15-418/618, Fall 2023
Final Report

Haoxi Zhang haoxiz@andrew.cmu.edu
Junxi He junxih@andrew.cmu.edu

December 15, 2023

1 Summary

We implemented a social network visualization algorithm in both OpenMP and Open MPI on PSC and compare the performance of the two implementations. We also explore a modified algorithm to achieve larger speedup and compare the final layout.

2 Background

In force-based layout systems, the graph drawing software modifies an initial vertex placement by continuously moving the vertices according to a system of forces based on physical metaphors related to systems of springs or molecular mechanics. In this project, we implement a similar algorithm to visualize node with 4 different types. The pseudo code is as followed:

```
For i in range(Epoch):
    # for each pair of points, calculate repulsive force
    for x, y in zip(points, points):
        updateRepulsive();
    # for each edge, calculate attractive force
    for x, y in edges:
        updateAttractive();
    # for each points, calculate gravity to the center of graph
    for x in points:
        updateGravity();
    # update position
    for x in points:
        updatePosition();
```

attractive force is calculated using the formula:

$$f_a(d) = d^2/k$$

repulsive force is calculated using the formula:

$$f_r(d) = -k^2/d$$

The formula is designed so that when the nodes of the same type are far, the attractive force is huge enough to pull them together and when their distance is small, attractive force will not make a big

difference. On the contrary, when the nodes of different type are far, the repulsive force is small and when their distance is large, repulsive force is huge.

It's also worth nothing k here is hyper parameter calculated by:

$$k = C\sqrt{area/number\_of\_vertices}$$

C is changeable during the process, allow the graph to first aggregate then disperse.
The gravity is calculated using:

$$f(p) = k * gravity * d$$

Gravity will be huge when point if far away from the origin point and otherwise have no effect.

The key data structure in our problem is a graph. Some pair of node of the same type exists an edge between then. The edge will provide attractive force between nodes. There is also repulsive force between nodes. Our algorithm will make nodes of the same type closer and use gravity to make the overview layout as a circle. We also reuse the quad-tree structure to achieve faster lookup.

The input of the algorithm is a graph, each node with mass, position and velocity. The output is the update position of each nodes. Then we use python to visualize the result layout.

The algorithm needs to go through a number of iterations to converge. each iteration is computationally expensive because we need to loop through each pair of nodes to calculate the force. What we do is paralleling the process of computing force for each pair of nodes in each iterations.

Each iteration must wait for all tasks to finish before it continue to the next iteration. We utilize space locality and data-parallel to do the parallelism. For each node, it will only compute nodes with nearby nodes. For further nodes, we just ignore the repulsive force between them. But in this case, we still need to go through all nodes to compute attractive force. Later we use a modified algorithm that also ignore the attractive force from the nodes that are too far from the original node.

3 Approach & Performance

We used C++ with OpenMP and Open MPI as our primary weapons. The target machine of our product can be any type of Linux machine that support OpenMP and Open MPI with sufficient computational resources. Ideally, PSC could represent a typical machine to run our program.

3.1 Workload Mapping & algorithm

For OpenMP version, data are shared among cores, so we divide the work into tasks and cpu is responsible for scheduling tasks among cores. In open mpi version, we divide the space to blocks, each block will compute the result for nodes in its area. Each block will be mapped to a core in the PSC machine. Nodes may change position between blocks so we also update information.

We also changed the original serial algorithm, which is $O(n^2)$ by iterating through all nodes from all nodes. It could produce an idea results by is very slow. We will use that result as standard results in comparing our results generated by the parallel version of our code. To achieve a better speed up and to create more possibilities for parallel, we experimented 2 algorithm.

The first algorithm is the Barnes Hut algorithm, which uses a Quad-Tree to split all the nodes in different regions. With a maximum radius for a node to be affected by other nodes, we could improve the time complexity to $O(nlogn)$ because the search time in a tree is $O(logn)$.

The second algorithm is the Fast Multipole Method (FMM). In the FMM algorithm, we have a similar set up as Barnes Hut, but instead of calculating relationship between nodes, we are calculating relationship between regions (cells, blocks, whatever we call it). A region of nodes can be viewed as

one node in the center of the region with all nodes in that region combined. Moreover, a region could be considered as a combination of its 4 sub-regions. In general, the FFM sacrifice accuracy for an incredible $O(n)$ time complexity. However, FFM might be a very suitable algorithm of the N-Body problem we had as homework, but we couldn't find a very constructive coefficient algorithm to apply to our Social Network Visualization, resulting in very bad graphic drawing. Hence, we decided to proceed with the Barnes Hut algorithm, which is more suitable for our purpose and more convenient to add more forces (repulsive, attractive, global gravity, regional gravity) on the nodes.

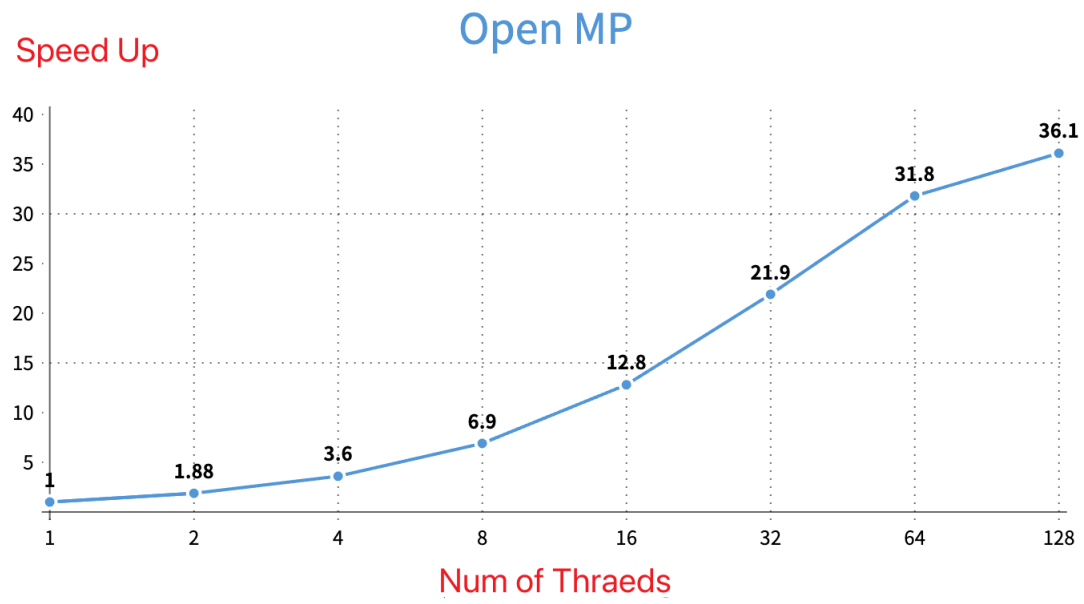
To measure our performance, we used a wall clock timer by using the Timer class and Timer.elapsed in C++.

The basic set up for performance test is using 5000 randomly generated and placed node in a 500x500 world. We also used other scenes like placing nodes in a corner of the world or preventing the nodes from some specific regions of the world and test our code in different scenes. We borrowed the idea of how to simulate a world full of particles from Project 4: N-Body Simulation.

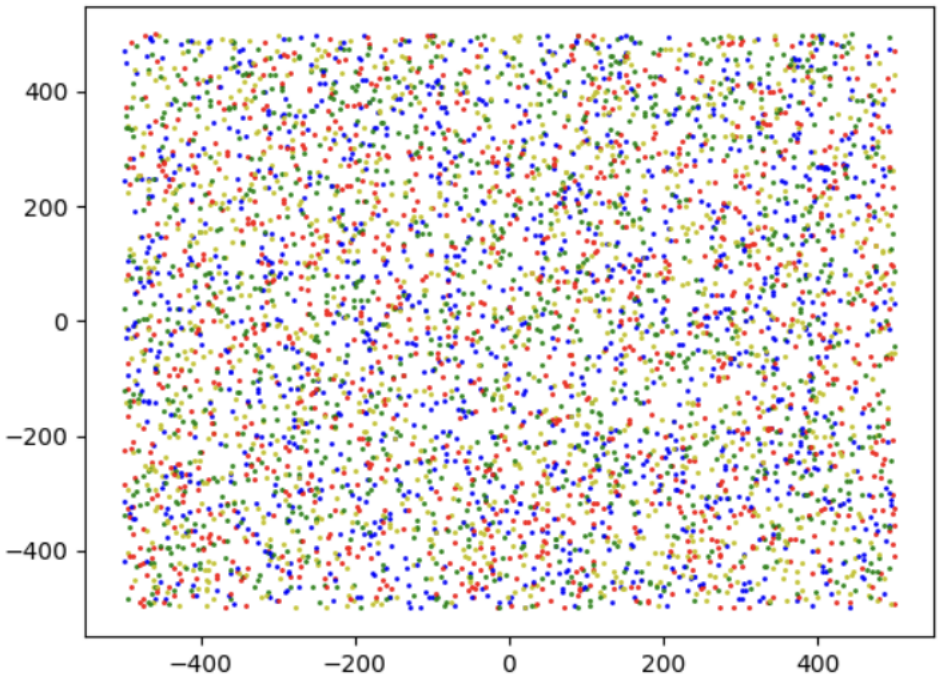
3.2 Different Approach for Speedup

IMPORTANT NOTE: All performance measurement in the below graph includes 50 iterations, not just a single iteration.

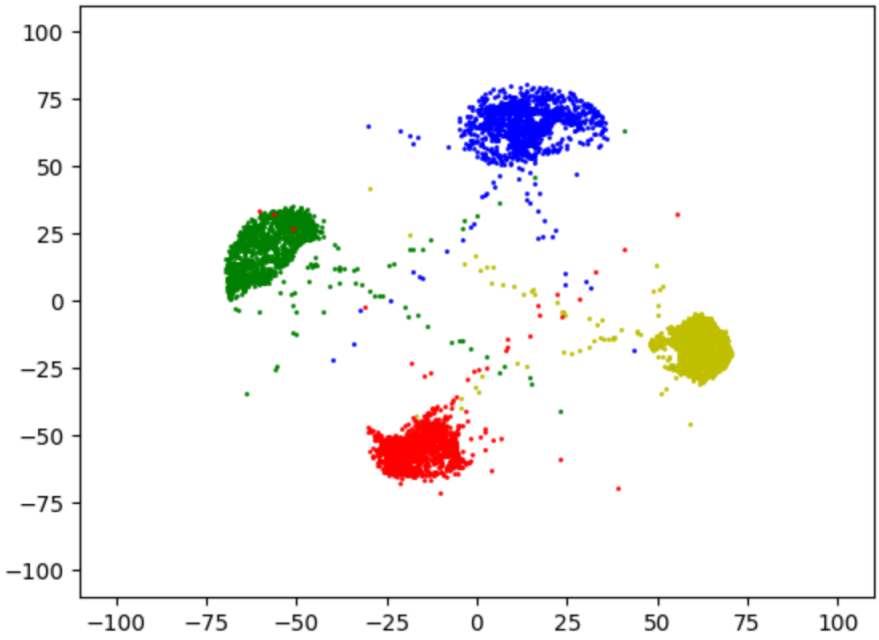
Optimization 1:
To further optimize our program, first, we add the Open MP part in our program. We use a graph with 5000 randomly generated and spreaded nodes to test.



As the graph tells, our program has a almost linear speed up in terms of Number of Threads generated by OpenMP.
A standard input is like:

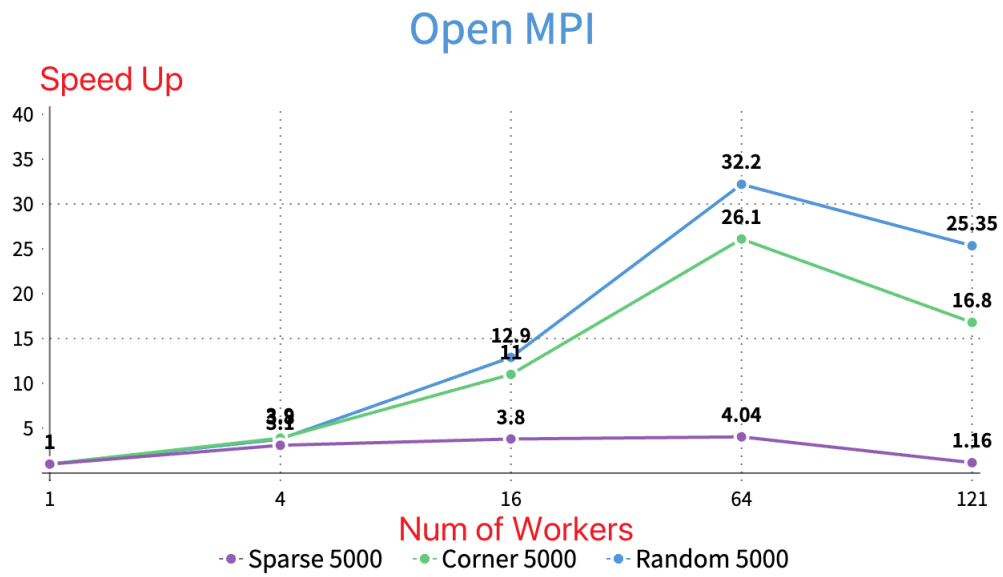


A standard output is like:



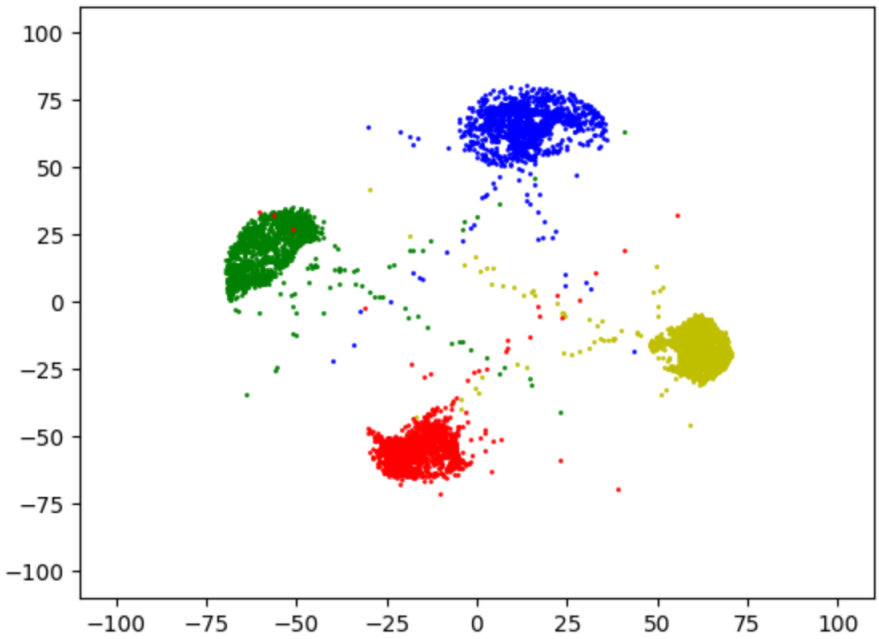
Notice the scale between input and output, which indicates that our final result concentrates all nodes in the center.

Optimization 2:
The second optimization comes with the Open MPI. Firstly, we use Open MPI with fixed regions for each worker node in the MPI group. That means each worker node has a region assigned to it. To test our performance, we generated more types of scenes.
Note that some scenes might not be possible in real world's social network data, but they are very good for performance evaluation.

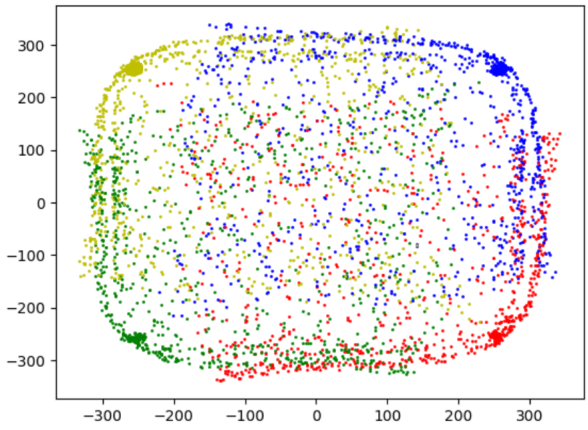


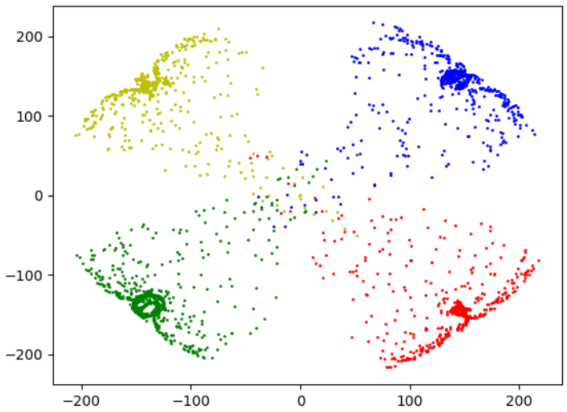
As the graph shows, our Open MPI implementation has a pretty good speed up related to the number of workers. The drop after 64 workers is expected because a single node, that we requested on PSC, only has 64 cores, hence having more than 64 workers does not produce any good but harm the overall performance.

The output of random 5000 form Open MPI is the same as Open MP, which proves the correctness.



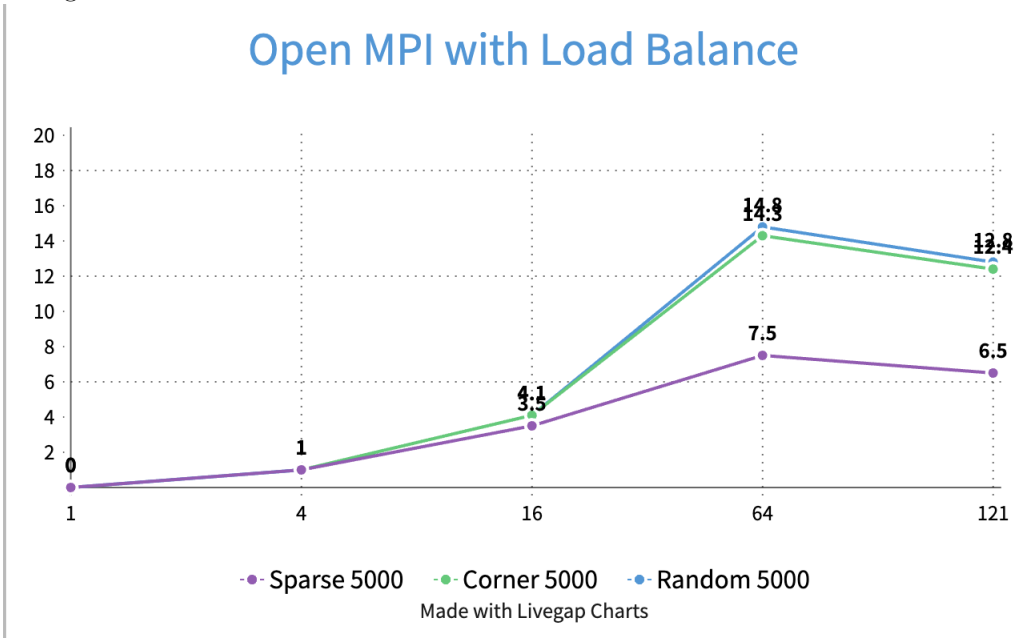
As you may probably noticed, the Sparse 5000 has a very bad speed up and we are sorry about that, even though it generated pretty beautiful graphs.





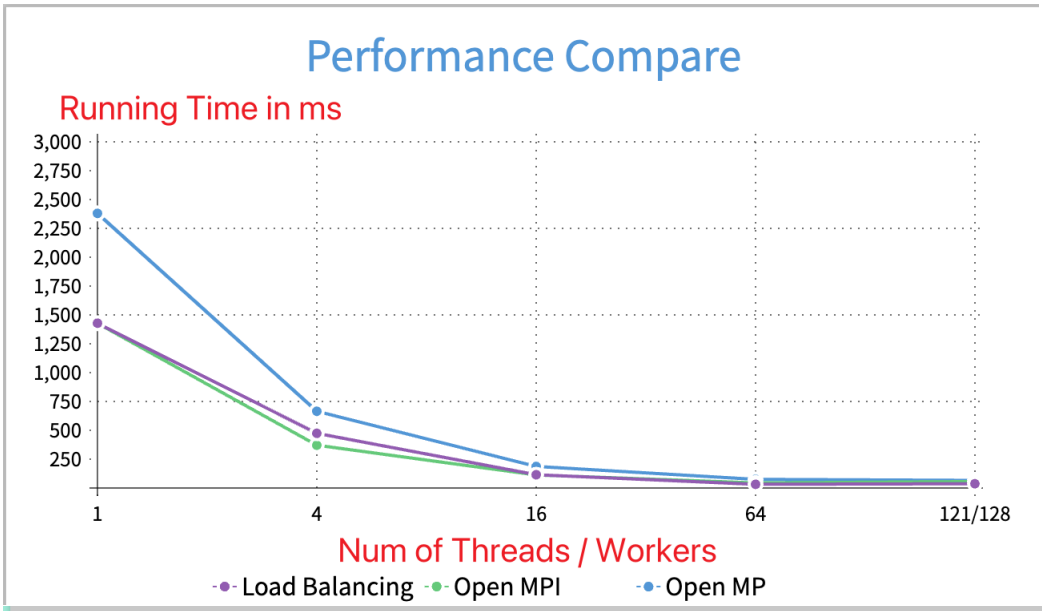
The problem with the Sparse scene lead to think about load balance in our MPI program. Any graph, after running each iterations of our algorithm, is concentrating in the center, which leads to a very unbalanced workload for workers. Remember that our graphs starts with 500x500 but the graphs above are only 200x200, which means that more than half of the workers are not doing anything but wasting resources. That explains why there is a huge drop after having 121 workers.

Optimization 3:
To solve the workload imbalance problem, we created a version of 1 + N solutions of MPI. A master node for dividing all nodes into smaller size of tasks (such as 10 nodes in 5000 nodes scenario), and then give each tasks to all available worker nodes.



As shown in the above graph, the performance for sparse scene is now related to number of workers.

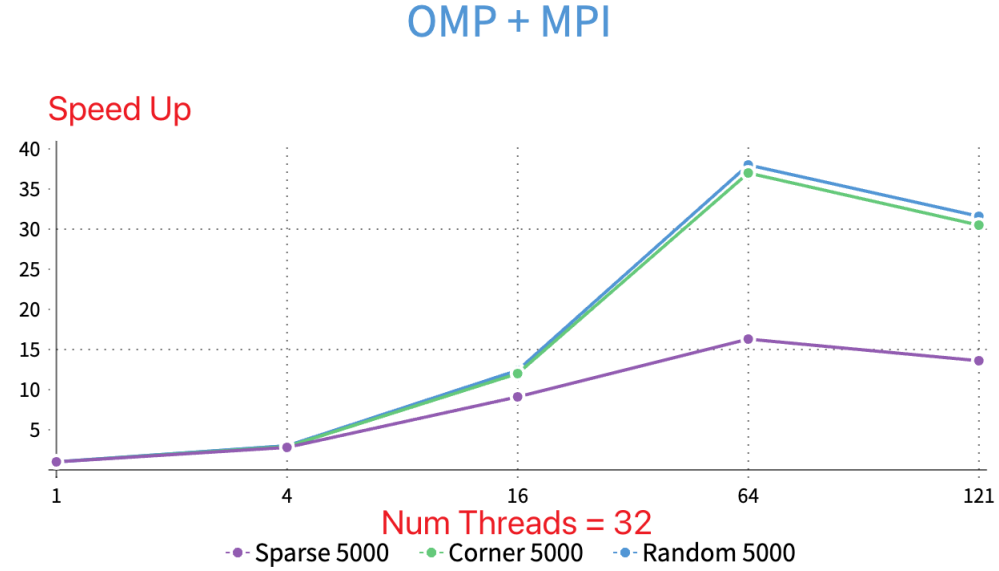
The speed up graph of only Load Balancing could not indicate how much performance gain from implementing the Load Balancing techniques. The below graph compares 3 strategies we used, Open MP, Open MPI, and Open MPI with load balancing, in the random 5000 scene.



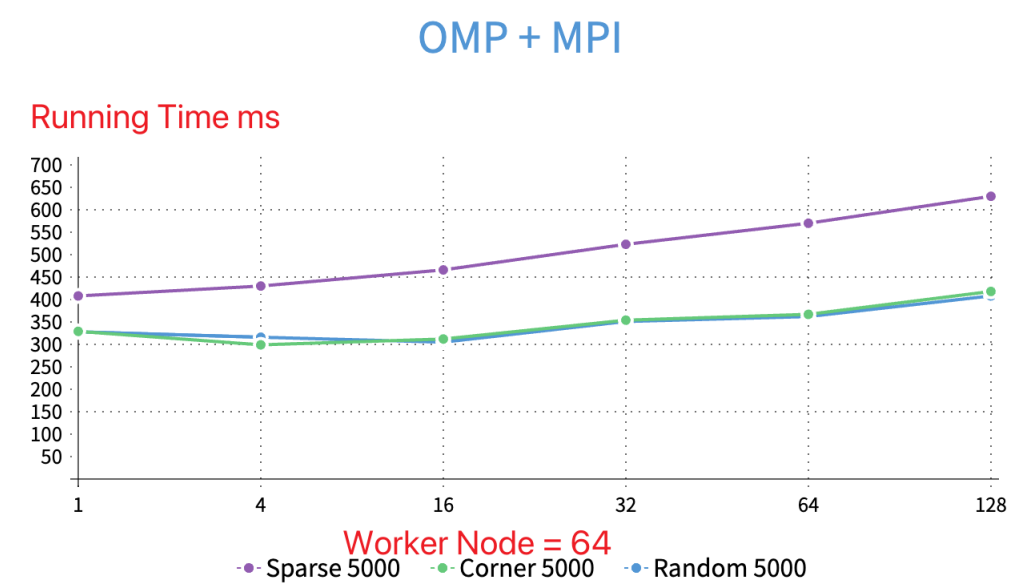
	1	4	16	64	121/128
Open MP	2380	665	187	75	66
Open MPI	1428	371	113	44	56
Load Balancing	1428	474	115	32	37

The above charts indicates the better performance achieved by implementing load balancing. Load Balancing have approximately **37%** speed up against MPI without load balancing and **132%** speed up against Open MP.

Optimization 4:
Our final optimization is combining Open MP with Open MPI, expecting to get more speed up. We first tried to run our code in the single node from PSC, but the result we gained is not good.



As the above graphs shows, with fixed threads count of Open MP, the speed up is related to number of worker nodes given to Open MPI. However, the running time actually did not improve from OMP + MPI, the next graph of running time will explain it.

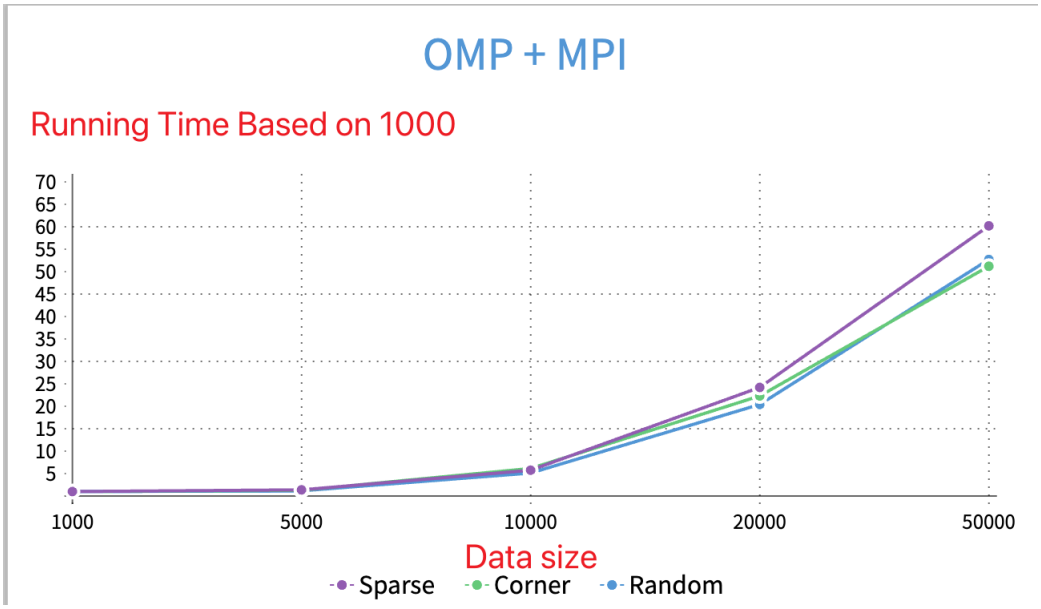


The above graph is generated by using fixed number of 64 workers in Open MPI and testing different number of threads. There is very little speed up if the thread count is set to 4 but also a performance lose for all scenes if the thread count is set to larger than 4.

The reason behind a zero-performance gain or a performance loss is because we are running our MPI + MP program within a single node in the PSC. Based on the resources information of PSC, a single node is an AMD CPU with 64 cores, and either Open MPI or OMP could drain the single node's resource and thus adding more parallelism to our code could only lead to a performance loss as it adds more complexity and resource allocation to the program without using more resource of the physical device.

A solution to this problem is using more nodes from PSC by using the **interact -p RM -n number-of-nodes** command to request more nodes for computing. And then use **srun -n 8 -c 40 –ntasks-per-node=4 ./hybrid-prog** to run our program so that it could take advantage of both OMP and Open MPI. Due to lack of time in our schedule, we didn't successfully launched that in PSC as PSC's Slurm does not support initializing Open MPI among different nodes rather than inside a single node. We believe that there is a solution to it but we didn't have time to figure it out.

As of data size, we compare our performance from data size of 1000 to 50000, and our results is shown in the below image.



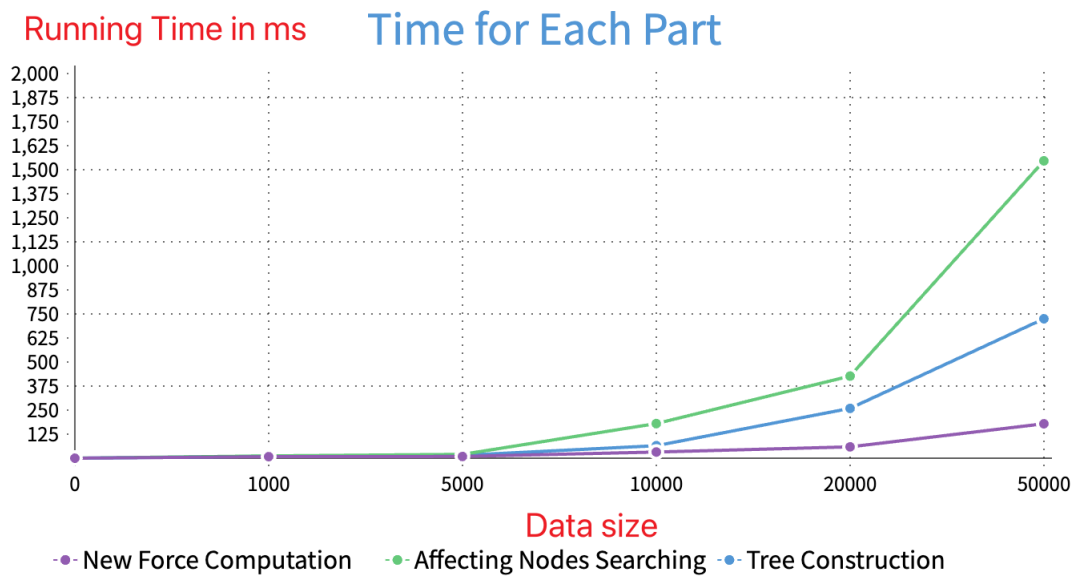
The test set up of the above image is using the best version of our code, with 64 workers and 4 thread counts, and running the graph 50 iterations. The graph showed that for small data size, the running time does not having much difference because the resource allocation and switching is dominating there. When the data size gets large, the running time is dominated by the data size and the running time is linear to the data size. It is not completely linear because our program is $O(nlogn)$ time complexity.

3.3 Limitation of Speedup

One main limitation for speedup is from the algorithm itself. As the iteration goes on, nodes will gather into specific blocks and several blocks will be empty, resulting workload imbalance. This case greatly reduce the speedup. This also accounts for why speedup goes down as iteration number increase. In this case, a good way to improve performance would be changing the mode of parallelism to master workload balancing after detecting workload imbalanced.

Other limitation of our speedup is the synchronization overhead. When number of workers is less than 64, we get expected result that speedup follow the exponent pattern. But we also observe the speedup is not linear because we have overhead except parallel computing. For the OpenMP version, from the graph above, we observe the overhead is synchronize data between cores and schedule tasks between cores. For the Open mpi version, the overhead is synchronize updated node position. A good way to get more speedup is changing the algorithm to reduce overhead. The following paragraph provide more detail analysis for OpenMP overhead.

We use wall-timer and accumulative global variables to record our running time.



Deeper Analysis: Our running time is mostly from 3 components: building the quad tree, finding the node neighbor, and computing the force.
The first part of building a quad tree takes about 20 - 35% of time in each iteration, depending on how the nodes are separated in the graph.
The second part of finding the node neighbors takes about 60 - 75% of time in each iteration. This is the part where we could improve the most, if we could replace our algorithm with the FMM algorithm we experimented earlier, we could achieved about 2 - 4x speed up in this part.
The third part of computing force only takes about 5 - 10% of time in each iterations. It uses a lot of divisions though. To improve the performance here, we could avoid division by pre-calculating a division value and then using multiplication.

Limitation is also related to the resource. When the number of worker is 128, all of our test achieves worse result than 64 workers. This is limited by the resource a cpu can provide. Each node on PSC has 64 cores, though each cores support 2 threads, scheduling this two threads in fact introduces additional overhead. As our algorithm is computing intense, one thread is enough to fully utilize the core resource.

Based on the performance evaluation, we believe that choosing the PSC machine is very sound. Our program could utilize its all resources and take advantage of Open MPI there.

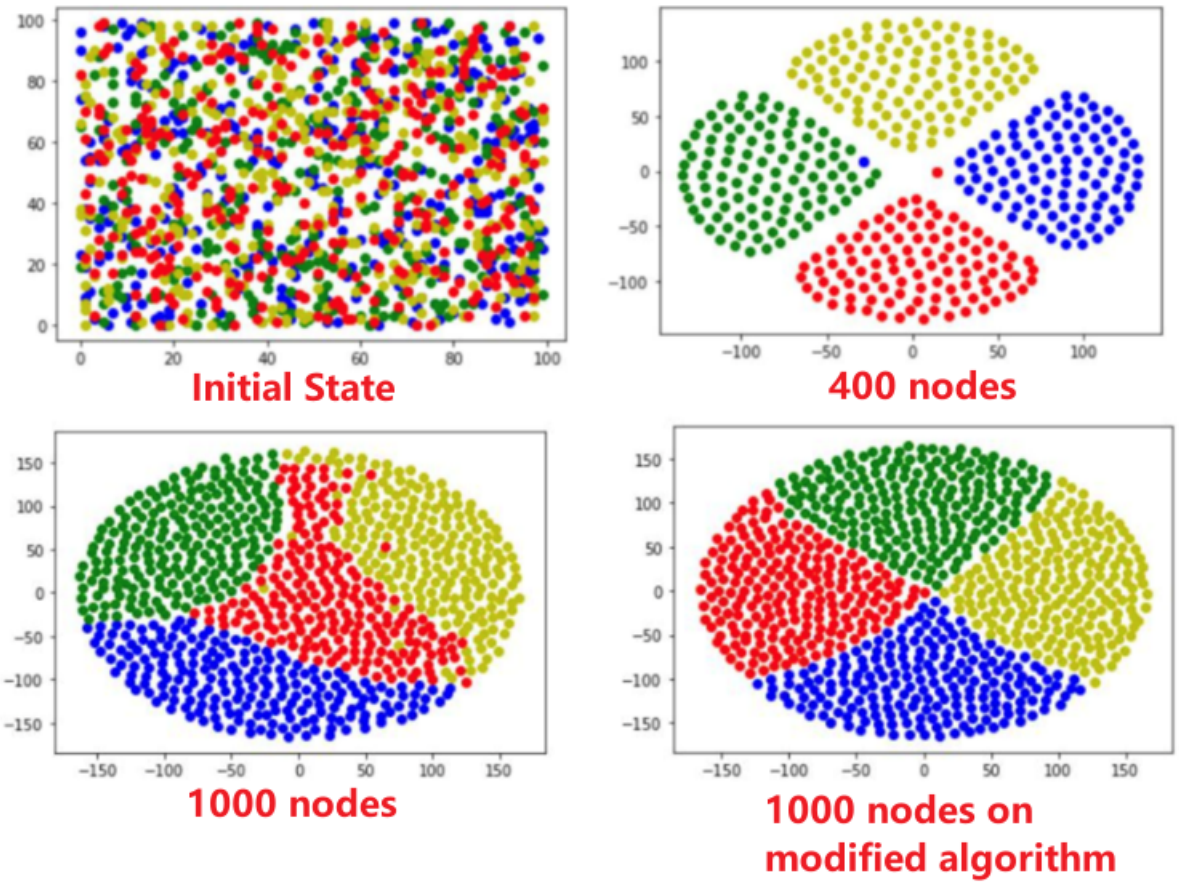
4 Result & Correctness

In this section, we analyze the performance and layout effect of changing different parameter and a modified algorithm.

4.1 Experiments on Small Dataset

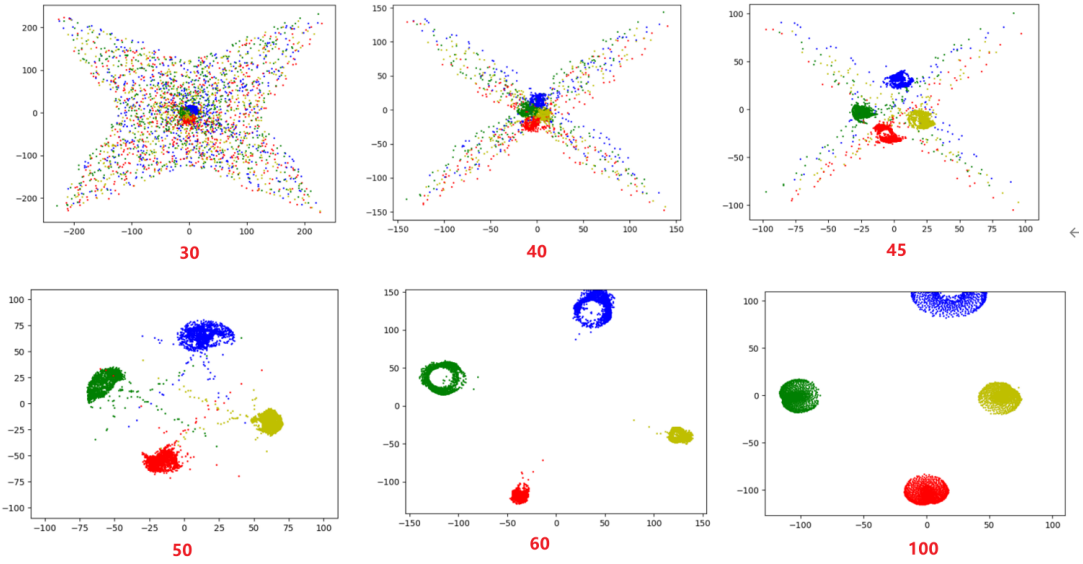
Our first step is testing on small dataset that are less than 1000 nodes to ensure the algorithm correctness. The layout is as followed. We observe the layout can change greatly as nodes number grows. That's because the gravity force is unchanged as nodes number grows but the repulsive force between particles will grow. To achieve the same layout, we need to manually increase the gravity parameter.

From the graph we also observer not all vertices will converge. Some points are in the other's area but the force is already balanced. The same thing happen for more nodes that not all nodes are in desired position.



4.2 Experiments on Large Dataset

After confirming correctness on small dataset, we do experiments focus on 5000 nodes. The following graph is the layout of different iterations.



We observe different stages of our algorithm and the dominant force in each stage:

1. In the 30 iteration stage, gravity is the dominant force for nodes near center so they overlap together. But for nodes that are far away from the center, they suffer from the repulsive force at the diagonal line so there position are not changed. We also observe a tendency that the edge points begin to shrink.
2. In the 40 iteration stage, we observe the center cluster start to disperse because more and more nodes gather together and increase the repulsive force between colors. More edge nodes get closer to the center and only diagonall line remain unchanged.
3. In the 45 iteration stage, we observe clearly each color cluster starts to move further from each other because the total repulsive force is larger than the gravity. At the same time, the remaining scatter nodes move slowly to the target color cluster.
4. In the 50 iteration stage, we observe the four cluster separate with each other. In this case, gravity is no longer important, the main force between nodes are attractive force between same color nodes and repulsive force.
5. In the 60 iteration stage, we get a desirable result that separate different color but some parts are too small because nodes overlap with each other and some parts have a hole. In this case, force between different types are no longer important and the repulsive force between the same type node plays the main role. It need more iterations to make a cluster circle.
6. In the 100 iterations stage, we get really good result but compare to the result in 50 iterations stage, it takes three times of running time but only provide similar layout. We choose 50 iterations as our main experiment parameter is considering this tradeoff.

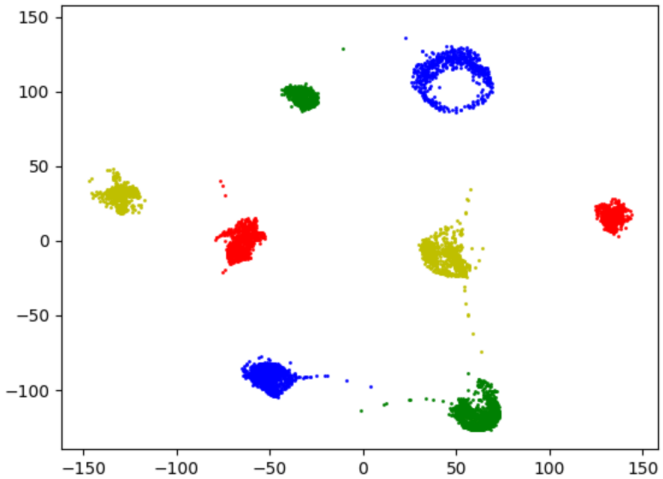
From this example, we find our algorithm has really complicated stage and it's hard to optimize parameter to get a desired result. For example, if we set the gravity to a larger number, nodes will overlap with each other in the early stage and unable to reach convergence. Then it will take longer time to reach the final scene, affecting the speedup. This also accounts for why the layout of large number of nodes are really different from layout of small number of nodes.

It's also worth noting from 40 iteration stage, some blocks no longer hold nodes and this lead to inefficient under open mpi implementation. Here is a potential point to achieve a bigger speedup.

4.3 Experiments with Release Constrain

This algorithm is original $O(n^2)$ because we need to provide attractive force to the furthermost node pairs. By only considering nearby particle, we try to release this constrain and this what we get.

We observe instead of having only one cluster, the result contains two clusters for the same colors. This is because the nodes out of boundary will no longer provide attractive force so they will form a cluster itself. This algorithm is really fast because it no longer need to go through all node pairs. It reduce the complexity to $O(nlog(n))$. We consider this is also an acceptable layout under certain scene.



4.4 Experiments with Modified Algorithm

To solve the above problem, under the case that we only want one cluster for each color, we design a new algorithm that sets a center position for each color at the initialization step. Instead of adding gravity to origin point for every node, we add gravity to each color center according to their types. In this case, during the force calculation, we can not only ignore repulsive force from distant nodes but also ignore the attractive force because these nodes will come closer as the result of gravity. So we can use the same algorithm used in Project 4. This is a huge improvement on both layout and speed. The result is as followed.

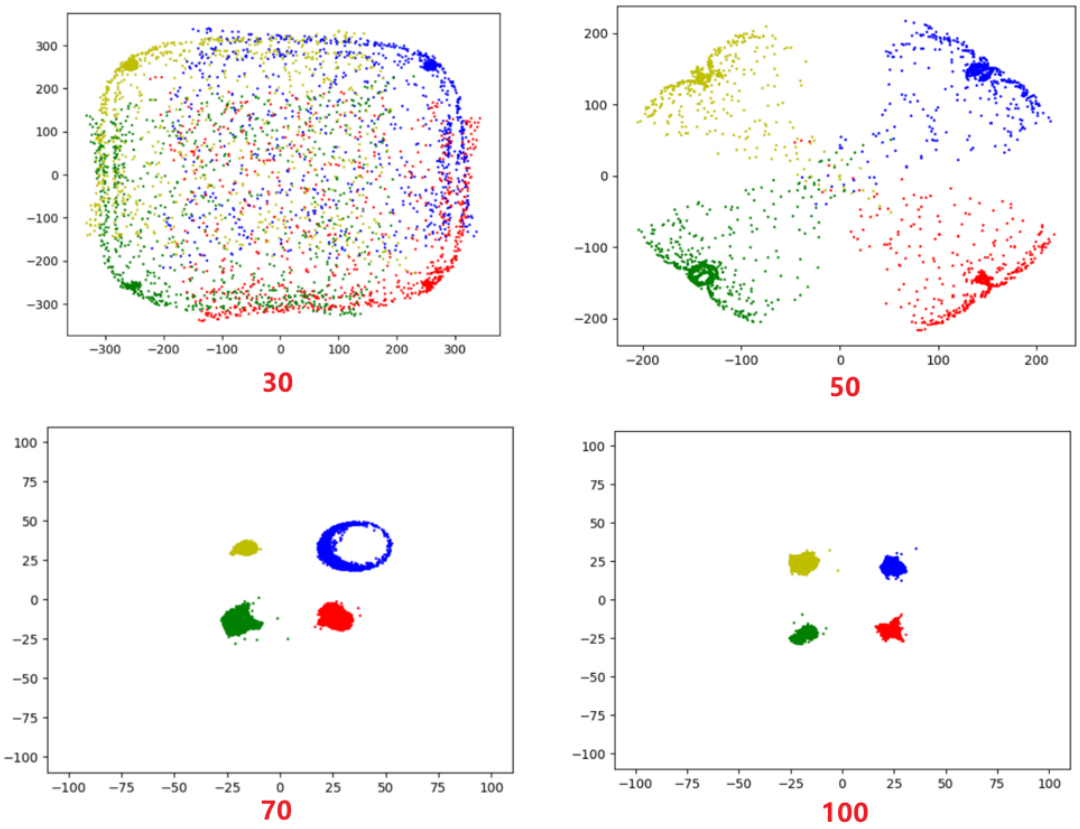
Under 70 iterations case, the initial algorithm used 7.4s, the new algorithm only use 0.34s. Compare to the initial algorithm, we achieve 22x speedup. The outcome boundary is also clearer than the initial method during the early stage. All of this prove this new algorithm to be a better choice under specific assumption.

The following are detail analysis for each stage.

1. In the 30 iteration stage, we observe the tendency that the nodes of the same color start to move towards their color center. In this stage, the result effect already better than the original algorithm.

2. In the 50 iteration stage, we observe a really beautiful layout consist of four component. During this time, the nodes have already been separated .

3. In the 70 and 100 iteration stage, we get a similar layout as the origin algorithm but the speedup achieved is huge.

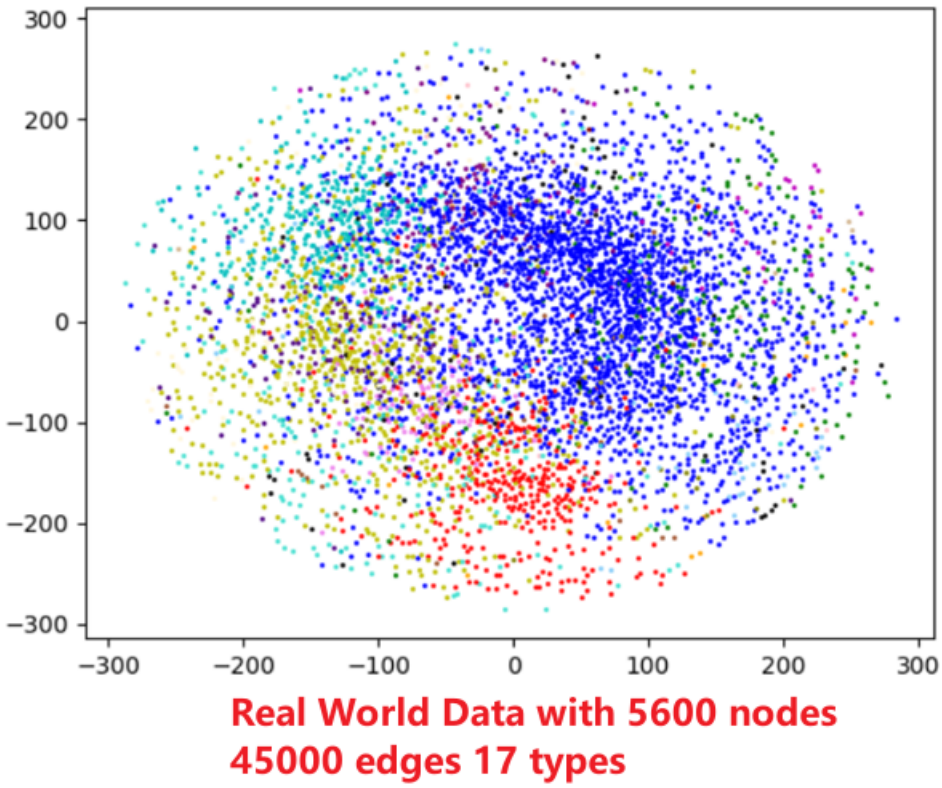


4.5 Experiments on Real world Dataset

Finally, we test on the real world dataset. In this case, we have around 4500 nodes and 42000 edges. Not all pairs of nodes of the same type have the edge between them and there is also edges between

different type nodes. From this real world dataset, we prove our algorithm is practical and has real world application.

We observe a very different view from previous test because the repulsive force is no longer the dominate force. We also increase the gravity to make sure the graph is around a circle. From the graph, we can observe a lot of information related to the network.We observe the blue type is the main component and there is other components like yellow and black are close to each other, which provide motivation for the next step to analyze the social network.



5 Unfinished work and Future Work

We tried to extend the algorithm to 3D space. In fact, we already finish codes for Oct Tree but have no time to test it out. Changing the algorithm require a lot of effort to rewrite the basic framework. Especially for open mpi, we need to completely change the way nodes passing out of bound nodes to neighbour.

We also tried to use more nodes on PSC to test the effect of more resource on our algorithm. The idea is promising since our bottleneck for OpenMP + open mpi version is the lack of resource. From our research, we need to use the Slurm manager but it seems PSC doesn't support this application. As we are not familiar with PSC, we didn't work on this direction.

Our very first goal is achieving 100000 nodes parallelism but later we find it impossible because even using the modified algorithm that complexity is $O(nlog(n))$, we still need to run 50 iterations and the workload is becoming imbalanced. It will take around 2 hours to run the whole task and 20 minutes to draw the graph. So we decide to use 5000 nodes for testing purpose. As 5000 nodes provide the correct result, we believe the algorithm also works for 100000 nodes.

In the future, we can explore more algorithms according to our speedup bottleneck. Our algorithm doesn't care about accuracy but only care about final layout effect so we have a large tuning space. As we analyze above, we have several stages for this algorithm and a large space to achieve bigger

speedup for each stage. We can also dynamically adjust the parameter to achieve desired layout like gravity will grow bigger after nodes gathered together.

6 Reference

Blelloch, Guy E. and Girija J. Narlikar. "A practical comparison of N-body algorithms." Parallel Algorithms (1994).

7 List of Work

We work closely in an in-person mode.
Haoxi Zhang: 50%. Implemented Open MPI, Force Computation, Gravity, and other parameters.
Junxi He: 50%. Implemented Open MP, Graph Drawing, Running Open MP + Open MPI in PSC, and others.