EE451 Fall 2019
Course Project Final Report
Pengcheng Li, Yi Sui, Liheng Lin

Parallelization of Kempe's Influence Maximization Algorithm

1 Introduction

Influence Maximization is a NP-hard problem such that it takes a long time to solve. In this project, we implement a serial version of Kempe's algorithm to this problem and parallelize it with three strategies: Static Mapping, Dynamic Mapping and Graph Partitioning. We implement our parallel versions using OpenMPI. We test our program on a Bitcoin OTC trust network from Stanford Large Network Dataset Collection. We achieve 4.0x speedup with Static Mapping on 4 processors, 3.0x speedup with Dynamic Mapping on 4 processors and 1.1x speedup with Graph Partitioning.

2 Influence Maximization

While social networks, such as Twitter, Facebook and Snapchat, play an increasingly important role in modern life, maximizing influence over social networks[1, 2] turns out to be a valuable problem to be explored. Here, each user in the network has its own influence, measuring this user's power of imposing impacts on other users. We are interested in selecting a set of users of a fixed size such that the overall influence of these users is maximized. For instance, in advertising, we may want to select a few powerful users as our starting point so that these users could recommend our products to as many people as possible.

We regard a social network as a weighted directed graph $G=(V, E, w)$, where V is the vertex set, E is edge set and w is a mapping from an edge to a weight in [0, 1]. All nodes in V are inactive at the beginning, but they could be turned into active later by incident nodes. Weight of an edge from node u to node v represents the probability that active u could activate v successfully.

Now we select a node set S as our seeds, and nodes in S start to try activating their neighbors according to edge weights. Each edge could only be tried once. The process terminates when no new nodes could be activated any more. We denote the expectation of the number of active nodes in G as $\sigma(S)$, the influence of set S.

The Influence Maximization Problem is that, given a number k, we are looking for a set S of size k that maximizes $\sigma(S)$.

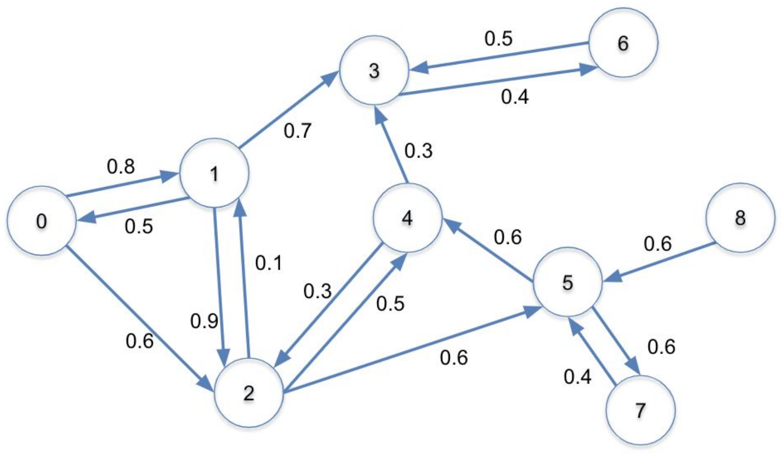

Figure 1

We shall explain the problem with an example. In the following graph, we choose seed set as {1}. Now, with probability 0.7 vertex 1 could activate vertex 3. Suppose that this is successful. Vertex 1 could also activate other nodes but let us assume that all these fail. Now vertex 3 could activate vertex 6 with probability 0.4. Suppose that this is also successful. Now vertex 6 try to activate vertex 3 but it is already active. Now all activation is done and we could conclude that $\sigma(\{1\}) = 3$ as vertices {1, 3, 6} are active.

3 Kempe's Greedy Influence Maximization Algorithm

Despite the fact that this problem is NP-hard[1], a few approximate algorithms have been proposed. One of them is Kempe's Greedy Algorithm[1]. Intuitively, we, starting from an empty set, greedily add a node *x* to *S* such that $\sigma(S \cup \{x\})$ is maximized until the size of *S* reaches *k*. We formally present this algorithm as follows:

```
Kempe's Greedy Algorithm:
S = ∅
while |S| < k:
        for each node x in V-S:
                simulate σ(S∪{x}) by sampling for a large number of times
        end for
        x*=argmax σ(S∪{x})
        S=S∪{x}
end while
return S
```

Code 1

Notice that since the activation process is stochastic, for every single *x,* we have to calculate $\sigma(S \cup \{x\})$ by repeatedly applying the process for sufficiently large number of times.

4 Platform

We use OpenMPI to implement our project as message passing model fits our parallelization methods, especially Graph Partitioning when a single process does not own the entire data.

5 Hypothesis and Parallelization

We take three approaches to tackle the parallelization task. As everyone could notice, the job to sample $\sigma(S \cup \{x\})$ for multiple times could be parallelized because each sampling instance is independent from others. Moreover, the for loop in the algorithm is also parallelizable because there is no data dependency in the loop. Hence, we could exploit this to parallelize Kempe's algorithm.

5.1 Static Mapping

A simple strategy to distribute these independent jobs is to evenly assign them to processes. When given a seed set *S*, we would consider each vertex *x* as a candidate and calculate $\sigma(S \cup \{x\})$. If the size of the graph is $|V|$, we have almost |V| candidates and hence $|V|$ independent jobs. In Static Mapping, we evenly and statically divide these candidates as $|V|/p$ blocks, where p is the number of processes. Each process is responsible for calculation of a block and select the best candidate inside the block. After All processes finish calculation, a master process gather *p* best candidates from *p* processes along with their influence value, and find out the actual best candidate over these *p* candidates.

For instance, if we have already chosen a seed set $S = \{1\}$ and vertex set $V = \{0, 1, 2, …, 3999\}$, and we have 4 processes, we would let process 0 compute $\sigma(S \cup \{x\})$ for x = 0, 1, …, 999, process 1 for x = 1000, 1001, … , 1999, process 2 for x = 2000, …, 2999 and process 3 for x = 3000, ..., , 3999. After all is finished, process 0 would collect results from all 4 processes.

We show the pseudocode as follows:

```
while(current seed size < seed size required){
        Broadcast seed to each worker;
        Master calculates block size and sends out block to workers;
        Master and workers calculate average influence of vertices they own;
        Workers select the vertex with max influence in their block;
        Master gathers vertices selected from last step and finds the one with max
        influence, adds it to the seed;
}
```

Code 2

5.2 Dynamic Mapping

As the activation procedure in Influence Maximization is randomized, we notice that the size of influenced set from the same seed set varies drastically, leading to unbalanced loads between different processes. Hence, we believe that dynamically distribute jobs to different processes would help accelerate our parallel algorithm.

In Dynamic Mapping, we employ a Master-Worker model to control the dynamic process. We have a master process who is only responsible for distributing jobs and collecting results. It maintains a list of unfinished jobs and, upon receiving a signal from worker processes, collects results and allocate a new job to the worker. All the rest processes would be the worker which is responsible for all the computation. Once a worker finishes, it sends its results to the master process. In this way, we try to keep workers busy.

It would be important that how large a job is. In our work, we call this parameter block size, which is the number of candidate nodes sent to a worker each time. We vary block size from an experiment to another, but it would be fixed in one experiment. That is, each time a worker would receive a job with fixed number of candidate nodes.

The pseudocode for the master is as follows:

```
while (current seed size < seed size required){
    Non-blocking send seed to all workers;
    Non-blocking send initial block size to all workers;
    while(calculated vertices<total vertices){
        Non-blocking receive results from workers;
        Non-blocking send new block to workers;
    }
    Blocking send telling all workers that the calculation is done;
    Find out the vertex with max influence and add it to seed;
}
```

Code 3

The pseudocode for a worker is:

```
while (true){
    Blocking receive seed;
    if (MPI_TAG == 0) break;
    while (true){
        Blocking receive the block from master;
        if (MPI_TAG == 0) break;
        Calculate average influence of vertices in the block;
        Blocking send the result back to master;
    }
}
```

Code 4

5.3 Graph Partitioning

While each process in the previous two approaches stores the whole graph, we adopt one more approach, Graph Partitioning, where each process owns only a portion of the entire graph in its memory. More specifically, we evenly distribute vertices to different processes. An edge *e = (u, v)* would be stored in the process to which vertex *u* is assigned. In other words, a process owns a subset of vertices and outgoing edges from these vertices.

When starting to calculate the influence, the seed set might be located in different processes. Every process maintains a queue of vertices to start from. At the beginning, all processes add seeds belong to them into the queue. Each process then gets one vertex from the queue and activate other vertices by looking up the outgoing edges. If a vertex to be activated is also owned by the process itself, the activation would be done internally and immediately.

Otherwise, the vertex to be activated would be pushed into a set. After all activation the process could do is done, it sends messages to other processes to activate vertices in the set. Also, the process receives messages from other processes and push into their queue for further simulation. This “activate-communicate” procedure would be repeated for multiple rounds, until no processes sends messages any more.

```
while (current seed size < seed size required){
    for (all vertex v : V){
        Push all vertices in seeds into a queue for starting simulation;
        while (flag){
            Takes one vertex from the queue and begin simulation;
            if (target vertex to be influenced is in worker's own block)
                Push into the queue;
```

```
                    Mark as infected;
                else
                    Insert into a set to be sent to the worker owns the vertex
                later;
                for (w : all workers){
                    Non-blocking send the set to specific worker,
                sent_num++;
                    Non-blocking send dummy message if set is empty;
                }
                Barrier;
                Allreduce to calculate sent_num;
                if (sent_num>0){
                    Blocking receive message with set or dummy info;
                }
                else receive dummy message;
                Barrier;
            }
        }
        Calculate vertex with max influence;
        Add the vertex into seed;
}
```

Code 5

6 Challenges

Following are the technical challenges we experienced during the implementing and testing stages:

- Initially we were unfamiliar with the functionality of MPI non-blocking send/receive, therefore we spent unexpected amount of time fixing synchronization issues within the Dynamic Mapping Implementation.
- In order to implement the Graph Partitioning Implementation, we needed to come up with a proper way to partition the graph.
- We used the USC HPC platform for debugging and testing. As the end of the semester approached, the waiting time for resources allocation became longer. We spent a considerable amount of work time waiting for the allocation of computing nodes.

7 Experimental Setup

7.1 Data Set

The data set that would be used for testing was modified from a Bitcoin OTC trust network example from the Stanford Large Network Dataset Collection[3, 4]. The raw dataset[5] was a weighted signed directed temporal network of transaction ratings among users, constructed from the data from the online bitcoin trade platform Bitcoin OTC. On this platform, a buyer and a seller could rate each other after they made a transaction, in a range from -10 to + 10. The higher the rating, the more trustworthy that the rater believed in the ratee. In this network, each vertex represented a user. Each edge represented a rating, pointing from the rater(source) to the ratee(target). The edge weight represented the rating score, in a range from - 10 to + 10. This dataset also included the timestamp of the transactions. The network was stored in a csv file as the following picture shows:

```
source,target,rate,timestamp
6,2,4,1289241911.72836
6,5,2,1289241941.53378
1,15,1,1289243140.39049
```

Figure 2

For example, the above picture shown three transactions ratings in the network:

User #6 gaveUser #2 a rating score of 4
User #6 gave User #5 a rating score of 2
User #1 gave User #15 a rating score of 1

This network contained 5,881 vertices(users) and 35,592 directed edges(transaction ratings), and 89% of the edges carried positive weights.

In order to apply this network into our project, we needed to process the raw dataset into the format that our project desired. First, the timestamp field was eliminated after we noticed that there were no parallel directed edges in the graph, which meant there did not exist any duplicated ratings from the same rater to the same rater.

To use the edge weights as the probability factor in Kempe's algorithm, the ratings were shifted and normalized from ±10 to [0, 1]. The processing of the dataset was completed using a Python3 script and the Python Data Analysis Library(PANDAS). All the vertices and edges from the raw dataset were retained.

The following picture shown the format of the processed dataset that would be used for testing:

```
source,target,weight
1,2,0.9
1,3,0.8
```

Figure 3

Using the Python NetworkX library, we visualized the processed dataset into a network graph for better understanding. The result is shown below as follows:

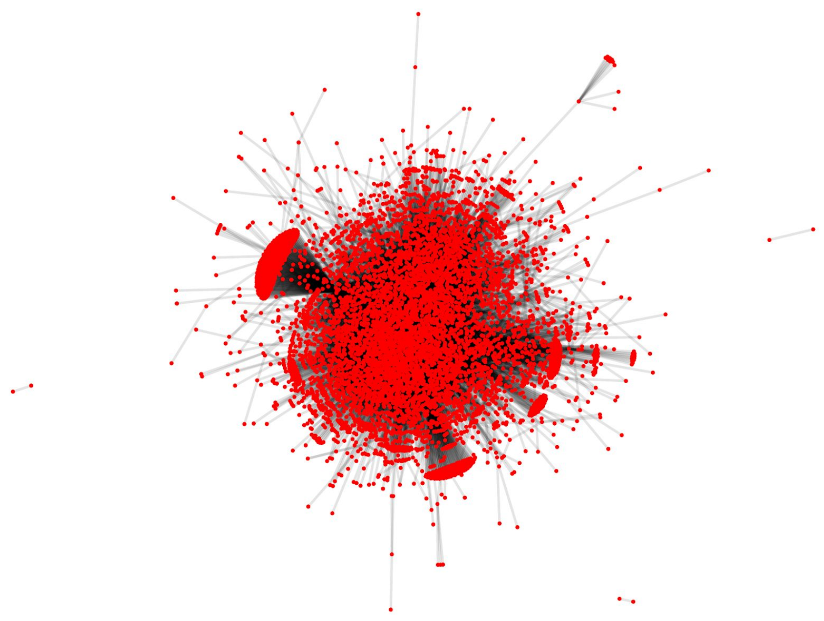

Figure 4

From the above picture, we can observe that most of the vertices in the network were well connected, while 3 isolated small groups of 2 vertices did exist. We also used the Python Matplotlib Library to visualize the distribution of the edge weights. Most of the edge weights were between 0.5 and 0.7, as is shown below:

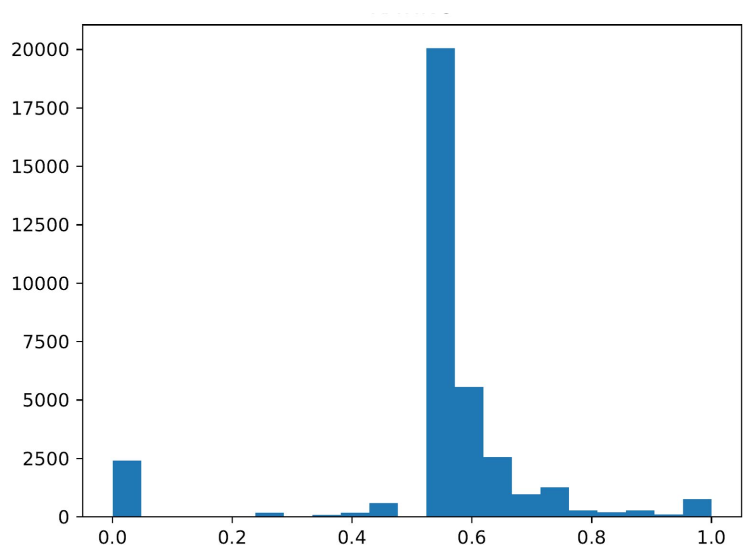

Figure 5

7.2 Testing Parameters

For testing purposes, we chose to use variants of the four following parameters, can compare the average total run time among different parameter values:

1. Seed Size(input parameter)

This is an input parameter to Kempe's algorithm. It decides how many vertices that the algorithm should pick and return. In other words, it decides how many "influencers" that we want to choose in order to maximize influence. Larger seed size would provide a better influence maximization result, but also increases the total run time.

2. Sample Times(input parameter)

This is also an input parameter to Kempe's algorithm. Since probability is used to decide whether the influencing steps are successful or not, each trial would return a different result. The algorithm averages the max influence of multiple trials(samples), and picks desired vertices based on the average max influence of each vertex. Therefore increasing sample times would improve the accuracy of influence maximization, but also increase the total run time.

3. Block Size in the Dynamic Mapping Implementation

This is a parallelization parameter that can be adjusted in the Dynamic Mapping Implementation. Block size refers to the number of vertices that each time the master process sends to the worker processes. A smaller block size would lead to more balanced workload, since the vertices are divided and distributed in a finer pattern, but also increases the amount of communication between the master and the workers.

4. Number of Workers in the Static Mapping Implementation

This is a parallelization parameter that can be adjusted in the Static Mapping Implementations. It decides how many parallel computing resources are allocated and used for testing. Due to the limit of the USC HPC platform, we would use 2, 3, 4 workers to test the Static Mapping Implementation.

7.3 Testing Platform

The testing of our serial and parallel implementations would be done on the USC High-Performance Computing Platform(HPC). One default computing node would be requested and allocated to test the serial implementation. Two or four default computing nodes would be requested and allocated to test the parallel implementations. We would use the same method to request and allocate computing nodes as we were instructed to use in the programming assignments.

8. Results

Test 1 : Average Time with Different Sample Times

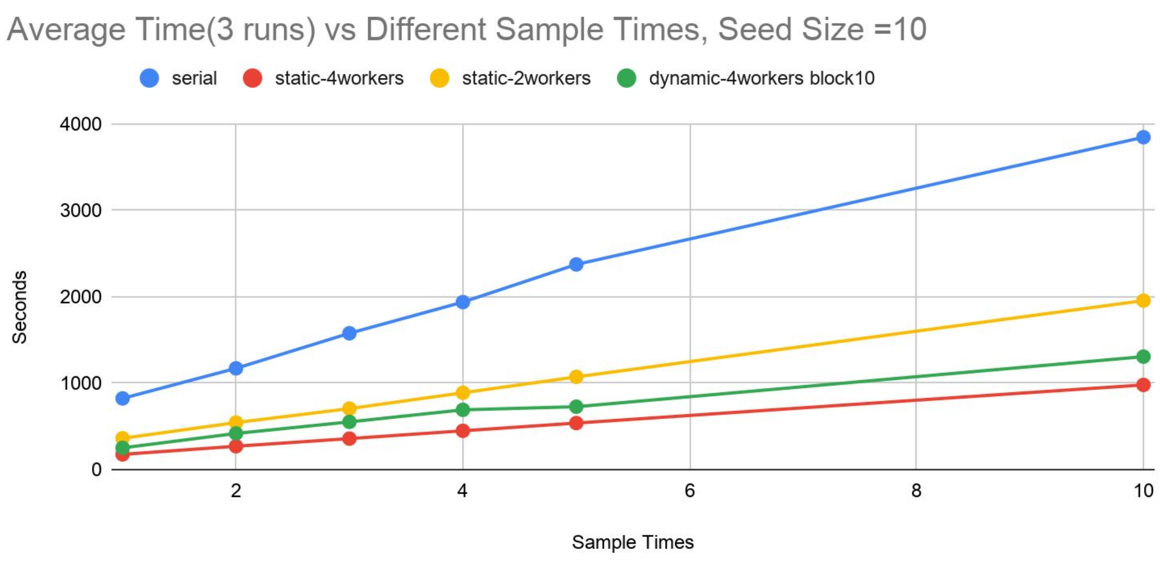

Figure 6

sample times	serial	*static-4workers*	*speedup*	*static-2workers*	*speedup*	dynamic 4 workers block size 10	*speedup*
1	824.061154	*176.641695*	*4.665156514*	*362.5094*	*2.273213202*	251.898471	*3.271401969*
2	1171.89231	*270.639537*	*4.330085408*	*544.571605*	*2.151952653*	418.678847	*2.799024404*
3	1577.608698	*359.085424*	*4.393407787*	*705.921789*	*2.234820801*	552.289395	*2.856489211*
4	1938.024556	*449.179626*	*4.314586958*	*889.004159*	*2.179994926*	691.930700	*2.800894014*
5	2373.621879	*538.699705*	*4.406206012*	*1072.90806*	*2.212325517*	727.87309	*3.261038101*
10	3845.355369	*980.049874*	*3.923632328*	*1955.24439*	*1.966687842*	1307.113073	*2.941868954*
Average Speedup			***4.338845834***		***2.16983249***		***2.988452775***

Table 1

In the first test, we ran each implementation 3 times, with different sample times input parameter. According to Kempe's Influence Maximization Algorithm, increasing the same times can improve the accuracy of the simulation. However, it would also increase the amount of computation. From the GraphX and TableX, we can see that the run time increased near-linearly as the sample times increased in all 4 implementations. As expected, the serial implementation had the slowest performance. The Static Mapping Implementation with 4 workers had an average ~4.34X speedup, and had an average ~2.17X speed with 2 workers. The Dynamic Mapping Implementation with 4 workers had an average 2.99X speedup.

Test 2: Average Time with Different Seed Sizes

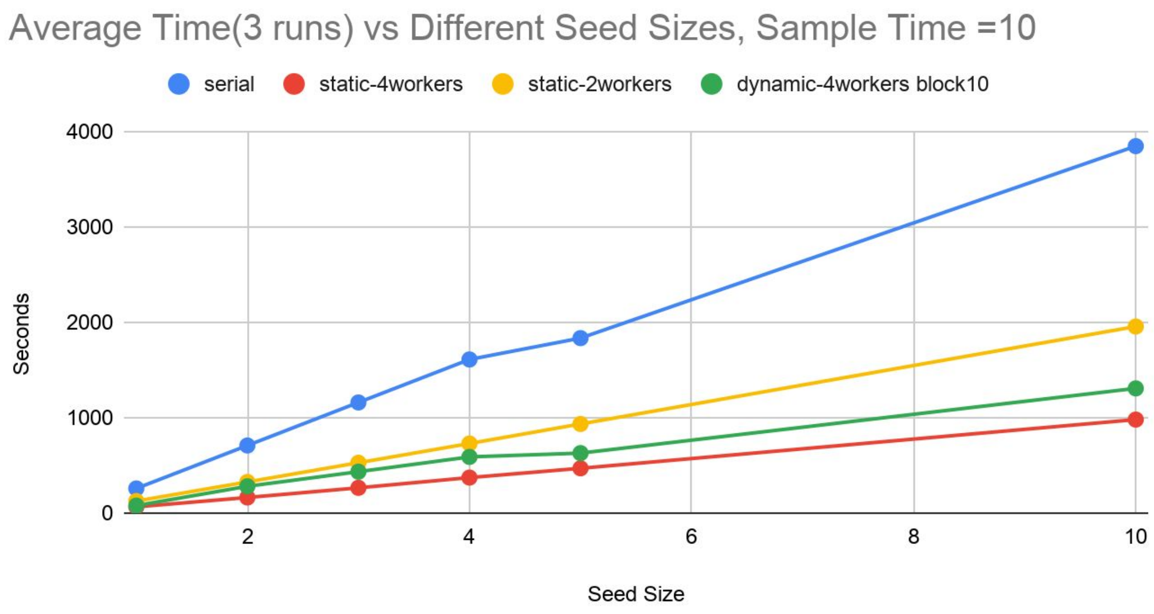

Figure 7

seed size	serial	static 4workers	*Speedup*	static-2workers	*Speedup*	dynamic 4workers block10	*Speedup*
1	258.507832	65.547084	*3.943849462*	126.625421	*2.041516071*	78.871524	*3.277581298*
2	708.69632	163.849988	*4.325275385*	326.75538	*2.168889522*	281.161104	*2.520605837*
3	1159.551057	265.161307	*4.373002495*	528.015159	*2.196056377*	434.170382	*2.670728141*
4	1610.088239	372.456771	*4.32288621*	729.428886	*2.207327225*	589.029946	*2.733457356*
5	1833.843281	469.77452	*3.903666978*	934.22515	*1.962956447*	628.780562	*2.916507589*
10	3845.355369	980.049874	*3.923632328*	1955.24439	*1.966687842*	1307.113073	*2.941868954*
Average Speedup			**4.132052143**		**2.090572247**		***2.843458196***

Table 2

In the second test, we ran each implementation 3 times, with different seed size input parameter. In Kempe's Influence Maximization Algorithm, the number of seed size decides how many vertices should be picked and returned. In other words, it decides how many "influencers" that the algorithm should choose to maximize influence. Larger seed size would provide a better influence maximization result, but also increases the total run time. From the GraphX and TableX, we can see that the run time also increased near-linearly as the seed size increased in all 4 implementations. As expected, the serial implementation had the slowest performance. The Static Mapping Implementation with 4 workers had an average ~4.13X speedup, and had an average ~2.09X speed with 2 workers. The Dynamic Mapping Implementation with 4 workers had an average ~2.84X speedup.

Test 3: Average Time with Same Seed Size and Sample Times

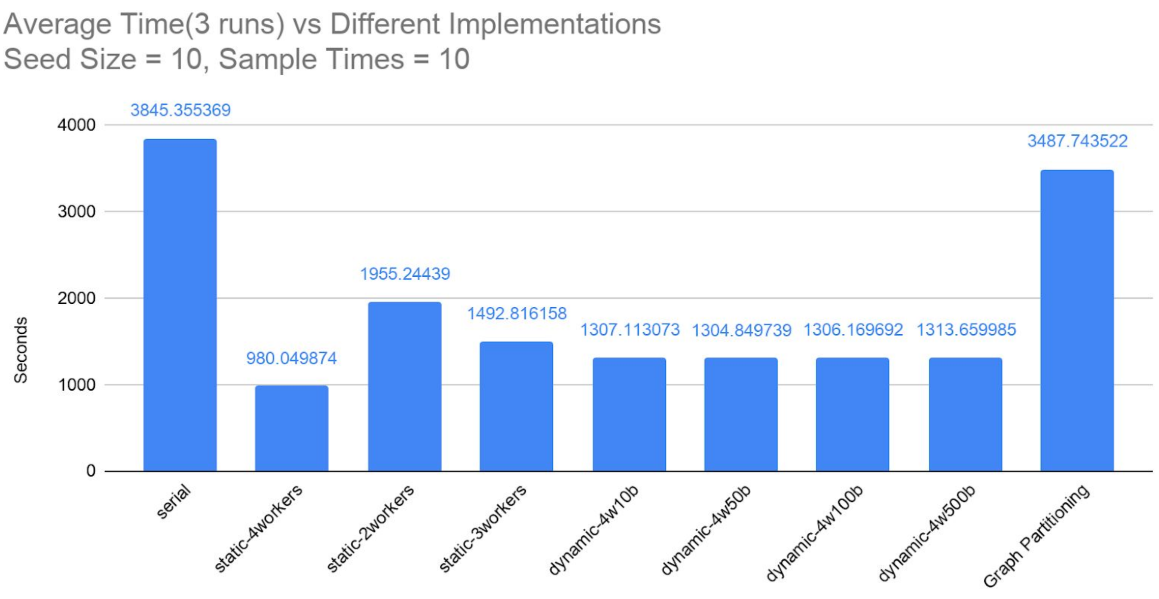

Figure 8

In test3, we compare and average run time of each implementation with the same seed size and sample times. For the Dynamic Mapping Implementation, we also tested it with 4 workers and 4 different block sizes(10, 50, 100, 500) to see the effects of workload balancing.

From the figure above, we can see that the Static Mapping Implementation with 4 workers had an average ~3.92X speedup, an average ~2.58X speedup with 3 workers, and an average ~1.97X speedup with 2 workers. The Dynamic Implementation with 4 different block size all returned close results with an average ~2.94X speedup.

Our Graph Partitioning Implementation is completed, but the performance is not ideal, and not close to what we had expected. For seed size = 10, and sample times = 10, it took 3487.743522 seconds to finish. The speedup was only ~1.1X.

9. Analysis

From the above testing results, we have the following observation:

1. Both Static Mapping and Dynamic Mapping are faster than the serial version, with the same seed size and sample times
   - The Static Mapping Implementation with 4 workers had ~3.92 average speedup
   - The Static Mapping Implementation with 3 workers had ~2.58 average speedup
   - The Static Mapping Implementation with 2 workers had ~1.97 average speedup
   - The Dynamic Mapping Implementation with 4 workers had ~2.94 Speedup

2. Dynamic Mapping Implementation was actually slower than Static Mapping implementation with the same number of workers.

Given the same number of workers, the Dynamic Mapping Implementation had to assign one process to be the master who was responsible for workload balancing. As a result, this worker

process was not involved in computation, therefore this implementation lost 1/P compute resources.
But if we compare the Static Mapping Implementation w/ 3 workers with the Dynamic Mapping Implementation w/ 4 workers, as both cases have the same amount of workers(3) that were doing computation, we can see that the speedup goes up from 2.58 to 2.94.

3. Dynamic Mapping Implementation with different block size had similar run time
From the testing results, we noticed that the Dynamic Mapping Implementation with 4 different block size(10, 50, 100, 500) in fact had close run time. The time difference was so small and ignorable, since the difference was in the range of error margin. Contrary to our hypothesis, the communication cost did not have any noticeable effect on the average run time.

Possible causes:

- As graphX shows, the testing dataset only had 3 isolated small groups. Most of the vertices were interconnected in the main group. Hence, the computation workload among each starting vertex was not very unbalanced.
- The testing dataset was not large enough to show the performance difference.

4. Graph Partitioning Implementation only had a speed up of ~1.1X
We used a small test dataset to verify the implementation, and it did return the correct influence maximization result. However, the performance was not ideal, as it was slower than Static Mapping and Dynamic Mapping Implementations. The reason is that the algorithm spends too much time synchronizing. Each time a process finishes activation in its own subgraph, it remains idle for a long time. We conjecture that Graph Partitioning might have better performance on a larger dataset, or we need better ways to handle the synchronization issues.

10. Conclusion

Before the project we did not have much experience implementing parallelization, especially the MPI we use in our program. This hands-on experience provides us a great opportunity to better understand how blocking, non-blocking and barrier should be applied, as well as the Master-worker model we implemented in our dynamic mapping version. We also get more familiar with other features of MPI such as MPI_Probe and MPI_Waitany.

The speed up we get from static and dynamic mapping is fairly good. However the dynamic mapping does not improved much from the static mapping, which is different from our expectations. We figured out some possible explanations, such as the graph is too dense that vertices all get influence to large number of vertices, leading to not much imbalance between simulations. The graph-partitioning approach did make memory space requirement reduced and decreases the memory access time, however the graph we choose may not be big enough to see the expected speed up. We can still make some improvements on the way we partition the graph.

In all, it is a great experience, from choosing topic, implementing serial version, improving runtime through different parallelization approaches and analysing with different parameters.

Reference

[1] Kempe D, Kleinberg J, Tardos É. Maximizing the spread of influence through a social network[C]//Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003: 137-146.
http://theoryofcomputing.org/articles/v011a004/v011a004.pdf

[2] Banerjee, S., Jenamani, M. and Pratihar, D. (2019). *A Survey on Influence Maximization in a Social Network*. [online] arXiv.org. Available at: https://arxiv.org/abs/1808.05502 [Accessed 25 Oct. 2019]. https://arxiv.org/pdf/1808.05502.pdf

[3] S. Kumar, F. Spezzano, V.S. Subrahmanian, C. Faloutsos. Edge Weight Prediction in Weighted Signed Networks. IEEE International Conference on Data Mining (ICDM), 2016.S.

[4] Kumar, B. Hooi, D. Makhija, M. Kumar, V.S. Subrahmanian, C. Faloutsos. REV2: Fraudulent User Prediction in Rating Platforms. 11th ACM International Conference on Web Search and Data Mining (WSDM), 2018**.**

[5] Snap.stanford.edu. (2019). SNAP: Signed network datasets: Bitcoin OTC web of trust network. [online] Available at: http://snap.stanford.edu/data/soc-sign-bitcoin-otc.html [Accessed 25 Oct. 2019]. http://snap.stanford.edu/data/soc-sign-bitcoin-otc.html