effective to use for the complex machine and computation. Because it is very powerful in large-number size and more

processor.Then they can handle different sizes of matrices. The cannon algorithm can only handle matrices with N*N size. The summa algorithm can handle matrices with N*N size, and it can also handle between matrix A with size m*I and matrix B with size I*n. m, n, and I can be different numbers. Therefore, the summa algorithm is better than the cannon algorithm.

Reference

1. https://blog.csdn.net/xx_123_1_rj/article/details/39184105
2. https://gist.github.com/rehrumesh/b103636b6337baafb52f
3. https://github.com/suraj-deshmukh/cannon-algorithm-in-c-using-mpi/blob/master/cannon.c