CSE 6220 Programming Assignment 3 Report

Jingyi Feng, Ziya Ye, Wei Zhou

April 24, 2024

1 Theoretical run time analysis

Suppose we have a matrix of size n and sparse parameter s, then our expected number of non-zero entries will be $n^2 \cdot s$. As our algorithm only involves the non-zero entries of the sparse matrix, its is important if we analyze the running time concerning sparsity.

Communication Time

In matrix transposing part, we use one All-to-All communication primitives and one Many-to-Many communication primitives. The messages in All-to-All are just numbers of non-zero entries in each $\frac{n}{p} \times \frac{n}{p}$ block, therefore it uses $O(\tau \log p + \mu p \log p)$ communication time. The message in Many-to-Many can vary based on the non-zero entries, we use the average here. Assume the number of non-zero entries in each $\frac{n}{p} \times n$ block is $\frac{n^2 s}{p}$, the communication time of Many-to-Many will be $O(\tau p + \mu \frac{n^2 s}{p})$.

In matrix multiplication part, we use ring permutation to communicate the message. We have done p times communications and we send the non-zeroes entries in $\frac{n}{p}$ columns of matrix B in each permutation. Therefore, our communication time will be $O(\tau p + \mu n^2 s)$.

Overall, our communication time will be $O(\tau p + \mu(p \log p + n^2 s))$

Computation Time

In matrix transposing part, we need to calculate the displacement of the tuples in the destination processor after MPI communication, which takes $O(p)$. In All-to-All and Many-

to-Many, our computation time is $pO(1) + \log pO(1) \approx O(p)$.

In matrix multiplication part, we use two-pointer and dot product to calculate the matrix multiplication result of each pair of $\frac{n}{p} \times n$ and $n \times \frac{n}{p}$ blocks, where each block has $\frac{n}{p} * n * s$ number of elements. Thus, the computation time of this local matrix multiplication is $O(ns * \frac{n}{p} * \frac{n}{p}) = O(\frac{n^3 s}{p^2})$. For each processor, we repeat this part p times as we need to rotate the B matrix. Thus the total local matrix multiplication time is $O(\frac{n^3 s}{p^2} * p) = O(\frac{n^3 s}{p})$. Moreover, in two pointer, there is a bit of overhead where we need to iterate through the local A and local B matrix (each with number of non-zero entries $\frac{n}{p} * n * s$)p times. Thus the computation time of this overhead is $O(\frac{n^2 s}{p} * p) = O(n^2 s)$. Finally, before we can do two pointer, each processor need to sort their local A and local B matrix one time, which takes $O(\frac{n^2 s}{p} \log \frac{n^2 s}{p})$. Thus the total computation time for matrix multiplication is $O(\frac{n^3 s}{p}) + O(n^2 s) + O(\frac{n^2 s}{p} \log \frac{n^2 s}{p}) = O(\frac{n^3 s}{p})$. Thus, our overhead with two pointers and sorting didn't really impact the overall runtime, as we are able to achieve the optimal parallel computation time $O(\frac{n^3 s}{p})$ compared to the serial computation time $O(n^3 s)$.

Run-time

Combining the communication time and the computation time we have above, our final run time should be $O(\frac{n^3 s}{p} + \tau p + \mu(p \log p + n^2 s))$. If we directly take matrix multiplication of the original matrix, the run time is $O(n^3 s)$. We can see the running time is largely reduced.

Bonus Runtime In bonus, we use 2d partitioning and cannon's algorithm to achieve speedup. Each processor will have $\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}}$ elements.

1): We initialize local A and B which takes O($\frac{n^2}{p}$) time.

2): We use cannon's algorithm to shift $A_{i,j}$ left by i and $B_{i,j}$ up by j. We use MPI_Isend and MPI_Recv to only shift once instead of at most $\sqrt{p}$ times. The communication time for the shift is O($\tau + \mu * \frac{n^2 * s}{p}$).

3): We continue to perform cannon's algorithm to multiply each local matrix A and matrix B to have local sum matrix C. The multiplication follows the same two pointer methodology as mentioned in baseline, except the size is $\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}}$. The sorting takes O($\frac{n^2 s}{p} \log \frac{n^2 s}{p}$) time.

p=16, e=0.01	Runtime(s)
n=1600	0.002266
n=4800	0.017330
n=9600	0.110398

Figure 1: Matrix size vs. Run time

The two pointer takes $O(\frac{n^2s}{p} + \frac{n^3s}{\sqrt{p}^3})$. Finally, we will repeat the process $\sqrt{p}$ times. The total computation time plus the sorting time here is : $O(\frac{n^2s}{\sqrt{p}} + \frac{n^3s}{p} + \frac{n^2s}{p}\log\frac{n^2s}{p}) = O(\frac{n^3s}{p})$. Total communication time is: $O(\tau\sqrt{p} + \mu * \frac{n^2*s}{\sqrt{p}})$

The total communication time is: $O(\tau\sqrt{p} + \mu\frac{n^2s}{\sqrt{p}})$

The total computation time is: $O(\frac{n^3s}{p})$.

Compared to our baseline method, we didn't achieve 2 times speedup, mostly because the computation time of our baseline algorithm is already optimal, which is the same for our bonus algorithm as well. However, our bonus algorithm still manages to achieve some speedup as the communication time of our bonus algorithm is $O(\tau\sqrt{p}+\mu\frac{n^2s}{\sqrt{p}})$ which is smaller than our baseline by at least a factor of $\sqrt{p}$: $O(\tau p + \mu(p\log p + n^2 s))$.

2 Experiment and observation

We implemented experiments to support our theoretical analysis. In the experiment, we want to find the influence factors and how they can impact the program. We mainly tested 3 parameters: matrix size n, sparse parameters s and number of processors p.

Matrix size

Figure 1, 2, 3 shows the how matrix size can effect the running time of the program. We fixed the sparsity to be 0.01 and the processor number to be 16. The image shows the running time increases with the increasing of matrix size. The curve is more than quadratic by observing the ratio between matrix size and running time. This is closed to the run time complexity as we calculated. As n become large, the term $\frac{n^3s}{p}$ is dominating the run time.

p=16, e=0.01	Runtime(s)
n=1600	0.001499
n=4800	0.016888
n=9600	0.107923

Figure 2: Matrix size vs. Run time(Bonus)

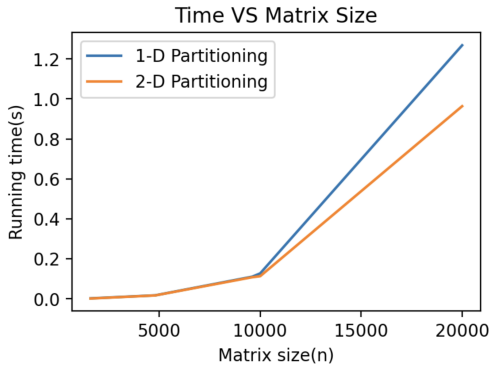

Figure 3: Matrix size vs. Running Time

Sparsity

Figure 4 shows the how sparsity can effect the running time of the program. We implemented totally 4 experiments on different number processors with matrix size $n = 10000$. The results are surprisingly consistent and all show the running time increases linearly with the increasing of density(decreasing of sparsity). In our theoretical run time, $n^3 s$ is dominating as $n \to \infty$. Hence, the linear curve shows a strong proof of the uniformity of the calculated run time and practical run time.

Scalability

To investigate the scalability of our program, we tested our program on different number $(2, 4, 8, 16)$ of processors. We fixed the matrix size to be $n = 10000$, and did 3 tests on different sparsity $(0.1, 0.01, 0.001)$. Figure 4 and the table below shows the results of our experiments. Overall, we can see the strong scalability in the pattern because the curve is decreasing in super-linear trend.

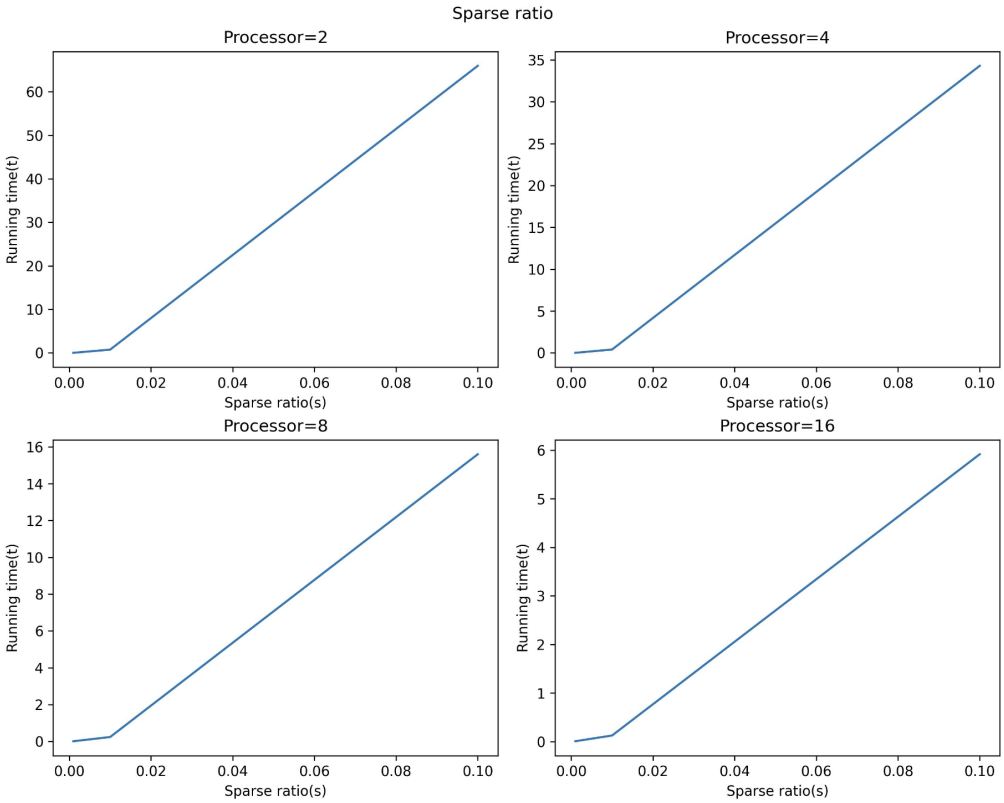

Figure 4: Sparse ratio vs. Running time

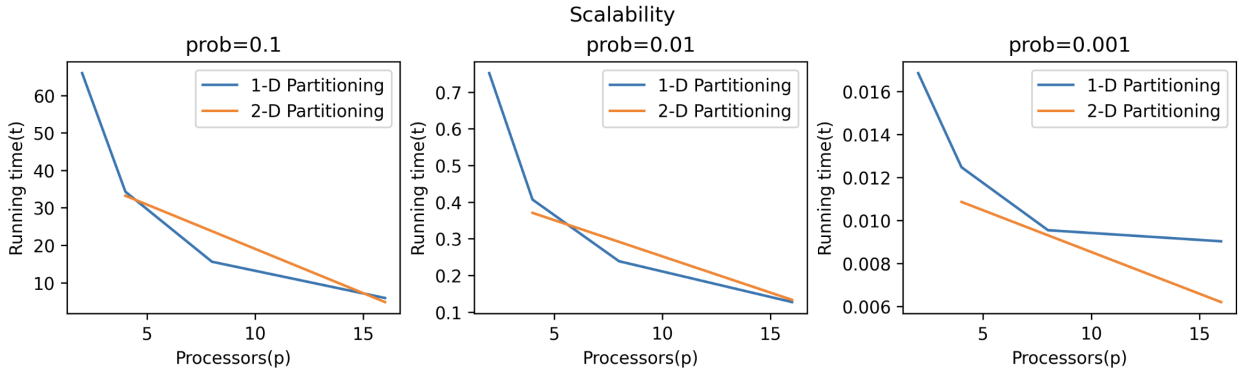

Figure 5: Scalability under different initial possibility

Sparsity(s)	Processor(p)	Baseline(1-D Partitioning)	Bonus(2-D Partitioning)	Speedup
e=0.1	p=2	65.974339	-	-
	p=4	34.302423	33.23639	1.04
	p=8	15.599698	-	-
	p=16	5.914406	4.823155	1.23
e=0.01	p=2	0.751782	-	-
	p=4	0.406782	0.370636	1.098
	p=8	0.238597	-	-
	p=16	0.127186	0.133175	0.956
e=0.001	p=2	0.016861	-	-
	p=4	0.012480	0.010864	1.15
	p=8	0.009547	-	-
	p=16	0.009031	0.006201	1.46

Extra experiments

We also did some extra experiments on the scalability on matrix of size $n = 20000$. As we can see from the table in Figure 6 and 7, there is a huge decrease on the run-time, around 20 seconds, when we increase the size of the input. Comparing to the run time decrease from 16 processors to 8 processors when $n = 10000$, we can conclude that the scalability is powerful when data size is large. We can have reasonable guess that the scalability will

n=20000	p=4	p=8	p=16
e=0.1	275.401953	154.975172	87.962074
e=0.01	3.077834	1.938872	1.268406
e=0.001	0.055631	0.046500	0.034129

Figure 6: Extra experiments

n=20000	p=4	p=16
e=0.1	260.278365	84.739738
e=0.01	2.925677	0.963607
e=0.001	0.052942	0.021116

Figure 7: Extra experiments(Bonus)

become higher as data size increases.

3 Summary

In conclusion, our program well follows our calculated run time and shows strong scalability. The efficiency of our algorithm is the optimal as follows, where n can be chosen with reasonable consideration of n and s to ensure high efficiency.

$$\begin{aligned} E(p) = \frac{T(n,1)}{pT(n,p)} &= \frac{O(n^3 s)}{p * O(\frac{n^3 s}{p} + \tau p + \mu(p \log p + n^2 s))} \\ &= \Theta(\frac{n^3 s}{O(n^3 s + \tau p^2 + \mu(p^2 \log p + p n^2 s))}) \\ &\approx \Theta(1) \end{aligned}$$

since $O(\tau p^2 + \mu(p^2 \log p + p n^2 s)) = O(n^3 s)$

4 Contribution

Baseline Algorithm	Wei Zhou, Ziya Ye
Bonus Algorithm	Jingyi Feng
Experiment & Plot	Jingyi Feng
Readme	Jingyi Feng, Wei Zhou
Report	Jingyi Feng, Ziya Ye, Wei Zhou