Assignment 3

Sreemanti Dey
Abhinav Barnawal

April 7, 2023

1 Task 1

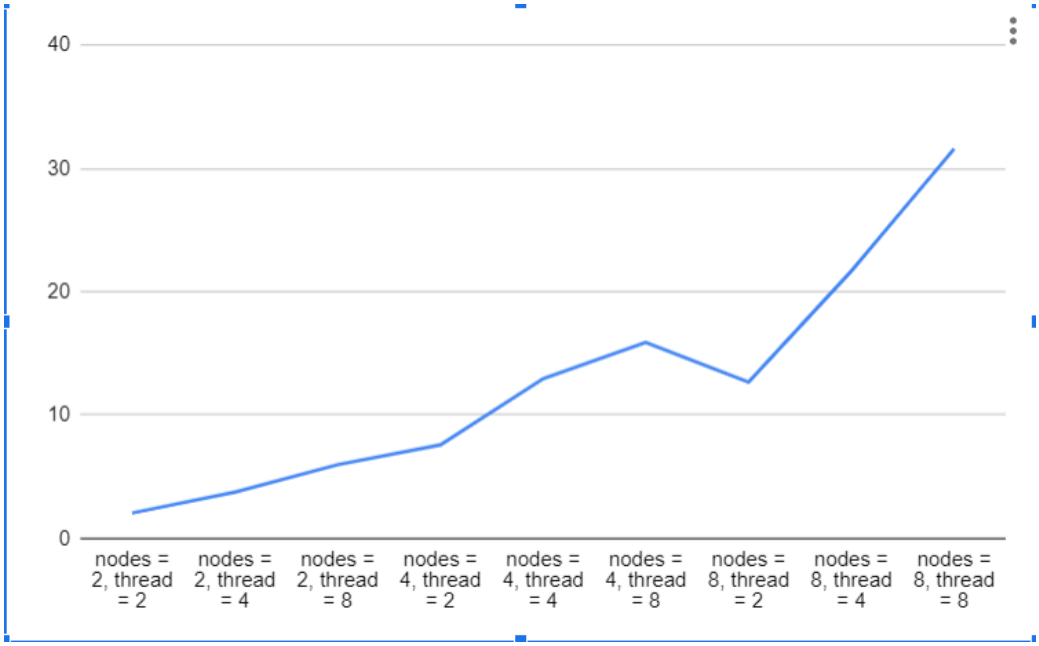
1.1 Detailed approach for task 1

We employed the following approach to implement and parallelise task 1 using multiple nodes and threads:

1. Divide the graph's vertices among all nodes and all threads over each node. Then, to ensure an almost equal division of the load, assign edges using the following algorithm:
   For each edge $\{i, j\}$,
   - If i is even and j is odd or vice-versa, assign the edge to the rank of the even node.
   - Else, if both are even, assign the edge to the rank of the smaller node.
   - Else, both are odd, in which case assign the edge to the rank of the larger node.
2. Then, for every edge, store the complete neighbours of its end-points and pre-calculate the number of triangles this edge is a part of (or equivalently, the number of the common neighbours of its end-points).
3. Perform edge deletion-based k-truss finding algorithm for each k independently over each node thread. At the end of each iteration, when it is time to share the deleted edges over each rank, aggregate the deleted edges of each thread and share with all other nodes using MPI_Allgather collective communication.
4. Repeat steps 2 and 3 until no node has deleted an edge.
5. Then, Breadth-First-Search over each node independently setting the head node for each vertex of every connected component. Share these edges with rank 0, which employs a modified version of the Union-Find algorithm to find overall connected components.

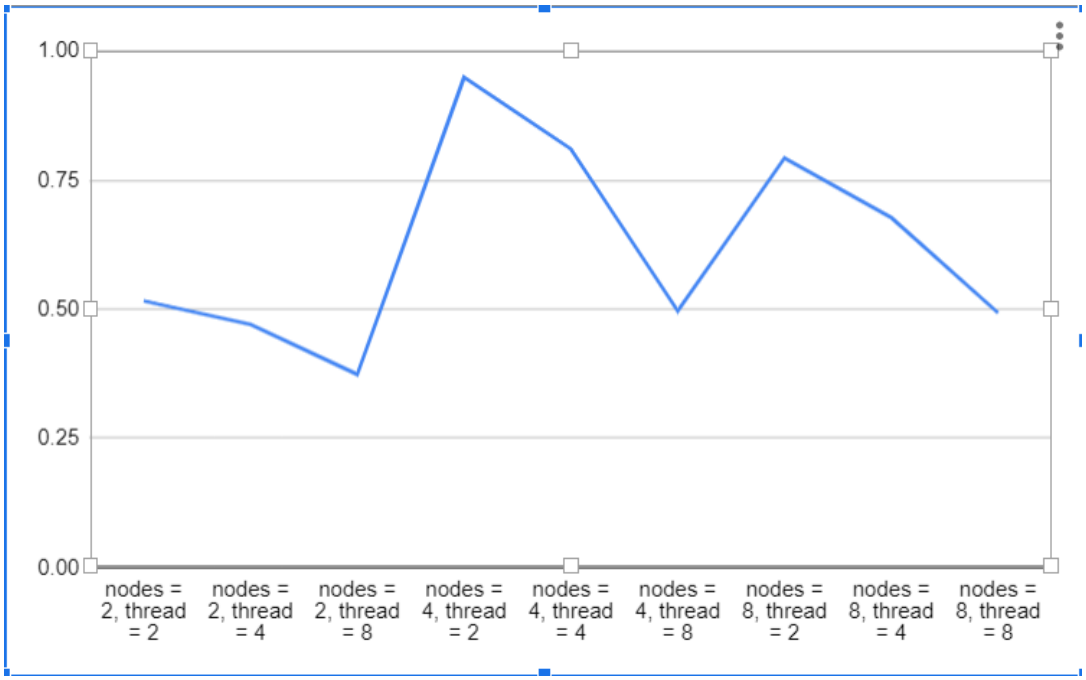
1.2 Plot Graph for speedup and efficiency for different core counts

Verbose 0 estimates:



Speedup observed vs number of cores

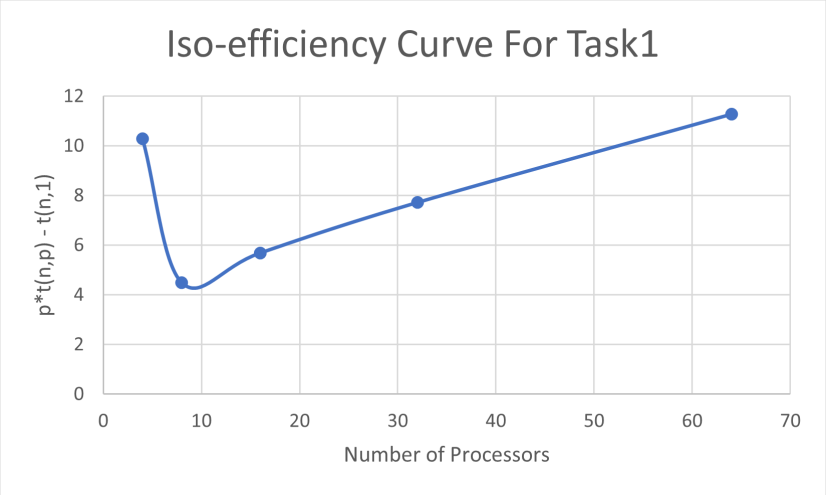
We see an almost linear increase in speedup, only a minor drop from (node=4, thread=8) to (node=8, thread=2) which is there because number of cores in second case is half of that of first case.



Efficiency observed vs number of cores

In calculation for efficiency, we have used p as number of cores which is number of nodes * number of threads per node. We see higher speedup in case of increasing nodes, keeping threads constant than increasing threads, keeping nodes constant, thus we see quite a large rise in efficiency from nodes=2, thread=2 to nodes=4,thread=2, but not so much with increasing threads, keeping nodes constant. This is because when we increase nodes, we see the work getting distributed among multiple nodes which have their own caches thus, with increase in resources available, provided sequential fraction is not too high, we see this change. Hence this curve is justified.

1.3 Value of iso-efficiency and its explanation



Iso-efficiency variation for task 1 with the number of processes

From the graph, it can be deduced that the iso-efficiency is $I(p) = \Omega(plog(p))$ because the curve for $y = nlogn$ resembles the plot. We have plotted the above using the equation $I(n,p) = \Omega(p \times t(n,p) - t(n,1))$. Since the size of our problem is constant for these analyses, $I(p) = \Omega(pt(p) - t(1))$. Here, p is the number of processes in use which is the number of nodes times the number of threads on each node. Since the time taken for, let's say, 4 threads and 4 nodes is different than that for 2 threads and 8 nodes (because of the differential speed-ups with threads and nodes), we have taken the average of these times for getting the time taken for 16 processors.

1.4 Estimation of sequential fraction

We have used Karp-Flatt metric for computing the sequential fraction. The formula is as follows:

f = $\frac{\frac{1}{Speedup} - \frac{1}{p}}{1 - \frac{1}{p}}$

where p is number of cores, here p = number of nodes * number of threads per node. So, we have computed the sequential fraction for every value of speedup with corresponding number of cores and we took the average of that as the sequential fraction estimation. This comes out to be 0.07667444109 for task1. This sequential fraction is very less in task 1 verbose 0, since we have effectively parallelized the entire task performed on 1 rank into all threads, only synchronizing when we need to gather or send information to other ranks, we do this communication among ranks via thread 0. Thus, we have around 7.6% of sequential fraction in our code.

1.5 Justification of why task1 is scalable

As seen in the efficiency versus nodes and threads curve, the efficiency is not monotonically decreasing with the number of threads and processes. This means the algorithm's overhead will not likely overtake the speed-ups due to multiple nodes and threads. We have distributed the computation of deletion of edges and decrement of triangles for each edge among different nodes and threads, only communication is when we need to send or gather information. Thus the sequential fraction being quite less, enables us to have a scalable task 1. Further, the Iso-efficiency suggests that the increase of overhead is $\Omega(plogp)$, which suggests that our code is weakly scalable since overhead does increase with number of processors, thus with increasing problem size, we shall find constant efficiency.

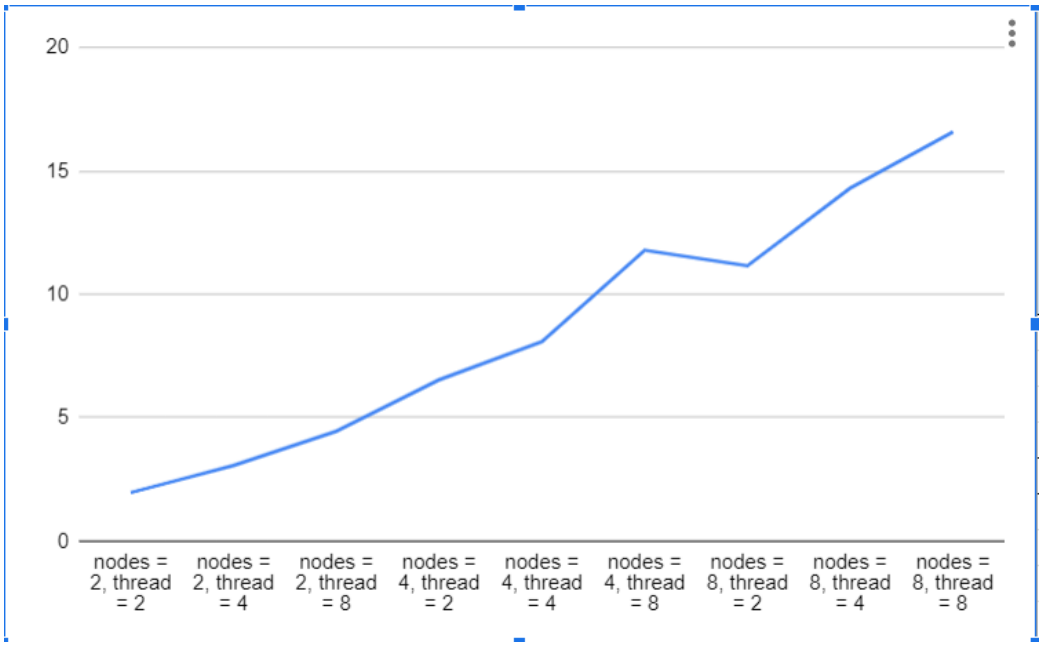
2 Task 2

2.1 Detailed approach for task 2

We perform all the operations of task 1 as described above. After we have the final connected components, we iterate over each vertex in the graph and check the number of connected components its neighbours belong to by storing the heads of its neighbours. If this number exceeds or equals p, we store this vertex in our influencer set. If a vertex already belongs to a connected component, we treat that whole component as one k-truss reachable by v.

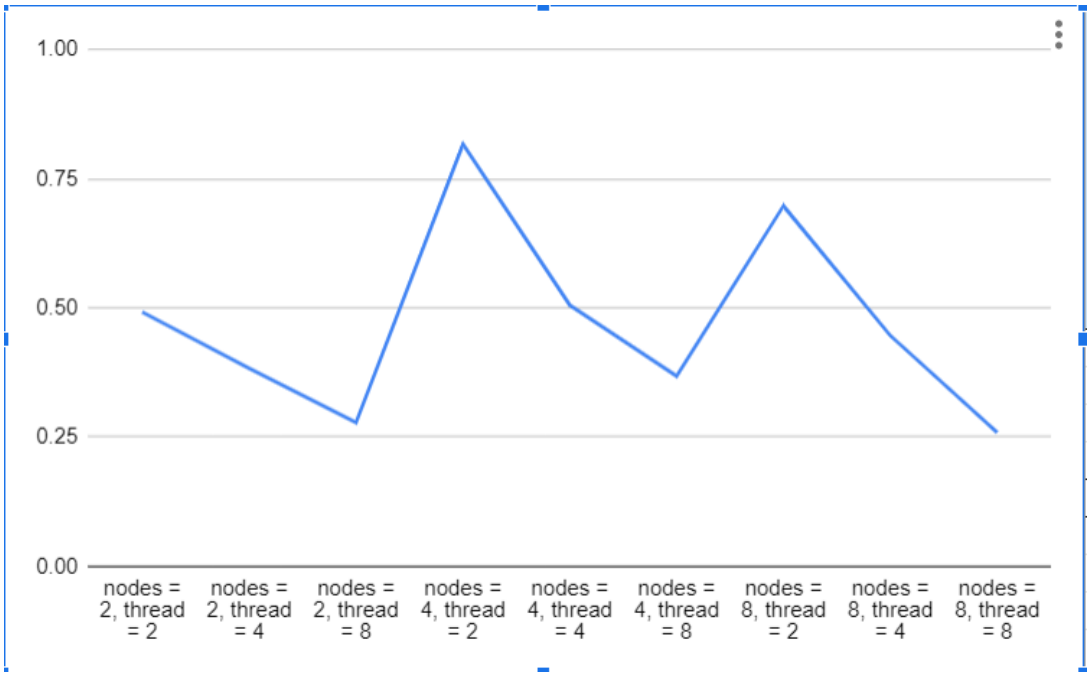
2.2 Plot Graph for speedup and efficiency for different core counts

Verbose 0 estimates:



Speedup observed vs number of cores

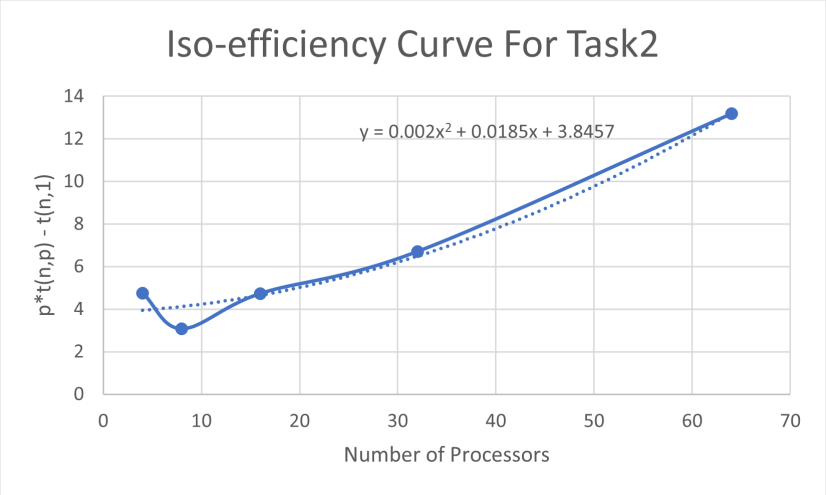
We see an almost linear increase in speedup, only a minor drop from (node=4, thread=8) to (node=8, thread=2) which is there because number of cores in second case is half of that of first case.



Efficiency observed vs number of cores

In calculation for efficiency, we have used p as number of cores which is number of nodes * number of threads per node. We see higher speedup in case of increasing nodes, keeping threads constant than increasing threads, keeping nodes constant, thus we see quite a large rise in efficiency from nodes=2, thread=2 to nodes=4,thread=2, but not so much with increasing threads, keeping nodes constant. Hence this curve is justified.

2.3 Value of iso-efficiency and its explanation



From the graph, it can be deduced that the iso-efficiency is $I(p) = \Omega(p^2)$ because the best fit for the curve has degree 2. Using the fact that iso-efficiency of task 1 is $\Omega(plogp)$, the iso-efficiency $\Omega(p^2)$ is justified because this part additionally involves parallel BFS and hence the communication among the connected components at each rank increase the sequential overhead. We have evaluated the iso-efficiency using the average times for each number of processors (as described in the iso-efficiency of task 1).

2.4 Estimation of sequential fraction

We have used Karp-Flatt metric for computing the sequential fraction. The formula is as follows:

f = $\frac{\frac{1}{Speedup} - \frac{1}{p}}{1 - \frac{1}{p}}$

where p is number of cores, here p = number of nodes * number of threads per node. So, we have computed the sequential fraction for every value of speedup with corresponding number of cores and we took the average of that as the sequential fraction estimation. This comes out to be 0.1127161854 for task2. This sequential fraction is more in task 2 verbose 0, since though we have effectively parallelized the entire task performed on one rank into all threads, only synchronizing when we need to gather or send information to other ranks and we do this communication among ranks via thread 0, the bfs portion when we get all the connected components on 1 rank and print that to the file, that requires some communication and is done primarily by 1 thread hence this increases the sequential fraction. Thus, we have around 11.2% of sequential fraction in our code for task2.

2.5 Justification of why task2 is scalable

The iso-efficiency suggests that the overhead is $\Omega(p^2)$, thus with increasing processors, our efficiency decreases due to the overhead but increasing problem size will counter that impact. Our code is scalable since, task 1 is scalable as justified above and final BFS is done parallely at each node, thus increasing nodes will distribute the computation well since we only communicate the final connected components and use union find to combine them together thus this is a small sequential fraction. Hence, we can say our code is weakly scalable. Also, with increasing problem size, we shall have more distribution of work among cores, thus our sequential fraction being quite small, we shall be able to achieve quite some scalability.

3 Table of values in the format shared

Task_id	1	1	2	2	1	2	1	2
Metric	Speedup	Efficiency	Speedup	Efficiency	Iso-efficiency	Iso-efficiency	Sequential Fraction	Sequential Fraction
nodes = 2, thread = 2	2.064360458	0.5160901144	1.967667407	0.4919168517	Omega(plogp)	Omega(p^2)	0.07667444109	0.1127161854
nodes = 2, thread = 4	3.76533345	0.4706666813	3.063993813	0.3829992267				
nodes = 2, thread = 8	5.980210689	0.373763168	4.451783913	0.2782364945				
nodes = 4, thread = 2	7.591551341	0.9489439177	6.536058695	0.8170073369				
nodes = 4, thread = 4	12.9617843	0.810111519	8.079070981	0.5049419363				
nodes = 4, thread = 8	15.8988653	0.4968395406	11.77682731	0.3680258533				
nodes = 8, thread = 2	12.673421	0.7920888125	11.15544784	0.6972154901				
nodes = 8, thread = 4	21.675346	0.6773545625	14.29250433	0.4466407604				
nodes = 8, thread = 8	31.56303037	0.4931723495	16.57633375	0.2590052148				

The values of speed-ups and efficiencies for different numbers of nodes and threads for each task.

4 References

1. Truss Decomposition naive algorithm from the PDF given
2. Slides shared