INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 2

APAR AHUJA | ENTRY NO. 2019CS10465

AVANATIKA AGARWAL | ENTRY NO. 2019CS10338

Course - COL380 | Prof. Subodh Kumar

March 14, 2023

Contents

Chapter 1

Chapter 1

1.1 Approach Used

We implement the truss computation algorithm described in the assignment statement to compute the truss decomposition of the given graph. We use the PreFilter step to first remove vertices which have degree smaller than k_1-1. This can potentially significantly reduce the size of the graph on which we run the truss computation algorithm. The algorithm used is now described in detail:

1. Divide the graph between the various MPI ranks using the process described in Section 1.1.1.
2. Run the PreFilter algorithm as described in Section 1.1.2 to remove vertices with low degree.
3. Run the TriangleEnumeration algorithm as described in Section 1.1.3 to compute the support for each edge.
4. Run the truss computation algorithm as described in Section 1.1.4 to find the edges belonging to k-truss in this graph.
5. Use the union-find data structure to compute connected components for k-truss as described in Section 1.1.5.
6. Repeat steps 4 and 5 for k ranging from $k_1 + 2$ to $k_2 + 2$.

Note that all the above steps are implemented in a distributed manner, so the entire graph is never stored on a single MPI rank. We now describe each step of the algorithm in detail.

1.1.1 Distributing the Graph

In this step, we distribute the graph between various MPI ranks in order to implement truss decomposition in a distributed manner. The graph is distributed between the various MPI ranks using a degree-based ordering. Let the input graph be $G = (V, E)$. Let $deg(u)$ be the degree of a vertex $u \in G$. Then the graph is distributed as follows:

1. Arrange the vertices in increasing order of their degrees. For vertices with equal degree, arrange them in ascending lexicographic order. Let the total order so created be $\prec$.

2. Divide the arranged vertices in cyclic order between the different MPI ranks. Let $rank(u)$ be the rank of the process which now owns the vertex u.
3. For each vertex v assigned to an MPI rank, send the adjacency list of v to this rank.
4. For an edge $e = (u, v) \in E$, assign the edge e to the process $rank(u)$ if $u \prec v$ and to process $rank(v)$ otherwise.

Note that while each process owns the complete adjacency list of each vertex that it owns, an edge is only assigned to one of the two processes owning the end points. The adjacency list is used for the PreFilter algorithm in Section 1.1.2. While implementing the PropTruss algorithm, the truss value of an edge is only computed by the rank to which this edge is assigned in Step 4. This distribution of vertices and edges is based on a similar division used for the Hybrid algorithm for truss decomposition. Note that the authors of Hybrid algorithm do a random allocation of each vertex to the MPI ranks, we instead do a cyclic allocation in order to simplify implementation.

Advantages of Our Implementation

1. Doing a cyclic allocation is much easier to implement than doing a uniformly random allocation.
2. Sending the adacency list of each vertex owned by a process allows us to run the PreFilter step, which could not have been run if this list had not been shared.

1.1.2 Removing Vertices with Low Degree

In this step, we remove vertices with degree smaller than k_1 in the input graph, to reduce the size of the graph on which we compute truss decomposition. Removing vertices in this manner significantly improves performance since it removes multiple edges from the graph thus greatly reducing the amount of computation we need to perform later. These vertices are removed as follows:

1. Set Deletable $= \{v \in V \mid deg(v) < k_1 - 1\}$.
2. Set $deg(v) = -1$ for all vertices $v \in$ Deletable, to denote that this vertex has been deleted.
3. For every vertex in Deletable, inform the owners of vertices in its adjacency list about the deletion of this vertex. This is done using the MPI_Alltoallv primitive to ensure synchronization across vertices.
4. Using the message received in MPI_Alltoallv, update the degrees of vertices owned by yourself.
5. Update Deletable to contain vertices which have degree smaller than $k_1 - 1$ and are not deleted yet.

6. Use MPI_Allreduce to determine whether Deletable is empty for every process. If so, go to Step 7. Else go to Step 2.
7. Use MPI_Allreduce to communicate the minimum value of degree for each vertex to all the ranks. This is used to determine which vertices have been deleted and the correct degrees of all remaining vertices.
8. Remove the vertices which have degree -1. Iterate over the adjacency lists of all the remaining vertices and remove all the vertices from this list whose degree is -1. This gives the updated adjacency lists of the modified graph.

This algorithm is a distributed implementation of the PreFilter algorithm discussed in the assignment 2 problem statement.

Advantages of Our Implementation

1. We do not immediately modify the adjacency lists of vertices when some vertices are deleted, instead we only update the degrees. By doing so at the end, we are able to avoid multiple iterations over the adjacency list.
2. Using MPI collectives such as MPI_Alltoallv and MPI_Allreduce allow us to synchronize the processes without using barriers.
3. Use of collectives ensures we do not have to keep track of multiple sends and receives, and we do not run into issues related to deadlock due to low network capacity.

1.1.3 Computing Support for Edges

In this step, we compute the supporting vertices for each edge $e = (u, v)$ assigned to a rank. The triangle enumeration algorithm makes use of the degree based distribution of vertices between ranks. For ease of description, we call a pair of edges $e_1 = (u, v)$ and $e_2 = (u, w)$ a monotone wedge if $u \prec v$ and $u \prec w$. Then the algorithm works as follows:

1. Initialize counters cnt1 and cnt2 to 0.
2. Iterate over all edges assigned to the process to find a monotone wedge (say) $e_1 = (u, v)$ and $e_2 = (u, w)$.
3. Send a message to $rank(v)$ if $v \prec w$ else to $rank(w)$, to ask whether (v, w) is an edge.
4. Use MPI_Iprobe to check if you have received any messages. If so, keep receiving all these messages and process them as follows:
   (a) If the message is of type FINISH_1 then increment cnt1.
   (b) If the message is of type FINISH_2 then increment cnt2.

(c) If the message is asking about the existence of some edge, check the same and reply if the edge exists.

(d) If the message is answering a previously asked question by you, update the support of the monotone wedge for which the question was asked.

5. Send FINISH_1 to all processes after the iteration over all monotone wedges is finished. This means that you are done asking all questions about existence of edges.

6. Keep receiving messages from other processes and process them as per Step 4. If cnt1 becomes equal to num_ranks - 1 then break out of this loop.

7. Send FINISH_2 to all processes. This means that you are done answering all the questions that others asked you.

8. Keep receiving messages from other processes and process them as per Step 4. If cnt2 becomes equal to num_ranks - 1 then break out of this loop. This means that now every process is done answering all the questions and you have received all the responses meant for you, so your triangle enumeration is done.

This algorithm is a distributed implementation of the triangle enumeration algorithm used in the Hybrid algorithm for truss decomposition.

Advantages of Our Implementation

1. Using the FINISH2 message gives us the flexibility to only reply to a question about the existence of an edge if the edge exists. This significantly reduces unnecessary communication overhead related to informing about the non-existence of an edge.

2. We use Iprobe after sending every message to completely empty out the receive buffer of the process. This ensures that other processes do not get stuck on blocking send calls due to lack of buffer space.

1.1.4 Computing the k-Truss

We use the FilterEdges algorithm described in the assignment statement to compute the k-truss. Our distributed implementation is as follows:

1. Set Deletable = $\{e \in E \mid supp(e) < k - 1\}$.

2. For every edge in Deletable, inform the owners of edges which support this edge, about the deletion of this edge. This is done using the MPI_Alltoallv primitive to ensure synchronization across vertices.

3. Using the message received in MPI_Alltoallv, update the support of edges owned by yourself. This is done by removing the vertex from support, which is not common between your edge and the deleted edge.

4. Update Deletable to contain edges which have support smaller than $k-1$ and are not deleted yet.

5. Use MPI_Allreduce to determine whether Deletable is empty for every process. If not, go to Step 2.

Advantages of Our Implementation

1. Using MPI collectives such as MPI_Alltoallv and MPI_Allreduce allow us to synchronize the processes without using barriers.

2. Use of collectives ensures we do not have to keep track of multiple sends and receives, and we do not run into issues related to deadlock due to low network capacity.

3. The processes do not need to update their adjacency lists in order to be able to compute connected components later.

1.1.5 Computing Connected Components

After we have determined which edges are part of the k-truss, we need to compute connected components in the graph to determine the groups. This step is implemented sequentially for ease of implementation, with only process working at one time. However, the graph is distributed across processes and is never stored fully in a single process. We use the union-find data structure to compute connected components as follows:

1. Create a vector to store parent of every vertex and initialize it so that parent of every vertex is itself.

2. If rank = 0 then iterate over all edges $e = (u, v)$ owned by the process. Make the vertex $find(u)$ point to $find(v)$. Send the parent vector to the next process.

3. If rank $>$ 0 and rank $<$ numranks - 1 then receive parent vector from the previous process. Iterate over all edges $e = (u, v)$ owned by the process. Make the vertex $find(u)$ point to $find(v)$. Send the parent vector to the next process.

4. If rank = numranks - 1, then receive parent vector from the previous process. Iterate over all edges $e = (u, v)$ owned by the process. Make the vertex $find(u)$ point to $find(v)$.

5. The process with rank = numranks - 1 uses this parent vector and iterates over all the vertices, to put vertices having same root in one component.

Advantages of Our Implementation

1. The entire graph is never collected at a single process to compute connected components. The computation is distributed across the processes (though conducted sequentially by each process after the previous one).
2. We do not need to ensure that every process knows the adjacency list of all the vertices it owns, unlike some other distributed algorithms to compute connected components.

1.1.6 Observations

After running the code on the given test-cases, we made the following observations:

1. The maximum amount of time is taken in the TriangleEnumeration algorithm from Section 1.1.3 from among all the algorithms used in our implementation.
2. The number of iterations in computing k-truss using the FilterEdges algorithm (Section 1.1.4) is higher than what we would have gotten for the PropTruss algorithm (Section 1.2.1). However the computational load in each iteration is low, so the communication network will not get congested.
3. While the PreFilter algorithm (Section 1.1.2) was implemented to reduce the size of the graph by removing vertices of low degree, most of the given test-cases give very low values of k_1 and therefore PreFilter does not reduce the size of the graph in these cases. However when the value of k_1 is large, many vertices get removed from the graph which have low degree.
4. Since TriangleEnumeration takes the most time to run, we do not perform PreFilter for every value of k when computing k-truss, to avoid redoing triangle enumeration. PreFilter and TriangleEnumeration are therefore instead run only at the beginning of the algorithm.

1.2 Approach 2 (PropTruss)

We try to implement the PropTruss algorithm to compute the truss decomposition of the given graph. However we were not able to debug this code within the given time to be able to compute correct output. We have described the pseudocode for our distributed implementation of PropTruss that we tried to implement. This algorithm is used because of the low number of iterations, even though it has a much higher computational load on each MPI rank. To further optimize the PropTruss algorithm, we do not run it on the entire graph, instead we use the PreFilter step to first remove vertices which have degree smaller than k_1-1. This can potentially significantly reduce the size of the graph on which we run PropTruss. The algorithm used is now described in detail:

1. Divide the graph between the various MPI ranks using the process described in Section 1.1.1.
2. Run the PreFilter algorithm as described in Section 1.1.2 to remove vertices with low degree.

3. Run the TriangleEnumeration algorithm as described in Section 1.1.3 to compute the support for each edge.
4. Run the PropTruss algorithm as described in Section 1.2.1 to compute the truss decomposition of the smaller graph.
5. Use the union-find data structure to compute connected components for k-truss as described in Section 1.1.5.

Note that all the above steps are implemented in a distributed manner, so the entire graph is never stored on a single MPI rank.

Note: The file ZPropTruss.cpp contains our code attempt for implementing the PropTruss algorithm.

1.2.1 Computing the Truss Values

This is the primary algorithm for the given assignment task, which computes the truss decomposition for the smaller graph obtained after running the PreFilter algorithm from Section 1.1.2. We implement the PropTruss algorithm in a distributed manner due to the low number of iterations it uses. The distributed implementation of PropTruss algorithm is now described in detail:

1. Initialize counter cnt to 0.
2. For each triangle $\Delta(u, v, w)$, set $\hat{\tau}(u, v, w) \leftarrow \infty$.
3. For each edge $e = (u, v)$
   (a) Set $\hat{\tau}(e) \leftarrow supp(e) + 2$.
   (b) Set $g_e \leftarrow supp(e)$. This is the number of triangles Δ with $\hat{\tau}(\Delta)$ at least $\hat{\tau}(e)$.
   (c) For $0 \leq j < \hat{\tau}(e)$, set $h_e(j) \leftarrow 0$. $h_e(j)$ stores the number of triangles Δ with $\hat{\tau}(\Delta) = j$.
4. Set the active set Act $= \{e = (u, v)\}$.
5. Iterate over all edges $e = (u, v)$ in Act and over all triangles $\Delta = (u, v, w)$ supporting e. If $\hat{\tau}(e) < \hat{\tau}(u, v, w)$ then
   (a) Set $\hat{\tau}(u, v, w) \leftarrow \hat{\tau}(e)$ and $val_{new} \leftarrow \hat{\tau}(u, v, w)$.
   (b) Send messages to owners of (v, w) and (u, w) identifying $((u, v, w), edge, val_{new})$.
6. Check if you have received any messages. If so, keep receiving these messages and process them as follows:
   (a) If message is of type FINISH then increment cnt.
   (b) Otherwise let data $= (\Delta(u, v, w), (v, w), val_{new})$. Set $val_{old} = \hat{\tau}(u, v, w)$.

(c) If $val_{new} < val_{old}$, set $\hat{\tau}(u, v, w) \leftarrow val_{new}$ and call Update$((v, w), val_{old}, val_{new})$. The Update function is described in Section 1.2.2.

7. After the iteration over all edges in Act finishes, send FINISH messages to all processes. This means you have iterated through all of the active set for this iteration.

8. Keep receiving messages from other processes and process them as per Step 6. If cnt becomes equal to num_ranks - 1 then break out of this receive loop. This means that you have made all the required Updates for this iteration.

9. Set Act $\leftarrow \{e \mid \hat{\tau}(e)$ modified because of the update calls$\}$.

10. Use MPI_Allreduce to determine if Act is empty for all ranks. If not, go to Step 5.

1.2.2 Update Function for Edges

This function maintains the values of $\hat{\tau}(e), g_e, h_e(j)$ for each edge $e = (u, v)$. The function takes as input $(e = (u, v), val_{old}, val_{new})$ and works as follows:

1. If $val_{old} \geq \hat{\tau}(e)$ and $val_{new} \geq \hat{\tau}(e)$ then do nothing.

2. Else if $val_{old} \geq \hat{\tau}(e)$ and $val_{new} < \hat{\tau}(e)$ then

   (a) Decrement g_e.

   (b) Increment $h_e(val_{new})$.

3. Else if $val_{old} < \hat{\tau}(e)$ and $val_{new} < \hat{\tau}(e)$ then

   (a) Decrement $h_e(val_{old})$.

   (b) Increment $h_e(val_{new})$.

4. If $g_e < \hat{\tau}(e) - 2$ then

   (a) Decrement $\hat{\tau}(e)$.

   (b) Set $g_e \leftarrow g_e + h_e(\hat{\tau}(e))$

Chapter 2

2.1 Scalability Analysis

We now discuss the speedup and efficiency of our implementation:

Test ID	(V, E)	k-range	1 rank	2 ranks	4 ranks	6 ranks	8 ranks
0	(100, 1489)	(1, 10)	318ms	293ms	291ms	292ms	246ms
1	(1000, 12481)	(1, 10)	335ms	301ms	290ms	287ms	297ms
2	(5000, 49755)	(1, 8)	343ms	342ms	314ms	309ms	310ms
3	(10000, 1250245)	(1, 5)	29100ms	21873ms	13710ms	10392ms	8486ms
4	(15, 59)	(1, 6)	255ms	291ms	235ms	280ms	245ms
5	(25000, 628968)	(1, 8)	3013ms	1737ms	1200ms	991ms	907ms
6	(10000, 55184)	(1, 29)	566ms	506ms	473ms	443ms	426ms
7	(5000, 55184)	(10, 25)	950ms	915ms	862ms	822ms	800ms
8	(100000, 517830)	(2, 6)	1490ms	1090ms	811ms	715ms	688ms

Table 2.1: Runtime comparison for different number of processes

Test ID	(V, E)	k-range	1 rank	2 ranks	4 ranks	6 ranks	8 ranks
0	(100, 1489)	(1, 10)	1	1.08	1.09	1.08	1.29
1	(1000, 12481)	(1, 10)	1	1.11	1.15	1.16	1.12
2	(5000, 49755)	(1, 8)	1	1.003	1.09	1.11	1.10
3	(10000, 1250245)	(1, 5)	1	1.33	2.12	2.8	3.43
4	(15, 59)	(1, 6)	1	0.87	1.085	0.91	1.04
5	(25000, 628968)	(1, 8)	1	1.73	2.51	3.04	3.32
6	(10000, 55184)	(1, 29)	1	1.12	1.19	1.27	1.32
7	(5000, 55184)	(10, 25)	1	1.04	1.1	1.15	1.18
8	(100000, 517830)	(2, 6)	1	1.36	1.83	2.08	2.16

Table 2.2: Speedup comparison for different number of processes

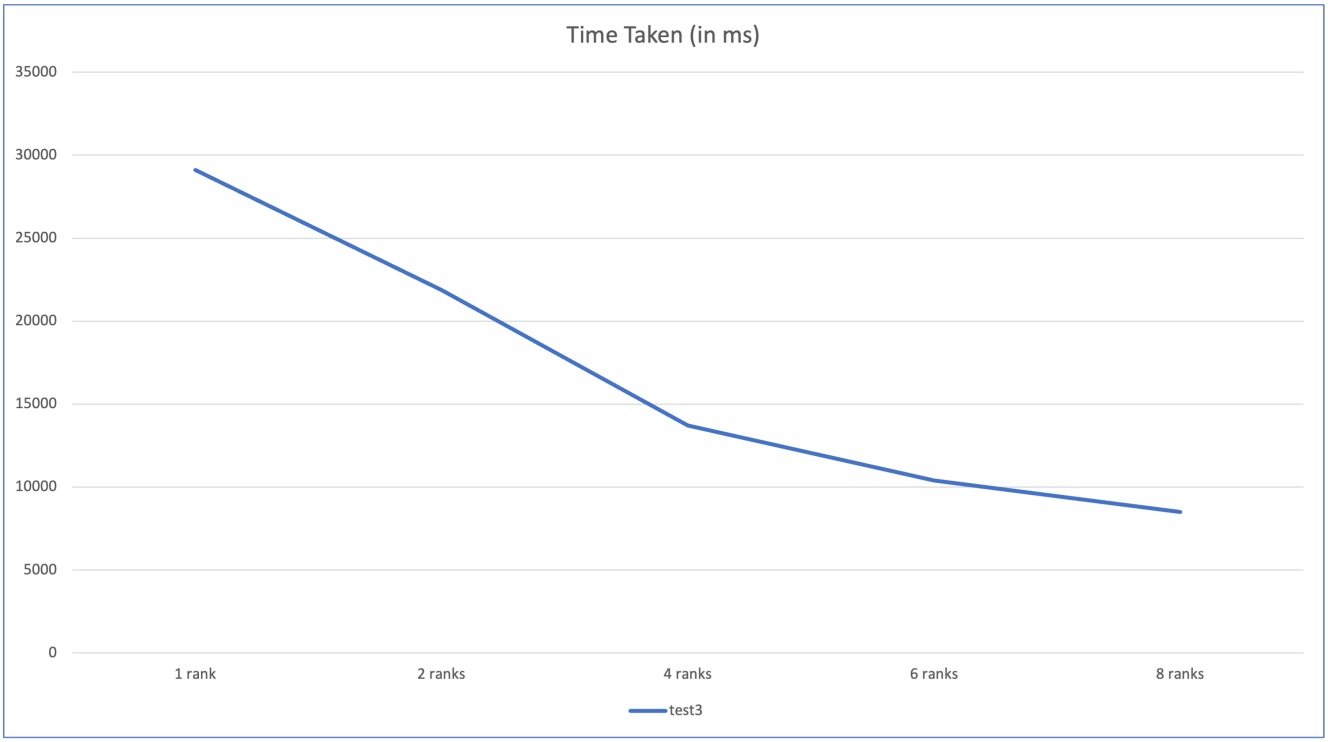

Figure 2.1: Runtime for test3

Test ID	(V, E)	k-range	1 rank	2 ranks	4 ranks	6 ranks	8 ranks
0	(100, 1489)	(1, 10)	1	0.54	0.2725	0.18	0.16125
1	(1000, 12481)	(1, 10)	1	0.555	0.2875	0.1933	0.14
2	(5000, 49755)	(1, 8)	1	0.5015	0.2725	0.185	0.1375
3	(10000, 1250245)	(1, 5)	1	0.665	0.53	0.466	0.428
4	(15, 59)	(1, 6)	1	0.435	0.27125	0.15166	0.13
5	(25000, 628968)	(1, 8)	1	0.865	0.6275	0.506	0.415
6	(10000, 55184)	(1, 29)	1	0.56	0.396	0.211	0.165
7	(5000, 55184)	(10, 25)	1	0.52	0.275	0.1916	0.1475
8	(100000, 517830)	(2, 6)	1	0.68	0.4575	0.3466	0.27

Table 2.3: Efficiency comparison for different number of processes

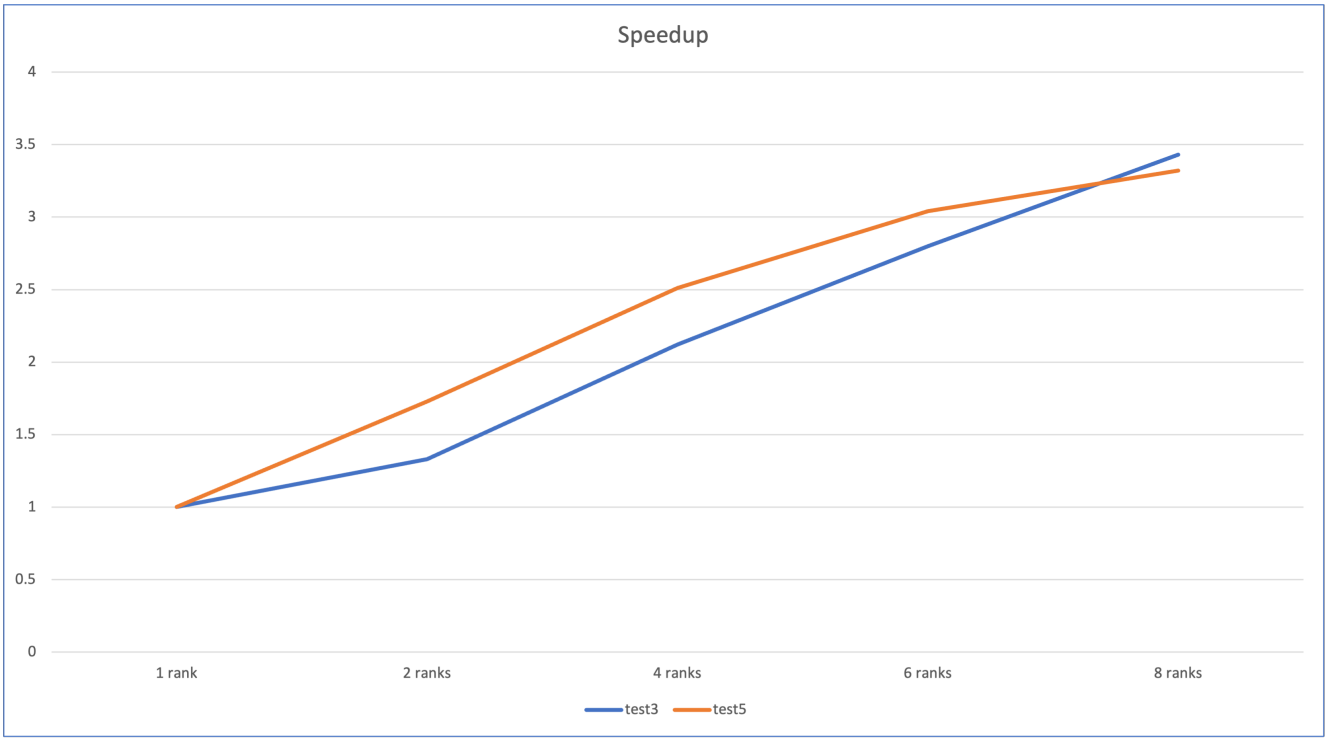

Figure 2.2: Speedup for test3 and test5

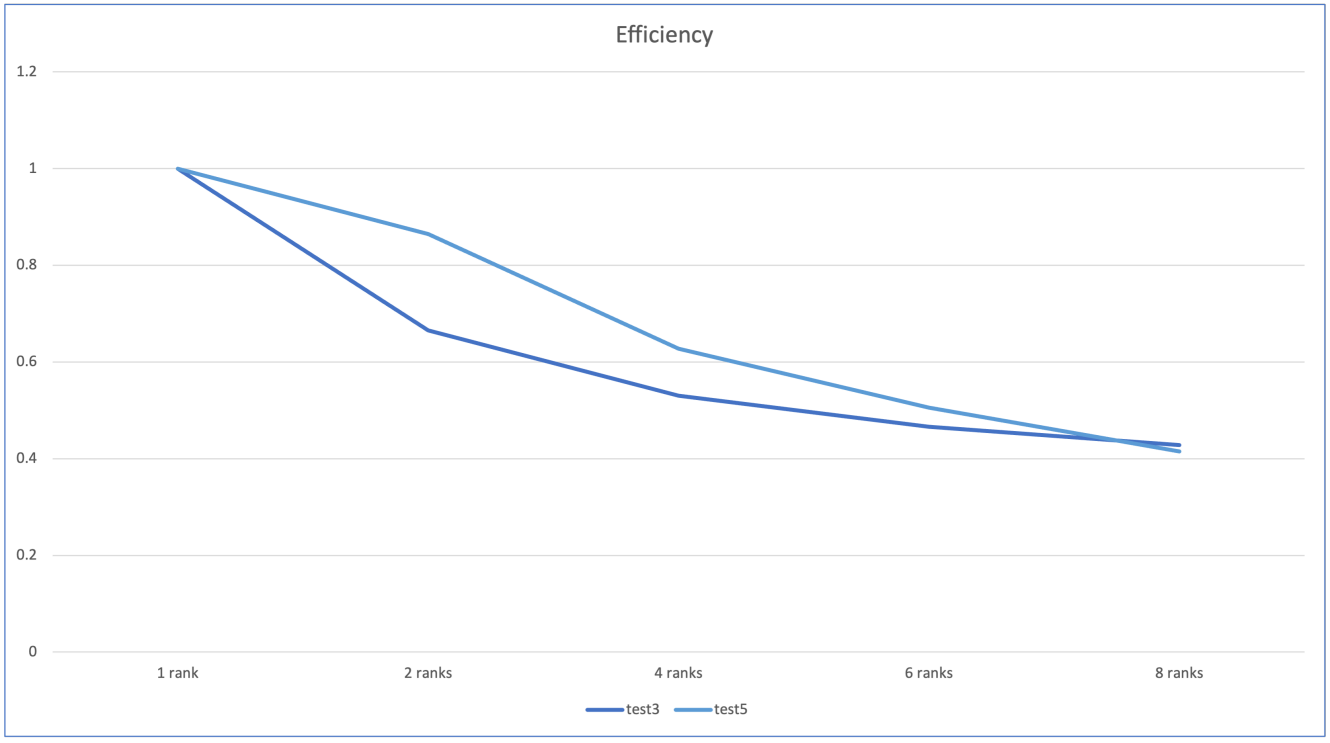

Figure 2.3: Efficiency for test3 and test5