Parallel Programming
Homework 3: Fully Distributed Shortest Path Algorithms

王鈺鎔 106062600

Implementation

SSSP

I extend the original Moore's single source shortest path algorithm to a distributed and asynchronous Moore's single source shortest path algorithm based on MPI. I use the adjacent matrix as the based data structure. Firstly, I distribute the vertices to every process. Every process will get the whole graph (adjacent matrix) in the beginning. However, it only handles the vertices assigned to it. When the distances of the assigned vertices are updated, the process will send the updated distances to other related processes, which is also called "reactive" other processes. The other processes getting the updated distances will determine their distances need to be relaxed or not. The relax progress is shown in fig. 1. In fig. 1, for example, the red line which is the updated distance from k to c. If the original distance from a to c (blue line) is larger than the distance from a to k (black line) plus the distance from k to c (red line), then update the distance blue line to the black plus the red line.

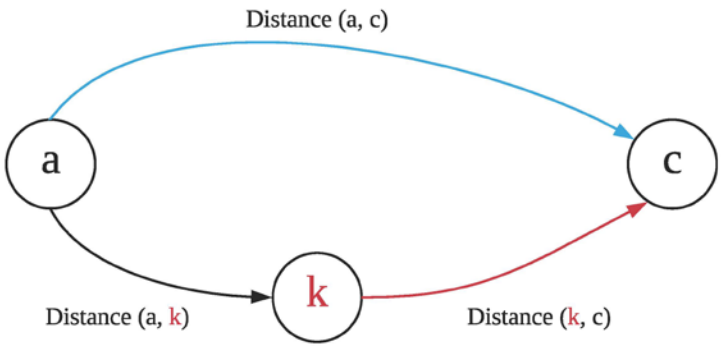

Figure. 1. For example, the red line which is the updated distance from k to c. If the original distance from a to c (blue line) is larger than the distance from a to k (black line) plus the distance from k to c (red line), then update the distance blue line to the black plus the red line.

APSP

I extend the original Floyd-Warshall's synchronous all pair shortest path

algorithms to a distributed and synchronous Floyd-Warshall's all pair shortest path algorithms based on MPI and OpenMP. I use the adjacent matrix as the based data structure. Firstly, I distribute the vertices to every process. Every process will get the whole graph (adjacent matrix) in the beginning. However, it only broadcast other processes the distances whose source is held by itself afterward. The progress is shown in the fig. 2. In the kth turn, process containing kth vertex broadcast other processes its k row. Other processes get the new row to update their distances from vertex k to all vertices. Since the distances from k to all vertices are updated, every process then checks the distances going through vertex k to determine whether to be "relaxed" or not. The relax method is similar to sssp and shown in fig.1.

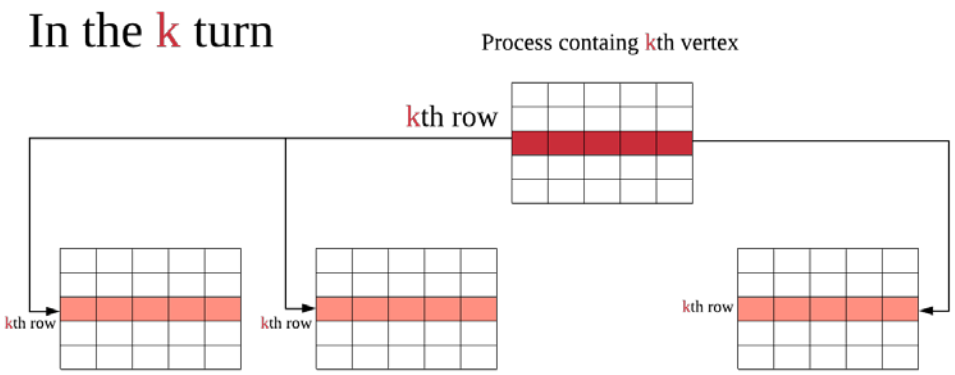

Figure. 2. In the kth turn, process containing kth vertex broadcast other processes its k row. Other processes get the new row to update their distances from vertex k to all vertices.

Partition

Firstly, I divide the vertex number by the process number and put the remainder to the last process. Therefore, we define the number of vertices in each process. Put the vertex 0 in process 0 if the vertex 0 has not to be put in any processes. Secondly, we take vertex as the next vertex and put it in process 0 if vertex 0 has a distance to a and vertex a has not been put in any processes. If there is no suitable vertex to be the next vertex. Choose another vertex which has not been put in a process. Repeat the progress until process 0 get enough vertices. Repeat the above progress for every process.

(a). *Why graph partition can have significant performance impact to your implementation?*

SSSP

Good graph partition chooses a group of closed vertices and put in the same process. Since if the vertices are closed the updated distances may reactive the same processes or even reactive itself process. However, graph partition improves little since the graph partition does not group vertices well.

APSP

In the apsp, graph partition has little improvement. The reason is that apsp is a dynamic algorithm, which means the total computing number is the same. That is, when the number of vertices in each process is fix, no matter vertices in which process, the numbers of turns and computing are fixed as well.

(b). *What are the pros and cons of synchronous and asynchronous*

Synchronous

- Pros: The implementation is easy. *Synchronrous* is easy since every process does the similar at the same time.
- Cons: It executes slower than *asynchronous* one. The reason why *synchronous* is slower is same as its pros. Thus, when there is a black sheep process, it will be the bottleneck of the whole program.

Asynchronous

- Pros: It executes faster than a synchronous one. The reason is that every process does not need to wait for other processes except data dependency. That is, it avoids the whole program blocks in some long-time processes.
- Cons: The implementation is hard. Asynchronrous is hard since processes are in the different stages at the same time. The data dependency needs to be handled. The terminal criteria are also hard to implement since each process does not know other process's status.

(c). *Other efforts you've made in your program*

When using omp to implement apsp, I choose dynamical omp instead of static omp to balance the execution time among thread and avoid the process blocks in some long-time threads, which can reduce execution time and increase scalability better.

Experiment & Analysis

Methodology

(a). System Spec

Using the provided cluster to measure.

(b). Performance Metrics

Use `MPI_Wtime()` to measure total time and time profile.

Plots

(a). Strong Scalability

Test data: random_5000, dense_5000, skew_5000

In all the experiment results, I choose the longest core time to record. I do the 4 experiments to test the scalability of SSSP and APSP in single node and multi nodes with 3 different density data.

1. In fig. 3, I test the speedup of SSSP in single node condition.
2. In fig. 4, I test the speedup of APSP in single node condition.
3. In fig. 5, I test the speedup of SSSP in multi-node condition.
4. In fig. 6, I test the speedup of APSP in multi-node condition.

We can observe that the scalability of *synchronous* algorithm (APSP) is better than the scalability of *asynchronous* algorithm (SSSP) from figures. The greatest scalability is when running *synchronous* algorithm (APSP) in less than 16 cores. In the contrary, the scalability of *asynchronous* algorithm (SSSP) is bad in any number of cores. The reason of the bad scalability of the both algorithms is dramatically fluctuating IO time. Fig. 10 shows a part of time profile in APSP with the data random_5000. We can find that the IO time dramatically fluctuate from cores 24 to 27.

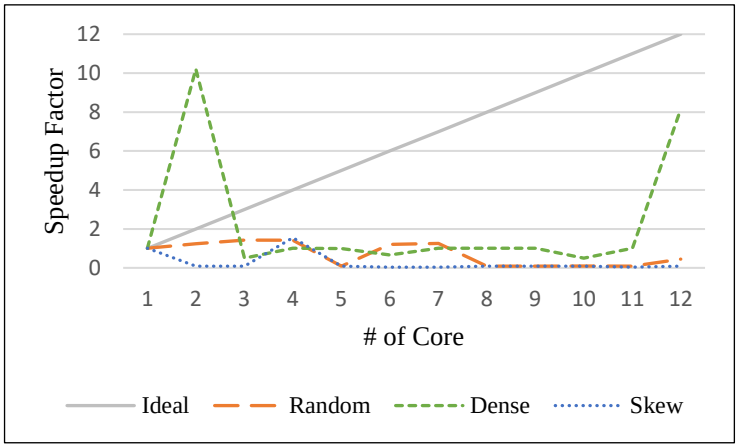

Figure 3. SSSP's speedup in single node.

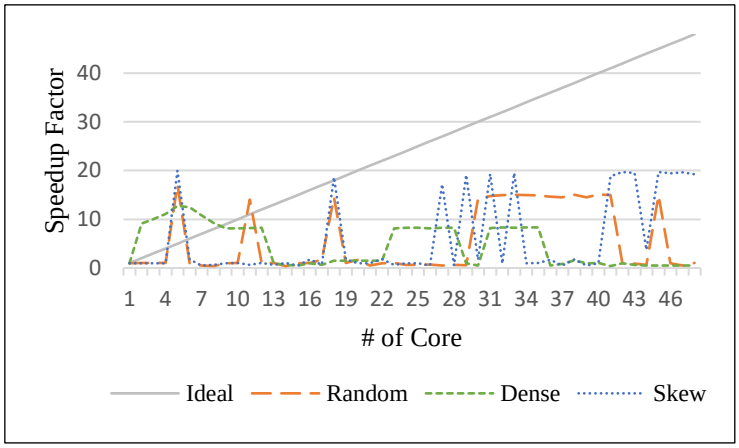

Figure 4. SSSP's speedup in multi nodes.

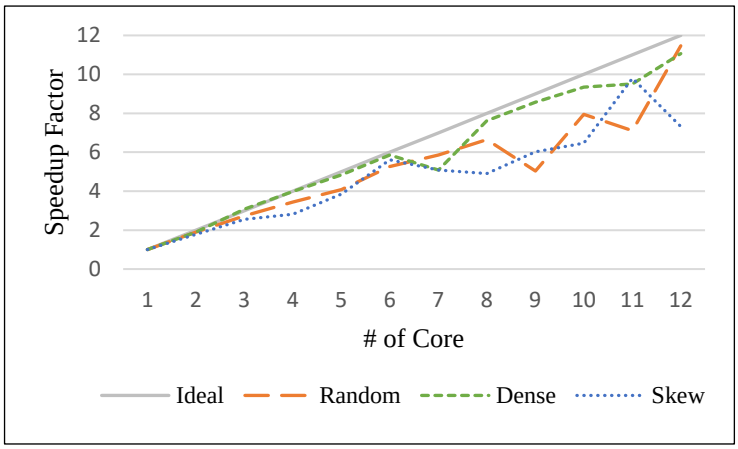

Figure 5. APSP's speedup in single nodes.

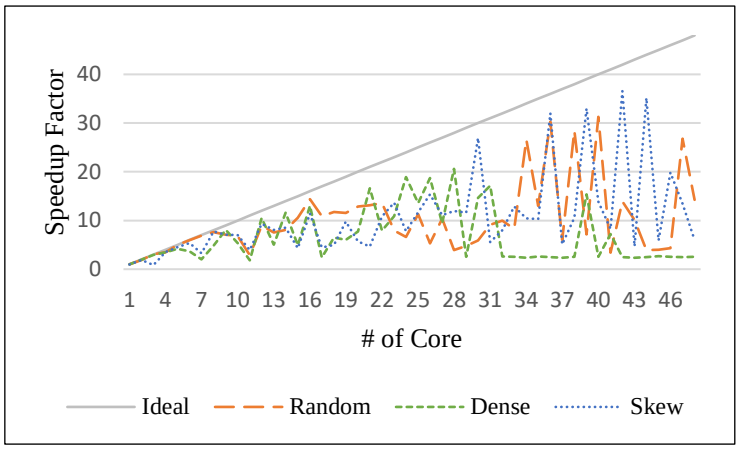

Figure 6. APSP's speedup in multi nodes.

# of Cores	Total time (s)	CPU time (s)	Network time (s)	IO time (s)
24	37.0423	10.1961	0.581347	26.2649
25	21.6879	9.78813	0.531522	11.3683
26	46.0545	9.10195	0.798567	36.154
27	23.5436	8.72541	1.05544	13.7628

Figure 10. A part of time profile in APSP with the data random_5000

(b). Load Balancing

Test data: skew_5000

In fig. 7, I test the load balancing of SSSP in multi-node condition. I also record the time profile of each core.

Since core 0 need to write the result into result file, the execution time of core 0 is about twice longer than other cores. However, except the core 0, the load of other cores is balance. One of the reasons is that the IO time is much longer than other kinds of execution time, and the IO time of each core is similar since that read the same data file and the same data size

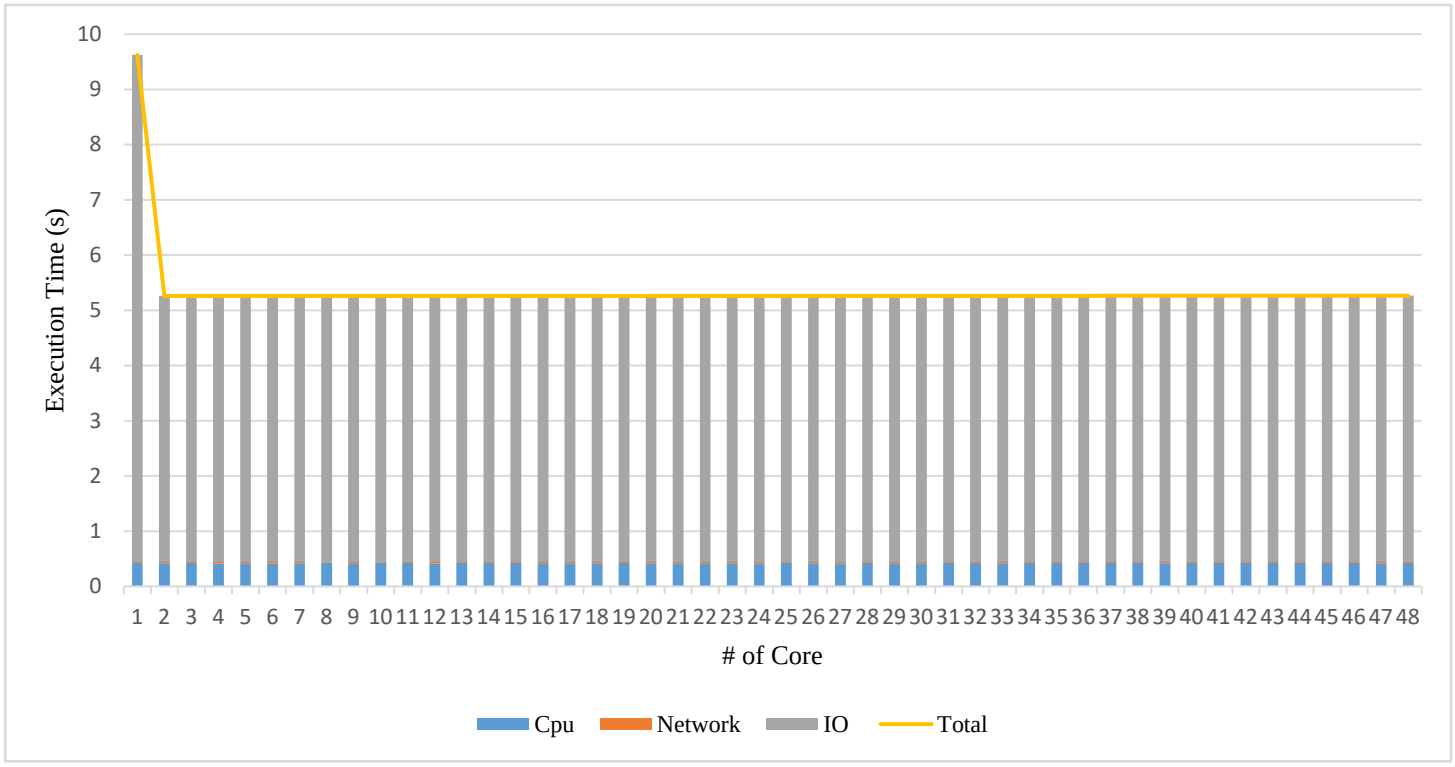

Figure 7. Load balance and time profile among each core for SSSP.

(c). Graph Partition

Test data: random_5000, dense_5000, skew_5000

In all the experiment results, I choose the longest core time to record. I do the 4 experiments to test the execution time of SSSP and ASAP with and without graph partition in single node.

1. In fig. 8, I test the execution time of SSSP without/with partition in single node.
2. In fig. 9, I test the execution time of ASAP without/with partition in single node.

According to my graph partition algorithm, the result groups are similar to the original groups. Thus, the execution time of the algorithm with and without partition should be similar. However, the IO time fluctuates dramatically, so the result looks messy.

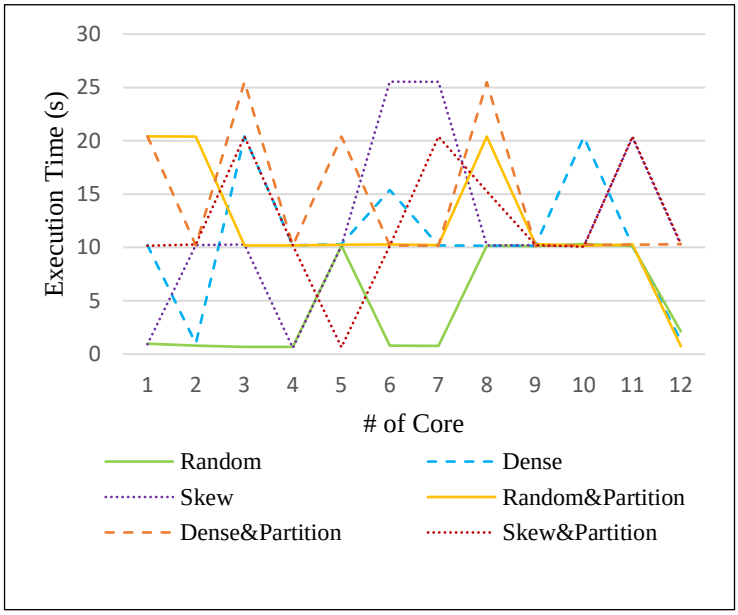

Figure 8. Execution time of SSSP without/with partition in single node.

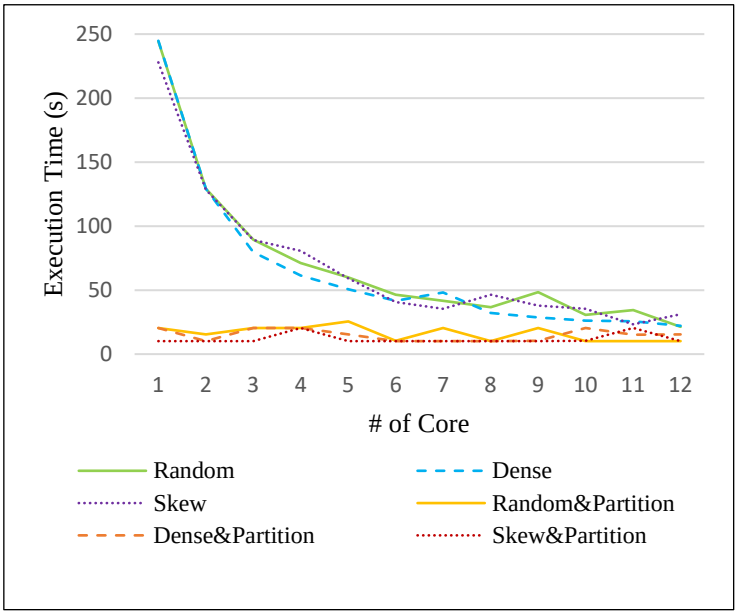

Figure 9. Execution time of APSP without/with partition in single node.

Experiences / Conclusion

(a). Conclusion

Compared to *asynchronous* algorithm (SSSP), *synchronous* algorithm (ASAP) has better strong scalability. However, when the number of cores are more than 16, the IO time dramatically fluctuates, so the scalability becomes bad. The same circumstance occurs in graph partition, which cause the result both with and without partition messy. Since the fix IO time dominate the execution time of each core, the load of each core in *asynchronous* algorithm (SSSP) is balance.

(b). Experiences

In this homework, the most difficult assignment is how to implement *asynchronous* algorithm (SSSP), especially the termination mechanism. From this homework, I can understand why we all know synchronous algorithm is slower than asynchronous one, but we still choose synchronous algorithm rather than asynchronous one sometimes.