COL380 Assignment 3

Sibasish Rout (2020CS10386)
Ankit Raushan (2020CS10324)

TASK 1

1. Approach

Here we have used a distributed approach using Message Passing Interface And OpenMP.

The steps involved are

i)First read the values of n,m from the .gra file using MPI_File_read_at. Next we read the .dat file to find the offsets of all the nodes and use them to calculate the degrees of each of the nodes. Then we sort the nodes according To the degrees and distribute them among all the nodes.

ii)We calculate and store all the triangles associated with a particular edge by again reading from the .gra file and then use two pointer method to calculate The common vertices for each edge.

iii)The next step involves deleting the edges having insufficient support values As per the mintruss algorithm. The edges which are deleted are then Communicated to the other processes using MPI_Alltoall collective call.

iv)After receiving the deleted edges from the other nodes the nodes make Changes into their own data structures as per the requirement.This continues till the entire graph stabilises.

v)After this we calculate the verbose using disjoint set union where all nodes Other than node 0 send their edges to node 0 which then constructs the DSU data structure.

vi)Using this data structure we perform a search to print all the components Accordingly. In this data structure all the nodes having the same root will Belong to the same connected component.

We are tabulating runtimes for approach 2 below. In the table np denotes number of processors on which we are running the code and num_threads denote the number of

threads. We have shown the runtimes for testcase 1,2,3 and 4 and old test case 3(which was taking a lot of runtime almost 60 sec in assignment 2). Runtimes are in seconds

np	**num_threads**	**Test 1 (4.8Mb)**	**Test 2 (3.2Mb)**	**Test 3 (5.3Mb)**	**Test 4 (18.9Mb)**	**old_Testcase 3 (10.1Mb)**
1	1	8.035	1.101	2.003	3.246	19.657
2	2	4.524	0.758	1.303	2.238	10.455
2	4	4.446	0.788	1.309	2.231	10.392
2	8	4.460	0.758	1.291	2.205	10.403
4	2	2.865	0.696	1.152	2.234	6.240
4	4	2.854	0.738	1.171	2.290	6.198
4	8	2.850	0.715	1.157	2.278	6.231
8	2	1.948	0.644	1.067	2.231	4.216
8	4	1.937	0.632	1.073	2.223	4.188
8	8	1.933	0.637	1.061	2.234	4.176

Table 1 - Running time(in seconds)

2.1 Graph for speedup for different core counts

Here we will be doing speedup analysis for our multithreaded distributed algorithm in tabulated form with the help of runtime table which we have calculated above.
In the table np denotes number of processors on which we are running the code
And num_threads denote number of threads

np	num_threads	Test 1 (4.8Mb)	Test 2 (3.2Mb)	Test 3 (5.3Mb)	Test 4 (18.9Mb)	old_Testcase 3 (10.1Mb)
1	1	1.000	1.000	1.000	1.000	1.000
2	2	1.776	1.452	1.537	1.450	1.880
2	4	1.807	1.397	1.530	1.454	1.891
2	8	1.801	1.374	1.551	1.472	1.889
4	2	2.804	1.581	1.738	1.452	3.150
4	4	2.815	1.491	1.710	1.417	3.171
4	8	2.819	1.539	1.731	1.424	3.154
8	2	4.124	1.709	1.877	1.454	4.662
8	4	4.148	1.742	1.866	1.460	4.693
8	8	4.156	1.728	1.887	1.452	4.707

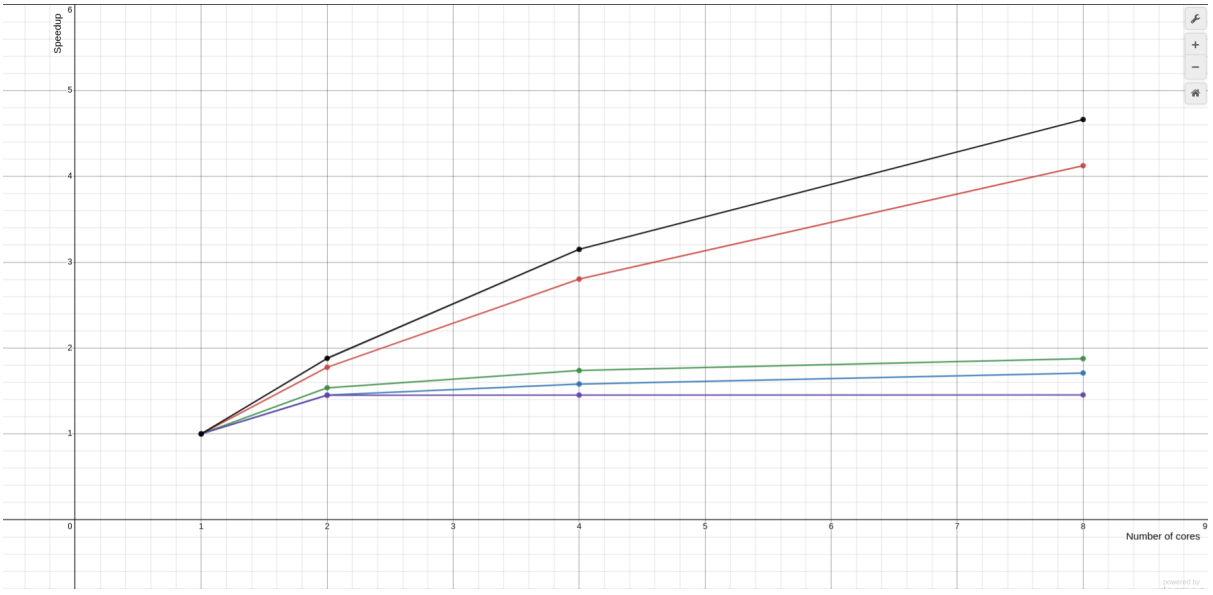
Table 2 - Speedup

We will make the graph for above table -

On X-axis there is

On Y-axis is shown

Here red-Test1,blue-Test2,green-Test3, purple-Test4,black-old test3

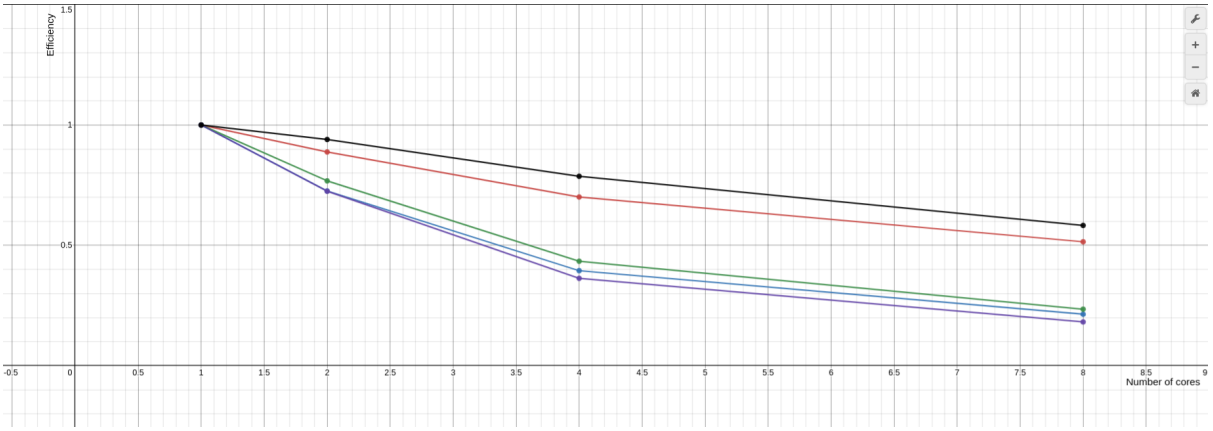


2.2 Graph for efficiency for different core counts

Here we will be doing efficiency analysis for our multithreaded distributed algorithm tabulated form with the help of speedup table which we have calculated above.
In the table np denotes number of processors on which we are running the code
And num_threads denote number of threads

np	num_threads	Test 1 (4.8Mb)	Test 2 (3.2Mb)	Test 3 (5.3Mb)	Test 4 (18.9Mb)	old_Testcase 3 (10.1Mb)
1	1	1	1	1	1	**1**
2	2	0.888	0.726	0.768	0.725	0.94
2	4	0.903	0.699	0.765	0.727	0.946
2	8	0.9	0.687	0.775	0.736	0.945
4	2	0.701	0.395	0.434	0.363	0.787
4	4	0.704	0.373	0.427	0.354	0.793
4	8	0.705	0.385	0.433	0.356	0.788
8	2	0.515	0.214	0.235	0.182	0.583
8	4	0.518	0.218	0.233	0.182	0.587
8	8	0.519	0.216	0.236	0.181	0.588

Table 3 - Efficiency



red-Test1,blue-Test2,green-Test3, purple-Test4,black-old test3

3. Iso - efficiency

The Iso-efficiency function is $I(p) = O(plogp)$
Explanation -
For calculating the isoefficiency we must first estimate the time complexity of working of our algorithm. The most time consuming part in our algorithm is the part that calculating mintruss at each step. If we assume that the number of iterations is dependent on O(n+m) wher n- number of vertices and m - number of edges. The overhead due to MPI_Alltoall used here is of the O(logp) as Allto all takes log time to be carried out in case we proceed in a binary tree type of method.
The time complexity of the algorithm =$O((n + m)/p + logp) = t(n, p)$
As we know that $E = t(n, 1)/pt(n, p)$ which must be constant for isoefficiency.
$pt(n, p) = O((n + m) + plogp)$
$t(n, 1) = O(n + m)$
Thus for the efficiency to be the same the problem must grow at the rate of O(plogp).

Thus we can say that our algorithm is moderately scalable as we need not increase the problem size much to keep the efficiency same.
This is also supported by the values from metric.csv file

4. Sequential fraction for 8 -64 cores

The sequential fraction can be estimated by using Karp-Flatt metric
Sequential part, f=(1/Sp-1/p)/(1-1/p) by using Karp Flatt Metric where Sp=speedup, p-number of processors
The speedup values for 8-64 cores for oldtestcase 3 are

Number of cores (1 thread)	Speedup	Sequential fraction
8	3.28	0.20557
16	4.55	0.1677
32	6.62	0.1236
64	8.05	0.1103

Thus the sequential fraction for the problem appears to be at around 15% .

5. Why our solution for task 1 is scalable?

A small isoefficiency function implies that small increments in the problem size are sufficient to use an increasing number of processors efficiently

As we can see from the scalability and isoefficiency values it becomes clear that we need not increase the size of the problem by a large extent for the efficiency to remain the same. Also the sequential portion for the problem is relatively quite small hence the solutionn for task1 is scalable.

TASK 2

1. Approach

Here we have used a distributed approach using Message Passing Interface

And OpenMP.

The steps involved are

i)First read the values of n,m from the .gra file using MPI_File_read_at.

Next we read the .dat file to find the offsets of all the nodes and use them to

calculate the degrees of each of the nodes. Then we sort the nodes according

To the degrees and distribute them among all the nodes.

ii)We calculate and store all the triangles associated with a particular edge by

again reading from the .gra file and then use two pointer method to calculate

The common vertices for each edge.

iii)The next step involves deleting the edges having insufficient support values

As per the mintruss algorithm. The edges which are deleted are then

Communicated to the other processes using MPI_Alltoall collective call.

iv)After receiving the deleted edges from the other nodes the nodes make

Changes into their own data structures as per the requirement.This continues

till the entire graph stabilises.

v)After this we calculate the verbose using disjoint set union where all nodes

Other than node 0 send their edges to node 0 which then constructs the DSU

data structure.

vi)We now read the neighbours of each node and check how many leftover connected components it is connected to in the actual graph. If the value crosses the threshold value of p then we take that as a valid answer and print all the components it is connected to. Here we have used threading to improve the speed.

We are tabulating runtimes for our algorithm. In the table np denotes number of processors on which we are running the code and num_threads denote the number of threads. We have shown the runtimes for testcase 1,2,3 and 4 and old test case 3(which was taking a lot of runtime almost 60 sec in assignment 2). Runtimes are in seconds

np	**num_threads**	**Test 1 (4.8Mb)**	**Test 2 (3.2Mb)**	**Test 3 (5.3Mb)**	**Test 4 (18.9Mb)**	**old_Testcase 3 (10.1Mb)**
1	1	3.216	1.523	3.268	4.949	8.234
2	2	1.913	1.268	2.227	4.039	4.657
2	4	1.981	1.272	2.219	4.086	4.678
2	8	1.952	1.327	2.229	4.156	4.648
4	2	1.508	1.301	2.069	4.152	3.293
4	4	1.530	1.276	2.077	4.223	3.299
4	8	1.502	1.277	2.038	4.151	3.274
8	2	1.311	1.299	1.880	4.032	2.634
8	4	1.335	1.292	1.872	4.042	2.620
8	8	1.320	1.284	1.878	4.038	2.603

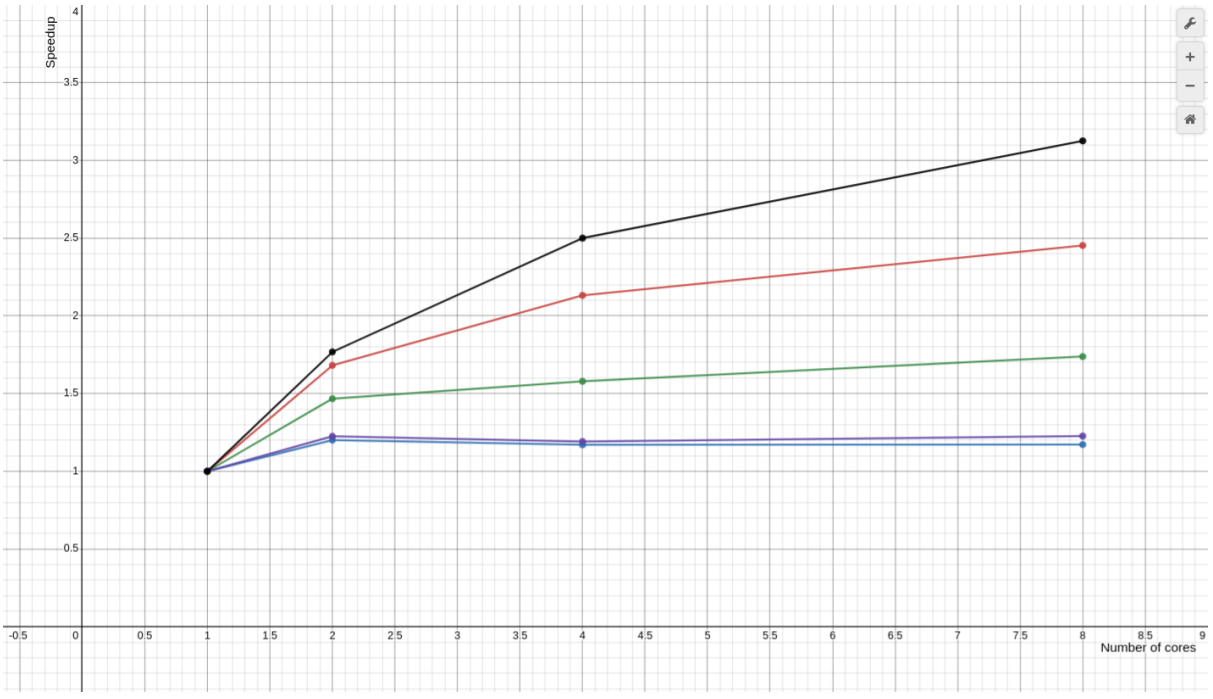
Table 1 - Running time(in second)

2.1 Speedup graph for different core counts

Here we will be doing speedup analysis for our multithreaded distributed algorithm in tabulated form with the help of runtime table which we have calculated above.
In the table np denotes number of processors on which we are running the code
And num_threads denote number of threads

np	num_threads	Test 1 (4.8Mb)	Test 2 (3.2Mb)	Test 3 (5.3Mb)	Test 4 (18.9Mb)	old_Testcase 3 (10.1Mb)
1	1	1.000	1.000	1.000	1.000	1.000
2	2	1.681	1.201	1.467	1.225	1.768
2	4	1.623	1.197	1.472	1.211	1.760
2	8	1.647	1.147	1.466	1.190	1.771
4	2	2.132	1.170	1.579	1.191	2.500
4	4	2.101	1.193	1.573	1.171	2.495
4	8	2.141	1.192	1.603	1.192	2.514
8	2	2.453	1.172	1.738	1.227	3.126
8	4	2.408	1.178	1.745	1.224	3.142
8	8	2.436	1.186	1.740	1.225	3.163

Table 2 - SpeedUp analysis



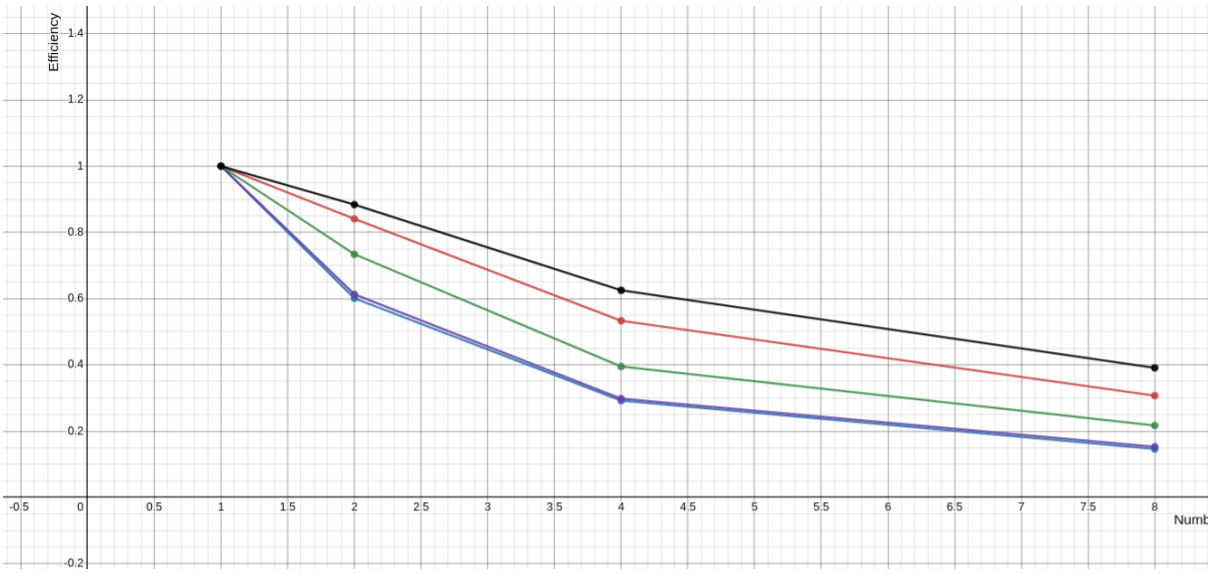
red-Test1,blue-Test2,green-Test3, purple-Test4,black-old test3

2.2 Efficiency graph for different core counts

Here we will be doing efficiency analysis for our multithreaded distributed algorithm tabulated form with the help of speedup table which we have calculated above.
In the table np denotes number of processors on which we are running the code
And num_threads denote number of threads

np	num_threads	Test 1 (4.8Mb)	Test 2 (3.2Mb)	Test 3 (5.3Mb)	Test 4 (18.9Mb)	old_Testcase 3 (10.1Mb)
1	1	**1**	**1**	**1**	**1**	**1**
2	2	0.841	0.601	0.734	0.613	0.884
2	4	0.811	0.599	0.736	0.606	0.88
2	8	0.824	0.574	0.733	0.595	0.885
4	2	0.533	0.292	0.395	0.298	0.625
4	4	0.525	0.298	0.393	0.293	0.624
4	8	0.535	0.298	0.401	0.298	0.628
8	2	0.307	0.146	0.217	0.153	0.391
8	4	0.301	0.147	0.218	0.153	0.393
8	8	0.304	0.148	0.217	0.153	0.395

Table 3 - Efficiency



red-Test1,blue-Test2,green-Test3, purple-Test4,black-old test3

3 Why our solution for task 2 is scalable

The solution is quite scalable as the indicated by the efficiency and scalability values. The scalability values show a positive trend with the number of cores. The code scales well with the number of cores and the isoefficiency function is not very large for large values. Since we have written the code for an unknown number of processors the code scales accordingly.

For Metric.csv

The isoefficiency is calculated as
iso-efficiency = (speedup) / (number of processors * efficiency)
Sequential part, f=(1/Sp-1/p)/(1-1/p) by using Karp Flatt Metric

References

1)https://www.geeksforgeeks.org/number-of-connected-components-of-a-graph-using-disjoint-set-union/
2)https://moodle.iitd.ac.in/pluginfile.php/343275/question/questiontext/619807/1/146100/MPI.pdf