15-618 Parallel Computer Architecture and Programming

Parallel Influence Maximization in Social Networks

Zican Yang (zicany)   Yuling Wu (yulingw)

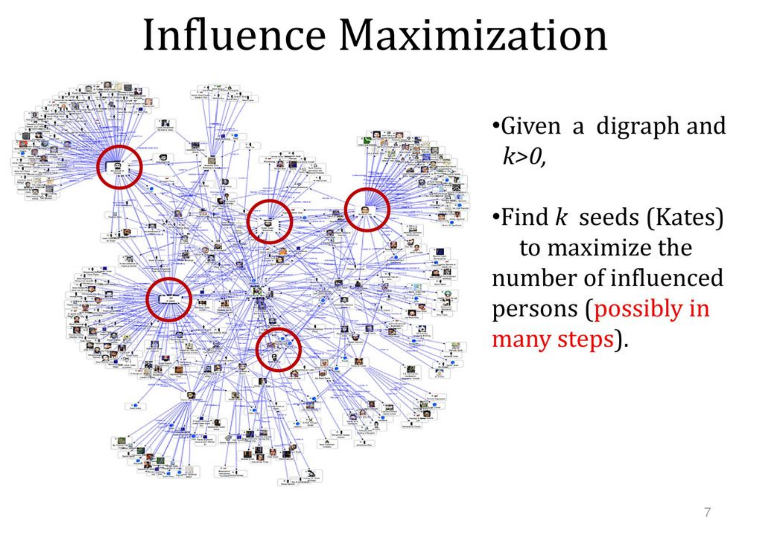
Summary

We implemented a parallel social network influence maximization algorithm on both greedy and heuristic settings with OpenMP. We implemented different kinds of parallelism and verified their performance to explain the best approach in different situations.

Background

The user relationships on many platforms can be represented by social networks. In the fields of public opinion monitoring and viral marketing, the goal is to find a fixed number of seed user nodes to maximize the spread of influence. Such goals can be abstracted as Influence Maximization, shown in the following graph[5].

We choose the independent cascade(IC) model. Suppose we have a directed Graph G=(V, E, W), where each vertex in V is a social network user, each edge in E is an influence spread route from one user to another, and each value in W is the weight of an edge which represents the probability of influence spread. The influence maximization problem can be depicted as, given the number N, selecting N seed nodes from V which will result in the maximum number of activated nodes.



Influence Maximization

•Given a digraph and *k>0,*

•Find *k* seeds (Kates) to maximize the number of influenced persons (possibly in many steps).

7

We will make every assumption simpler to focus on parallelism. We suppose our graph is an undirected graph, and each propagation probability is identical. We also suppose each vertex can only be visited once. This problem is NP-hard[3], and there are two kinds of mainstream algorithms to solve it. One is greedy algorithms[3], and another is heuristic algorithms[4].

Monte Carlo simulation is an important part. Given a seed node set S and rounds N, we can evaluate the spread result of S by performing Monte Carlo simulation N times on the graph. In each round, we start with S as activated nodes, and recursively activate nodes until stopped. The average number of activated nodes in N rounds is the simulation result. A larger Monte Carlo simulation in a graph means a better-chosen seed set. The larger number of rounds in the Monte Carlo simulation, the more precise the evaluation will be.

The Monte Carlo simulation is computationally expensive when given large rounds N, and it can be parallelized in two levels. The first is to parallelize N rounds of Monte Carlo simulation; the second is to parallelize the activation process in a single Monte Carlo simulation.

The basic greedy algorithm is to compute the Monte Carlo simulation result on each possible assignment of N seed nodes on V, and select the set of nodes with the maximum result. The algorithm is computationally expensive because the permutations of seed nodes will be quite large and each permutation will perform a Monte Carlo simulation. This can be parallelized by precomputing all the possible assignments of seed nodes and parallelizing the computation on each possible assignment of seed nodes and combining the result from each local maximum value. Parallelizing the Monte Carlo simulation will also help.

The basic heuristic algorithm is to compute the out-degree of vertices and sort in descending order, then select the nodes with top-N out-degree as seed nodes set. To make the result more precise, we need the minus-1 degree or degree-discount algorithm[4] to modify the out-degrees of neighbors of selected seed nodes to avoid selecting nodes too close to each other. This can be parallelized by parallelizing the selection of N seed nodes. In the minus-1 and degree-discount heuristic, there are dependencies between a selection of seed nodes because the selection of a seed node will change the degrees of its neighbors. Note that the selection of seed nodes in the heuristic algorithm does not involve Monte Carlo simulation. The Monte Carlo simulation is only used for the evaluation of spread results.

Greedy and heuristic algorithms share the same data structures, operations, inputs, and outputs. The key data structure is the graph representing the social network and vertices in the graph. A vertex has a unique ID and a list of neighbors. A graph has a list of vertices and a probability to represent propagation probabilities on each edge. The key operation on the data structures is the spread of influence. When performing Monte Carlo simulations, in each round it is like a breadth-first search with a memo. We keep a visited cache to record which of the vertices has been visited, and do BFS on all seed nodes with a probability until stopped. The output of both algorithms is a seed set and a spread result of the seed set.

The discussion of influence spread proposed by Pedro Domingos and Matt[1][2] shows they are serialization algorithms that need lots of time to finish their calculation. The reason is there are several internal dependencies among the data, parallelization will be a challenging task. Hence, the key challenge point is how do we find a suitable way to eliminate the dependencies and how do we balance the trade-off between correctness and performance.

The Monte Carlo Simulation has internal dependencies in that seed nodes share a visited memo to record which vertices have been visited in the BFS process. If we parallelize the rounds of it, there is no dependency problem and it is data-parallel and amenable to SIMD execution. We only need to use reduction to collect each round's result. If we parallelize the BFS process for the seed nodes per round, there are dependency problems because the parallelized workers share the visited memo.

The greedy algorithm has internal dependencies in that the goal is to choose the seed nodes set which give the best simulation result from all permutations of seed nodes. The greedy algorithm can be parallelized in each permutation's Monte Carlo simulation, or parallelize the computation on all the permutations. The parallelism in computation on all the permutations has dependency issues because each worker shares the maximum value of the simulation result.

The heuristic algorithm only uses Monte Carlo simulation to evaluate the seed nodes selected, the parallelism of simulation is stated before, so we only talk about the parallelism in seed node selection process. The basic heuristic algorithm does not have internal dependencies, but minus-1 and degree-discount algorithms have internal dependencies because a seed node will update its neighbors' degree. We can parallelize it by, in each round, select some seed nodes with highest degrees and parallelly update their neighbors' degree, then repeat until all seed nodes have been selected.

Approach

We have implemented four kinds of parallelism, as shown below. Monte Carlo simulation is used both in greedy and heuristic, and it can be parallelized on two levels. Greedy has its own parallel method, and so does heuristic.

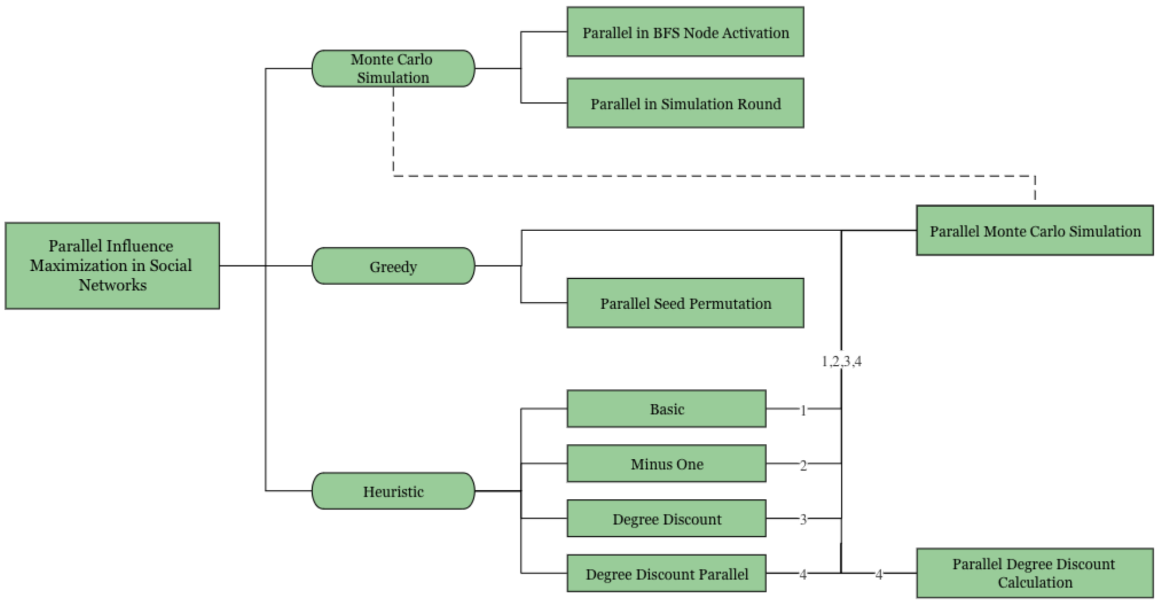

Parallel Influence Maximization in Social Networks
Monte Carlo Simulation
Parallel in BFS Node Activation
Parallel in Simulation Round
Greedy
Parallel Monte Carlo Simulation
Parallel Seed Permutation
1,2,3,4
Heuristic
Basic
1
Minus One
2
Degree Discount
3
Degree Discount Parallel
4
4
Parallel Degree Discount Calculation

Figure. Framework of Parallel Design and Implementation

We used OpenMP as well as OpenMP Locks to parallelize the algorithm. The programming languages we used are C++ and Java. C++ is responsible for the algorithm and Java is responsible for generating test datasets for greedy algorithms. We used GHC Machine with 8 cores Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz.

Since the machine has 8 cores, we mainly used OpenMP parallel (for) to parallelize the parallelizable parts and map each OpenMP thread to a different core. However, we have designed and implemented the parallel version in different dimensions and the whole framework is shown in the figure above.

Monte Carlo Simulation Parallelism

Approach 1: Since Monte Carlo Simulation will perform N rounds simulation with the seed node set S, we perform the parallel calculation by dividing the N rounds into N / nThread cycles. Specifically, each thread in one cycle will take the responsibility of one round's calculation. Each thread will hold a local sum of all its rounds' simulation results, and threads have to be reductively summed together to get the total simulation result. There is no other dependencies between threads.

Approach 2: In a single simulation round, all vertices in seed node set S will perform BFS. We can parallelize the BFS process of each seed node, but each vertex in the graph can only be visited once and this results in dependency between threads. Hence, we introduced a series of locks to protect a vertex when visited. Multiple BFS will share the same visited array with vertex locks and update the array simultaneously.

Greedy Parallelism

Approach 1: For each possible assignment S of seed nodes on all vertices, use parallel Monte Carlo simulation as shown in Monte Carlo Simulation Parallelism to accelerate the process.

Approach 2: Instead of parallelizing Monte Carlo simulation, we can precompute all the possible permutations of seed node set S, then compute each permutation's simulation result in parallel. In other words, each thread is responsible for part of the possible seed node sets.

Heuristic

The heuristic algorithm can be divided into two parts. The first part is heuristic seed set selection, second is the evaluation of selected seed nodes by Monte Carlo simulation. Note that only seed node selection is the real computation of the algorithm, so our main goal is to parallelize the first part.

For the Basic heuristic, there are no dependencies between a selection of seed nodes because selecting a seed node will change nothing to its neighbors. For Minus-1 and Degree-Discount heuristic, there are dependencies between a selection of seed nodes because selecting a seed node will change its neighbors’ degree. The algorithm is intrinsically serial, in that there are no such data structures to select a data with specific rank (For example, it is impossible to select vertices with top |S| degrees simultaneously), and for Minus-1 and Degree-Discount, selecting a vertex will change all its neighbors’ degree, which means we have to sort the vertices with its degree again.

To parallelize this, a tradeoff is needed. Because Degree-Discount is the best of these three, we will parallelize the Degree-Discount heuristic. If we have S seed nodes and T threads, the idea is like "take blocks every time". Imagine we have a data structure that maps vertex id to its degree, and the map is automatically sorted descendingly by its value. Suppose the data structure can support directly getting its key-value pair with a specific rank. Then we can take T vertices each time, add them to the seed set, and update their neighbors’ degrees in parallel. We repeat this process until S seed nodes have been selected. There is a tradeoff between spread result and speed, because we may select worse seed nodes. Every time we choose T vertices, the effect of degree-discount is ignored among them.

However, there is no such data structure in C++. We have found another way to do this. The data structures we use are an int-int map id2degree and an int-vector<int> map degree2ids. id2degree holds every node's degree, and degree2ids holds every degree's corresponding vertex ids. Because C++'s map is sorted by its key, degree2ids can let us get vertices with the highest degree, but are not parallelizable.

The step is as follows:

1. Get T vertices with top T degrees from degree2ids in serial and add them to seed nodes set, delete these vertices from id2degree and degree2ids

2. Update every selected seed nodes' neighbors' degree by the degree-discount formula in parallel, update in id2degree and degree2ids

3. Repeat steps 1 and 2 until select S seed nodes

The parallelism in getting top T vertices cannot be implemented, so the parallelism is only in step 2. We can update this round's selected seed nodes' neighbors' in parallel. Note that we have to use locks in id2degree and degree2ids to prevent contention.

For the evaluation of selected seed nodes, we can use parallel Monte Carlo simulation as shown in Monte Carlo Simulation Parallelism to accelerate the process.

We changed the original serial algorithm to enable better mapping to a parallel machine as follows.

Greedy Permutation: Since there are a great number of combinations from all permutations which cause high overhead, instead of doing the simulation one by one like the original serial version, we separated the combinations into several blocks, each block's size is equal to the multiple of Threads. Therefore, a block can be calculated simultaneously by several parallel threads.

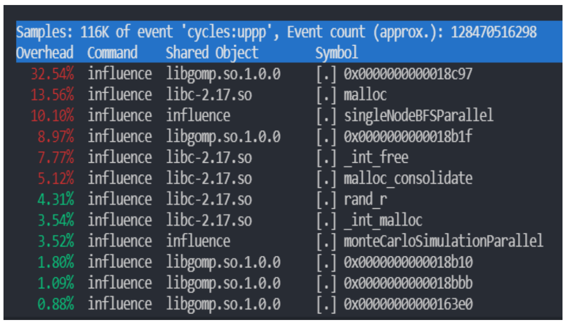
Parallel Degree-Discount Heuristic: In order to implement the parallel Degree-Discount Heuristic algorithm, we need to pre-assign OpenMP locks to each possible degree in the degree2ids map, or there would be a core dump when we tried to assign OpenMP locks in parallel threads. We simplified the calculation of the new degree by setting it to zero if the new degree is equal to or lower than zero. This approach may lose some precision, but it will lead to performance improvement.

Issues and Solutions

- Random Integer Performance in Parallel
  Initially, we used rand() to generate random numbers in our parallel code. However, we found that generating random numbers took over 50 percent of the runtime, and sometimes the random numbers are the same. So we replaced the rand() function with the rand_r(int* seed) function which is designed for parallel use by different seeds.

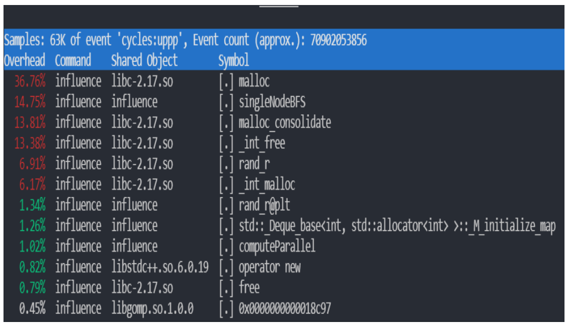
- OpenMP Overhead in Greedy Algorithm
  At the very beginning, we chose to parallelize the Monte Carlo simulation in our Greedy algorithm. Then we found that the OpenMP overhead took over 40% of the time. The reason was if we parallel in the Monte Carlo simulation, then in every possible seed set, we need to invoke and release N iterations of OpenMP threads, the total number of OpenMP operations was

N*PERMUTATIONS. Hence, we changed our code to first calculated all possible permutations, then parallel in blocks of permutations, and performed the Monte Carlo simulation in serial.



```
Samples: 116K of event 'cycles:uppp', Event count (approx.): 128470516298
Overhead  Command    Shared Object        Symbol
  32.54%  influence  libgomp.so.1.0.0     [.] 0x0000000000018c97
  13.56%  influence  libc-2.17.so         [.] malloc
  10.10%  influence  influence            [.] singleNodeBFSParallel
   8.97%  influence  libgomp.so.1.0.0     [.] 0x0000000000018b1f
   7.77%  influence  libc-2.17.so         [.] _int_free
   5.12%  influence  libc-2.17.so         [.] malloc_consolidate
   4.31%  influence  libc-2.17.so         [.] rand_r
   3.54%  influence  libc-2.17.so         [.] _int_malloc
   3.52%  influence  influence            [.] monteCarloSimulationParallel
   1.80%  influence  libgomp.so.1.0.0     [.] 0x0000000000018b10
   1.09%  influence  libgomp.so.1.0.0     [.] 0x0000000000018bbb
   0.88%  influence  libgomp.so.1.0.0     [.] 0x00000000000163e0
```

Before



```
Samples: 63K of event 'cycles:uppp', Event count (approx.): 70902053856
Overhead  Command    Shared Object        Symbol
  36.76%  influence  libc-2.17.so         [.] malloc
  14.75%  influence  influence            [.] singleNodeBFS
  13.81%  influence  libc-2.17.so         [.] malloc_consolidate
  13.38%  influence  libc-2.17.so         [.] _int_free
   6.91%  influence  libc-2.17.so         [.] rand_r
   6.17%  influence  libc-2.17.so         [.] _int_malloc
   1.34%  influence  influence            [.] rand_r@plt
   1.26%  influence  influence            [.] std::_Deque_base<int, std::allocator<int> >::_M_initialize_map
   1.02%  influence  influence            [.] computeParallel
   0.82%  influence  libstdc++.so.6.0.19  [.] operator new
   0.79%  influence  libc-2.17.so         [.] free
   0.45%  influence  libgomp.so.1.0.0     [.] 0x0000000000018c97
```

After

- OpenMP Malloc Overhead

  From the performance figure above, malloc took up 36.76% of the total time. We first tried to reduce the amount of memory that we allocated to store the permutations by separating the permutations into blocks. However, the malloc overhead was only reduced a little. Then we found that the overhead came from the *OpenMP Parallel* call, hence, we used the *OpenMP Parallel for* to invoke the parallel threads which successfully reduced the overhead to 26.77%.

- Heuristic OpenMP Lock Initialization Core Dump

  When we were trying to allocate memory for OpenMP locks as needed in threads, the core dump always occurred because OpenMP does not allow its threads to initialize locks. Therefore, we predefined all possible locks and assigned them to the process maps before the parallel part.

Results

Performance Measurement

Dimension 1: We used the high_resolution_clock to record the computing duration time. The less time it takes, the better the performance is.

Dimension 2: Average propagation result is recorded to evaluate the selected seed nodes. The more vertices that can be activated, the better the selected seed set is.

Experimental Setup

Input: Except for the datasets below, we also provided a Data Generator which is *BuildData.java* to help people generate any kind of social networks they want. We use easy.txt to test greedy algorithms and use the other two datasets to test heuristic algorithms. The reason is greedy algorithms take a long time to run on a large dataset, and heuristic algorithms take a short time to run on a small dataset.

Dataset Name	easy.txt	facebook_combined.txt	hep.txt
Number of Vertices	40	4039	15233
Number of Edges	600	88234	58891

Generate Request: We provided a helper for the program, use `./influence -h` will print the information below.

```
Parameters:
	<-f FILE_PATH> Test file path
	[-p PROBABILITY] Propagation probability, default 0.1
	[-i MONTE_CARLO_SIMULATIONS] Iterations for monte carlo simulation, default 100
	[-s # of SEEDS] Number of initial seeds, default 10
	[-t IS_GREEDY] Use greedy method, default 1
	[-m PARALLEL_MODE] Use parallel mode, default 1
	[-n # of THREADS] Number of threads, default 1
	[-x HEURISTIC_MODE] Heuristic approach, 0 for BASIC, 1 for MINUS_ONE, 2 for
DEGREE_DISCOUNT, 3 for DEGREE_DISCOUNT_PARALLEL, default 2
	[-l WITH_LOCK] Lock implementation in heuristic approach, default 0
	[-h]: Print helper

Usage:
	./influence -f <filename> [-p <PROBABILITY>] [-i
<MONTE_CARLO_SIMULATIONS>][-s <# SEEDS>] [-t <IS_GREEDY>][-m
<PARALLEL_MODE>] [-n <# THREADS>][-x <HEURISTIC_MODE>] [-l <WITH_LOCK>]
[-h]
```

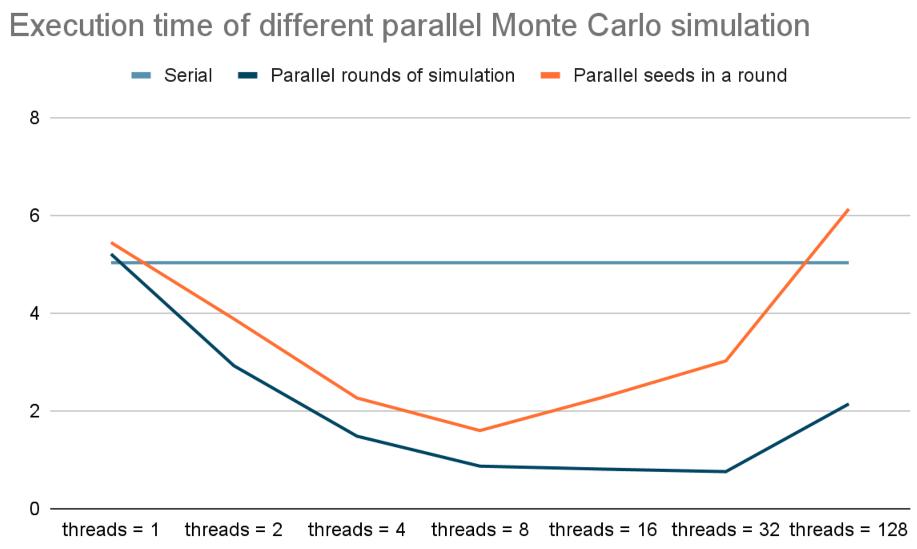
Project Result

Then we test our implementations on different settings. This includes a test of two kinds of parallel Monte Carlo Simulation, a test on parallel greedy and a test on parallel heuristic seed node selection.

Monte Carlo Simulation

First we test the two different parallel versions of the Monte Carlo simulation itself. To test it, we choose to run the heuristic algorithm because, in heuristic settings, the Monte Carlo simulation takes much of the program execution time.

To start with, we run the basic heuristic on facebook_combined.txt with Monte Carlo simulation rounds = 10000, number of seed nodes = 1000, propagation probability = 0.01, on serial and parallel implementations. We also change the number of threads in parallel implementation. The result is as below. Because the parallel Monte Carlo simulation does not decrease the spread result, we will only test the execution time. The baseline serial implementation is single-threaded CPU code.



From the graph, we can observe that the parallel version of the Monte Carlo simulation can give a reasonable speed up. The parallel rounds of simulation have a better speedup compared to the parallel seeds in one round because the second has a lot of lock operations.

The computation inside the Monte Carlo simulation is like:

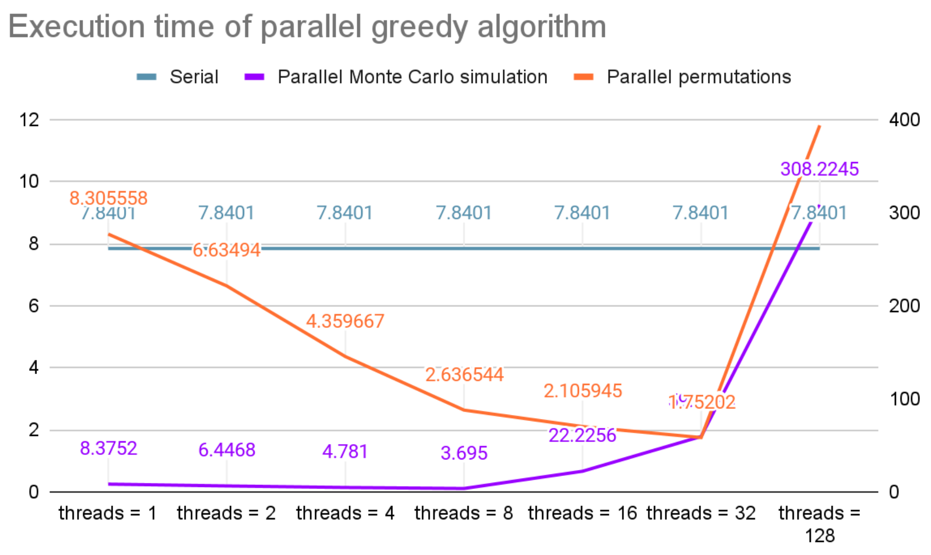
```
foreach (rounds) {
      foreach (vertex) {
            BFS(vertex)
      }
}
```

Talking about scale. The parallel rounds of simulation parallelize the first foreach, and the parallel seeds in one round parallelize the second foreach. With a large number of rounds, we tested that parallel rounds of simulation are better than parallel seeds in one round. The number of rounds scales because each round is an independent computation process. We thought that if we have small rounds and larger numbers of seeds, the parallel seeds in one round will be better than parallel rounds of simulation. However, we tested on hep.txt dataset with number of seeds = 10000, probability = 0.01, number of rounds = 10, and the parallel rounds of simulation still behaved better than parallel seeds in one round. The reason is that a larger number of seeds will result in more locking and unlocking overhead, and it does not scale well because seeds with a small degree will likely be visited before.

Greedy

There are two kinds of parallelism in parallel greedy algorithms. The first is parallelizing the Monte Carlo simulation in each permutation's computation of spread result, the second is parallelizing the computation of permutations (Each thread will be responsible for part of all permutations). Because the parallelism will not decrease the spread result, we will only test execution time.

We tested the greedy algorithm on the easy.txt dataset, with number of seeds = 5, propagation probability = 0.01, number of Monte Carlo simulations = 10. The result is as follows. The parallel permutations give a better speedup. The reason is that every possible permutation will call Monte Carlo simulation once, and if we parallelize the Monte Carlo simulation, the OpenMP overhead will be quite high. Precomputing all possible permutations and assigning part of permutations to each thread will result in lower OpenMP overhead.

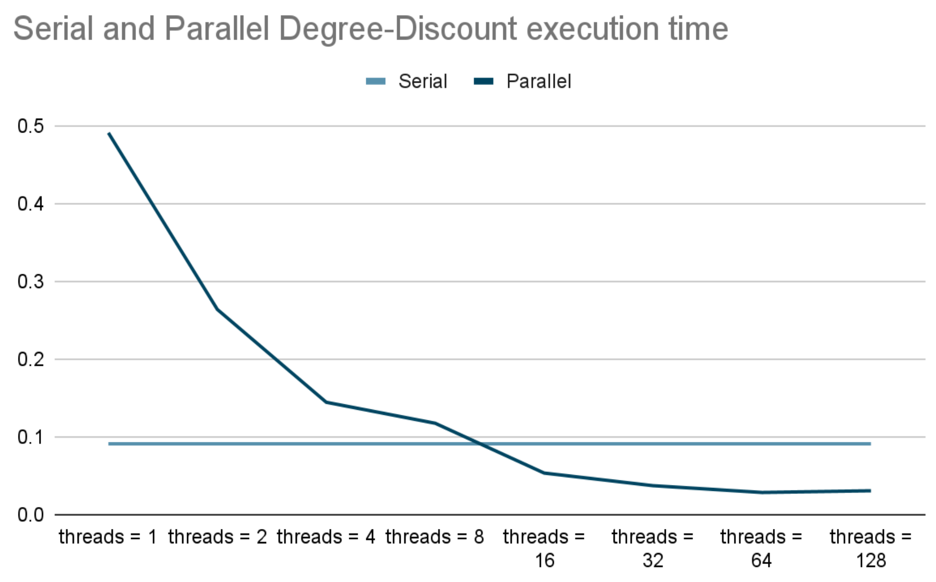


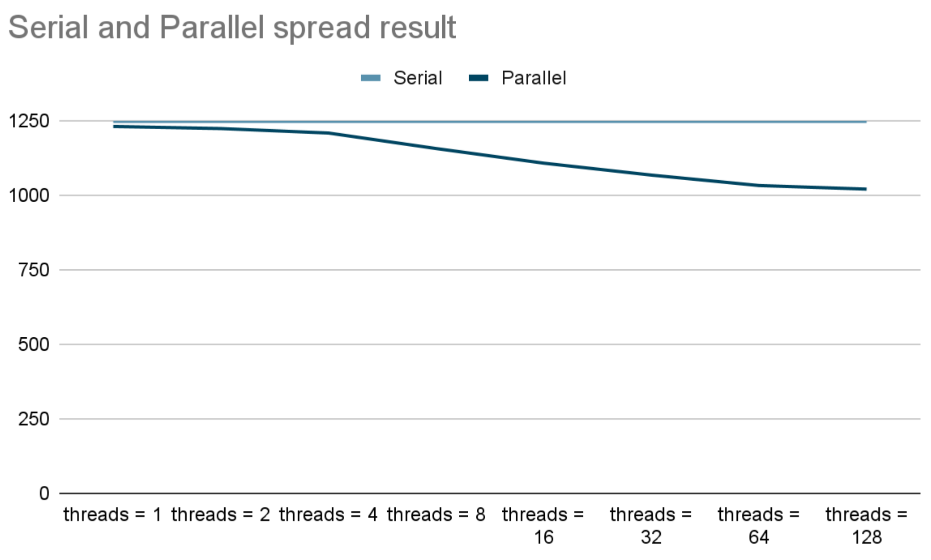
Heuristic

Because we mainly focus on the seed node selection process of heuristic algorithms, and there is a tradeoff within parallelism, we will test parallel and serial versions of Degree-Discount heuristic algorithms, and compare their spread result and execution time.

The heuristic algorithm will first select seed nodes and evaluate the result by Monte Carlo simulation. To make our test more precise, we will first run on two kinds of Degree-Discount heuristics without running a simulation to get only the execution time of the selection process. Then we will run a very large number of simulation rounds to get only the spread result.

We performed the test on the facebook_combined.txt dataset with number of seed nodes = 1000, propagation probability = 0.01. The results are as follows. We can observe that the larger number of threads will make the parallel Degree-Discount algorithm run faster, and as the number of threads increases, the overhead of parallelism will affect the speedup. There is an obvious tradeoff in that the larger number of threads will make the spread result worse because the probability of selecting the wrong seed nodes increases.





The Importance of Problem Size

As mentioned above, greedy algorithms take a very long time to run on a large dataset, while heuristic algorithms take a very short time to run on a small dataset. Therefore, we compared these two algorithms' performance on different sizes of datasets.

One foreseeable result about the target dataset is if the social network is a tightly connected network which means everyone knows most of the others, the Degree-Discount heuristic algorithm will run much slower than the Basic heuristic algorithm.

Speedup Limitation

Monte Carlo Simulation

When the number of threads <= 4, the speedup is close to the number of threads. As the number of threads increases, the speedup will be affected more by the overhead of OpenMP and synchronization. The parallel seeds in a round have a worse speedup because of synchronization overhead.

Greedy

The best speedup we can achieve is 4.47x when nThreads = 32. The speedup is not good for each nThreads and the main reason is the OpenMP overhead. We used the perf tool and found out that the malloc and int_free take over 40% of the execution time. Most of them are OpenMP overheads and this could be the performance bottleneck of our greedy approach.

Heuristic

The best speedup we can achieve is 3.18x when nThreads = 64. The speedup is not good for each nThreads and the main reason is the locking overhead. Because in the parallel version we have to update the degree2ids very frequently and there will be locking operations very often.

Deeper Analysis and Future Work

We cannot break the execution time of our algorithms because the execution times from different components are heavily dependent on input parameters.

In order to further improve the performance, one possible way is to reduce the use of OpenMP to reduce its overhead. Another way could be to reduce the synchronization between threads with a reduction in spread results.

Meanwhile, we do not recommend implementing our algorithms using CUDA/GPUs due to the size of processing data, heavy dependence between different components, and the algorithm logic.

References

[1] Domingos P, Richardson M (2001) Mining the network value of customers. In: Proc SIGKDD, San Francisco, pp 57–66

[2] Richardson M, Domingos P (2002) Mining knowledge-sharing sites for viral marketing. In: Proc SIGKDD, Edmonton, Alberta, pp 61–70

[3] Kempe D, Kleinberg J, Tardos E (2003) Maximizing the spread of influence through a social network. In: Proc SIGKDD, Washington, pp 137–146

[4] Chen W , Wang Y , Yang S . Efficient influence maximization in social networks[C]. Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009. ACM, 2009.

[5] Ding-Zhu Du, Presentation on theme: "Influence Maximization", https://slideplayer.com/slide/13485154/

Work Distribution and Goals

Team Member	Work Distribution	Credit Distribution
Zican Yang (zicany)	Design and implementation	50%
Yuling Wu (yulingw)	Design and implementation	50%

Goals:

In this project, we not only reached the 100% target in the proposal but also finished the 125% goals. Especially regarding the 125% goals, we provided a very detailed and comprehensive multidimensional scaling report in the Results part that analyzed the features of our influence maximization algorithms.