COL380 Assignment 2

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

1. Approaches

1.1 Approach 1: Naive sequential min-truss algorithm -

At first we implemented basic min-truss algorithm without any filtering. We used mostly vectors or vector of vectors for storing degrees, adjacent nodes, support value etc. Though it runs correctly on test cases, it took much time on slightly larger test cases like 5, 6 and 7 and almost ran for much time on test case 3 before we have to terminate the execution.

Explanation -

This happened because since we had not done any filtering the algorithm performed Many redundant computations. Like the algorithm calculated support value for edges having degree less than k-2 which was unnecessary.

1.2 Approach2: Sequential min-truss algorithm with preprocesses -

In this iteration we preprocessed the given graph before applying the min-truss algorithm. We used prefilter(G,k) function to eliminate vertices having degree<k-1, used Initialise(G,k) to gather edges with low support for subsequent Filtering and Filteredges(G,k) to eliminate edges with low support. We used these Algorithms as per pseudocodes of these given in assignment pdf -

```
Prefilter(G,k): (Eliminate vertices with degree < k-1)
      Deletable = {v s.t. deg(v)<k-1}
      Repeat until |Deletable|>0 {
      ∀ v ∈ Deletable :
      G = G-v
      k/Queue neighbors for elimination if their degree dips below
      ∀ u ∈ adj (v):
      if (deg(u)<k-1) Deletable += u
}
```

```
Initialize(G,k):
      Deletable = ϕ
      ∀ e=(a,b) ∈ E :
      Supp(e) = |adj(a) ∩ adj(b)|
      if (Supp(e)<k-2) Deletable += e

FilterEdges(G,k): (Eliminate edges with low support)
      Repeat until |Deletable|>0 {
      ∀ e=(a,b) ∈ Deletable :
      G = G-e
      // Removing this edge reduces support for others .
      ∀ edges f supporting e :
      Supp(f)--
      if (Supp(f) < k-2) Deletable += f
}
```

Since calculating support was taking much time hence filtering
Helped in calculating supports of only relevant edges henced saved a
Lot of time. Test case 3 which was taking almost infinite time now
Got executed in 480 seconds.

We are tabulating runtimes for approach 2 below. In the table np denotes number of processors on which we are running the code. We have shown the runtimes for test Case 2, 3, 5, 6, 8. Runtimes are in seconds

np	**Test 2**	**Test 3**	**Test 5**	**Test 6**	**Test 8**
1	2.680	487.763	58.466	188.293	7.522
2	2.782	496.232	52.868	182.269	7.508
4	2.598	468.424	60.090	192.464	7.804

Table 1 - Running time(in second) for approach 2

Observation -

We can observe that even when we increase the number of processors run time does not change significantly. This was expected because it is a sequential code hence instead of distributing the loads all processors are doing the same work.

1.3 Approach 3: Final modified parallel algorithm (distributed algorithm) -

Since initializer and filteredges were taking most of the times. So we distributed these tasks into multiple processors to decrease the total runtime. Also we used sets and maps instead of vectors which we had used in sequential and it reduced runtime significantly. In test case 3 time reduced significantly to 60 seconds using 4 processors.

Here are the steps we have used in parallelising using MPI -

i)The edges are a limiting factors in determining computation time of the problem. This is because most of the computation revolves around the edges. The steps involving filteredge and calculation of the support values involves the maximum time of computation.

ii)To introduce parallelism in the code we can **distribute the edges** among the processors And the calculate the support values for the edges individually using local vectors and a local View of the edges .

iii)The nodes after performing calculations for their edges using a local view send the values Send the values back to a master node which is then responsible for synchronisation Operations between the nodes using MPI_Send , MPI_Recv and MPI_Barrier to synchronise All the views between the processors. The additional MPI items used are MPI collectives , MPI Datatypes and non blocking calls.

We are tabulating runtimes for approach 2 below. In the table np denotes number of processors on which we are running the code. We have shown the runtimes for test Case 2, 3, 5, 6, 8. Runtimes are in seconds

np	**Test 2**	**Test 3**	**Test 5**	**Test 6**	**Test 8**
1	0.633	74.054	9.697	32.932	1.333
2	0.595	62.240	8.375	20.82	1.222
4	0.611	60.132	8.115	16.164	1.291

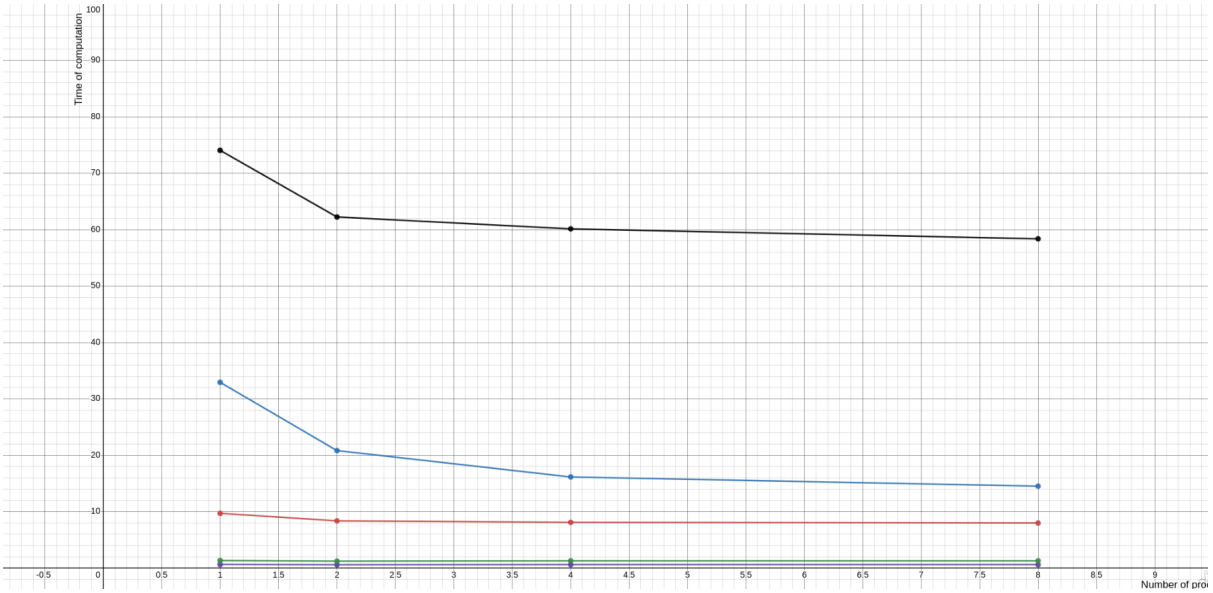
8	0.608	58.380	7.988	14.522	1.298

Table 2 - Running time(in second) for approach 3

We will make the graph for above table -

On X-axis there is number of processors

On Y-axis runtime(in seconds) is shown



Graph 2 - Run time analysis for approach 3

In the graph purple line is for test case 2, black is for test case 3, red line is for test case 5, light blue is for test case 6, green line is for test case 8.

Observation and explanation -

We observe a considerable decrease in runtime in large test case. This was expected because we are distributing the work load from 1 processor to a number of Processor hence runtime decreases. But in small test cases we can see that runtime is not varying much. This happens because overhead in parallelising is large.

2. <u>Speedup</u>

Here we will be doing speedup analysis for our distributed algorithm(approach 3) in tabulated form.

In the table np denotes number of processors on which we are running the code.

We have calculated the speedup for test case 2, 3, 5, 6 and 8 using the data from table 2.

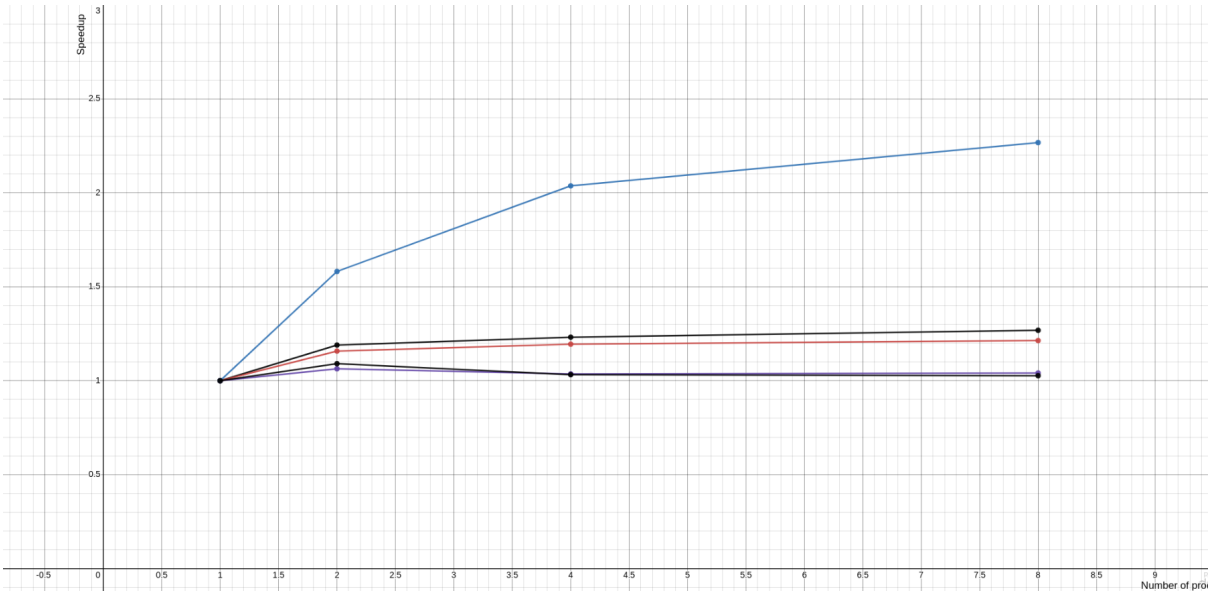
np	Test 2	Test 3	Test 5	Test 6	Test 8
1	1	1	1	1	1
2	1.06386	1.18981	1.15785	1.58175	1.09083
4	1.03601	1.23152	1.19495	2.03736	1.03253
8	1.04112	1.26848	1.21395	2.26773	1.02696

Table 3 - Speedup analysis for approach 3

We will make the graph for above table -

On X-axis there is number of processors

On Y-axis speedup(in unit) is shown



Graph 2 - Speedup analysis for approach 3

In the graph purple line is for test case 2, black is for test case 3, red line is for test case 5, light blue is for test case 6, green line is for test case 8.

Observation and explanation -

We observe a considerable speedup in large test cases.This also suggest the fact that our program is well-scalable. The considerable speedup was due to the fact that we are distributing the work load from 1 processor to a number of Processor hence speedup decreases. But in small test cases we can see that runtime is not varying much. This happens because overhead in parallelising is large.

3. <u>Efficiency</u>

Here we will be doing efficiency analysis for our distributed algorithm(approach 4) in tabulated form.

In the table np denotes number of processors on which we are running the code.

We have calculated the efficiency for test case 2, 3, 5, 6 and 8 using the data from table 3.

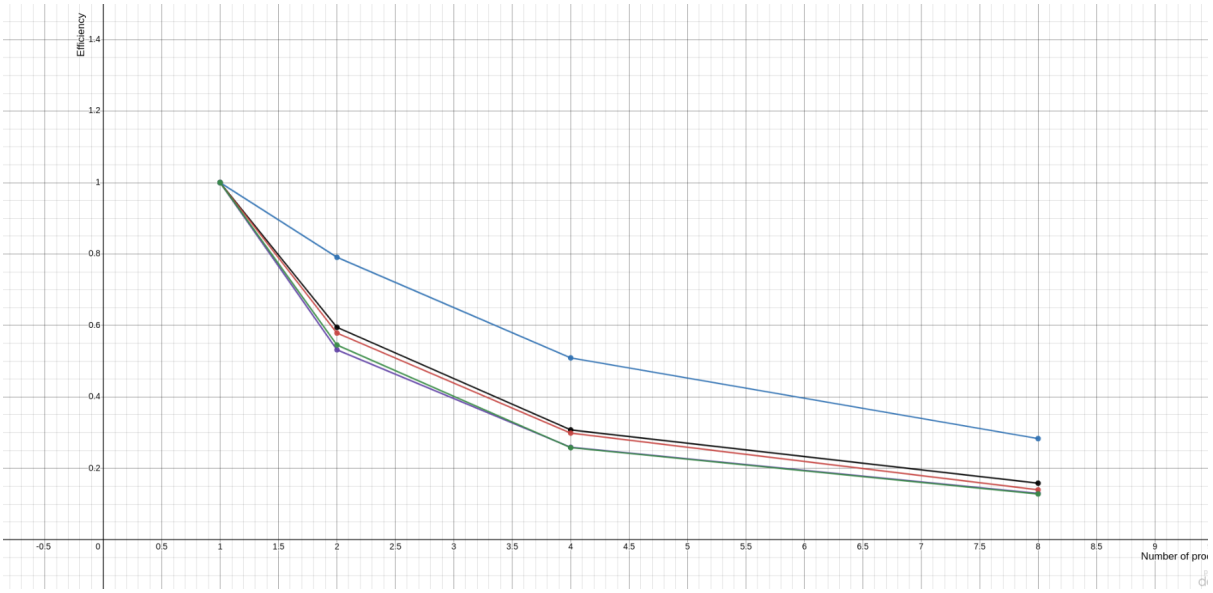
np	Test 2	Test 3	Test 5	Test 6	Test 8
1	1	1	1	1	1
2	0.53193	0.594905	0.57892	0.79087	0.54542
4	0.2590025	0.30788	0.29873	0.50934	0.25813
8	0.13014	0.15856	0.14017	0.28347	0.12837

Table 4 - Efficiency analysis for approach 3

We will make the graph for above table -

On X-axis there is number of processors

On Y-axis efficiency is shown



Graph 3 - Efficiency analysis for approach 3

In the graph purple line is for test case 2, black is for test case 3, red line is for test case 5, light blue is for test case 6, green line is for test case 8.

Observation and explanation -

We observe that efficiency is average when number of processors is small. As number of processor increases smaller and smaller unit of work is distributed among them but the overhead due to synchronization increases sharply. Hence the efficiency decreases as the number of processor increases.