INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 3

APAR AHUJA | ENTRY NO. 2019CS10465

AVANATIKA AGARWAL | ENTRY NO. 2019CS10338

Course - COL380 | Prof. Subodh Kumar

April 7, 2023

Contents

Chapter 1

Chapter 1

1.1 Approach for Task 1

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 in a distributed manner, with processes communicating in groups of 2 to compute partial connected components. 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. Let the two communicating processes be A and B, which have computed partial connected components C_A and C_B. They merge their results as follows (we assume that results are merged in process B, which then participates in further rounds to compute final connected components):
    - For every pair (u, v) of vertices in C_A where v is parent of u, let vertices x and y be the roots of union find structure in C_B of u and v respectively.
    - If x and y are different, then merge the sets rooted at x and y in C_B.
3. The above step is run for $log_2(n)$ iterations, where n is the number of MPI ranks. In each step, the number of active processes is halved, and processes communicate in groups of 2. The final result is computed in process with rank 0.

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 and performed completely in parallel.
2. Due to processes communicating in groups of 2, we only need logarithmic iterations instead of linear in the number of MPI ranks.

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 **??**). 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.

1.1.7 Why is this scalable?

The different MPI ranks divide all the data of the graph between themselves and the edges of the graph are never collected on one MPI rank alone. This data distribution is fairly uniform thus making sure that the work is divided roughly equally between all the MPI ranks. All the steps in the algorithm described previously are performed completely in parallel. In the algorithms described in sections 1.1.1, 1.1.2, 1.1.3 and 1.1.4, the number of iterations for the given data is independent of the number of MPI ranks, and only depends on the size of data owned by a particular MPI rank. Furthermore, the number of iterations for distributed computation of connected components in section 1.1.5 only depends logarithmically on the number of MPI ranks, and not linearly. Thus as the number of MPI ranks increases, the computation performed by each rank for the same amount of data keeps decreasing. The total communication depends on the number of MPI ranks only in Section 1.1.5 (when the final solutions of all ranks are merged), and this dependence is also logarithmic and not linear.

1.2 Approach for Task 2

Once the connected components have been computed at MPI rank 0, it broadcasts this data to every process, so that they can determine influencer vertices from amongst the vertices they own. This is done as follows:

- Rank 0 broadcasts the connected components to all MPI ranks.
- Each rank iterates on the adjacency lists of the vertices it owns and computes the ego network of each of these vertices. If the ego network satisfies the contraints defined in the input, the vertex is marked as an influencer vertex. Otherwise, the ego network of this vertex is deleted to free memory.
- Since the computation of ego networks can be done independently for each vertex, the previous step is parallelized using OpenMP threads, where different threads perform the above computation for different vertices.
- The processes communicate the influencer vertices to rank 0 using MPI_Reduce.
- If verbose = 0, then rank 0 uses this information to generate the output. If verbose = 1, then the processes also send the ego networks of influencer vertices to rank 0, which then writes that to the output file.

1.2.1 Why is this scalable?

This task requires only one major computation, that of ego networks of the vertices. Since the MPI ranks perform this computation independently once the connected components are broadcasted, the computation time is only dependent on the size of data owned by each MPI rank. After the computation of ego networks, all the processes communicate this data to rank 0, which outputs it to a file. This communication size is not dependent on the number of MPI ranks, but instead on the input data itself, hence this communication amount remains same if the input data is same, thus there is no extra communication overhead with increasing MPI ranks.

1.3 Other OpenMP parallelization attempts

We tried to parallelize the TriangleEnumeration algorithm from Section 1.1.3 using OpenMP since this was taking maximum time. We attempted multiple approaches, all of which lead to an increase in runtime with increasing number of threads. We now describe these approaches.

1.3.1 Approach 1

We parallelize the iteration over adjacency list to find monotone wedges and query other processes about the existence of the third edge. Different threads iterate over adjacency lists of different vertices and send messages to the owners of the potential third edge, asking if the edge exists. In addition, to avoid deadlock due to limited buffer resources, thread 0 also empties the receive buffer while iterating over the adjacency lists to find monotone wedges. To make sure that thread 0 does not take too long to finish because of the extra work in its iterations, we use dynamic scheduling of the for loop, so that other threads can perform more iterations.

When this code is run with increasing number of OpenMP threads, the total runtime increases. This potentially occurs because of two reasons:

1. Multiple threads are making concurrent calls to MPI_Send, which leads to a lot of additional synchronization to serialize these calls.
2. Since the multiple threads modify the data structure storing all the triangles, a lock is needed to ensure consistency of the data structure.

To get around these difficulties, we try the next approach.

1.3.2 Approach 2

Instead of all the threads making concurrent MPI_Send calls, the threads now store all their messages in a vector (which is distinct for each thread, so no locks are needed). At the end, a single thread iterates through all the messages stored by all the threads and sends the required query message to every thread. In addition, to avoid deadlock, it also reads one message from the receive buffer after every send call. Once again, the runtime increases when the number of threads increase. This could potentially occur because of the following reasons:

1. The multiple threads spend a lot of time storing all the query messages in the vector, which is equal to the number of monotone wedges. A single thread then iterates over this huge vector and makes both send and receive calls, thus needing a lot of time.
2. A single thread is performing both send and receive in an interleaved manner. Before each receive, it makes an Iprobe call to check if there is a message in the receive buffer. This introduces a lot of time overhead.

To overcome these drawbacks above, we try the next three approaches.

1.3.3 Approaches 3, 4, 5

Approach 3

Instead of one thread making both send and receive calls after storing the monotone wedges, two threads run one of which only makes send calls and the other only makes receive calls. Since the second thread only makes receive calls, it does not need to make an Iprobe call, and it can directly make a blocking receive call waiting for the next message. However, the runtime still increases with high number of threads.

Approach 4

In this approach we do not store the monotone wedges, instead directly send a query message when a thread finds a monotone wedge, similar to Approach 1. However, only half of the threads work on finding monotone wedges, the other half of the threads make receive calls to read the messages so that sending threads do not wait on blocking send calls. However, the runtime still increases with high number of threads.

Approach 5

This is similar to Approach 4, but n-1 threads compute monotone wedges and send query messages, while one thread makes receive calls to read the incoming messages. This potentially reduces the synchronization between several threads reading from the same receive buffer, but still results in an increase in runtime.

Chapter 2

Chapter 2

2.1 Scalability Analysis

We now present plots for speedup and efficiency of our implementation. For computing these values we have used number of cores as number of MPI processes * number of threads. The timing for one core therefore is the runtime with one thread and one MPI process.

2.1.1 Task 1

The data generated is for test3 with $k_1 = 0$ and $k_2 = 3$.

Note that we have computed the overhead in milliseconds and then found the best fit curve to get the isoefficiency function. This function formally is:

$$I(p) = 6831.4 + 2919.2 * \text{number of cores} \tag{2.1}$$

The above values of slope and intercept are in milliseconds. When converted to seconds, the function becomes:

$$I(p) = 6.831 + 2.919 * \text{number of cores} \tag{2.2}$$

In metric.csv, we have reported the value of overhead for isoefficiency, since isoefficiency will be some constant multiplied by overhead.

2.1.2 Task 2

The data generated is for test3.

Note that we have computed the overhead in milliseconds and then found the best fit curve to get the isoefficiency function. This function formally is:

$$I(p) = 3157.2 + 1865.3 * \text{number of cores} \tag{2.3}$$

The above values of slope and intercept are in milliseconds. When converted to seconds, the function becomes:

$$I(p) = 3.157 + 1.865 * \text{number of cores} \tag{2.4}$$

In metric.csv, we have reported the value of overhead for isoefficiency, since isoefficiency will be some constant multiplied by overhead.

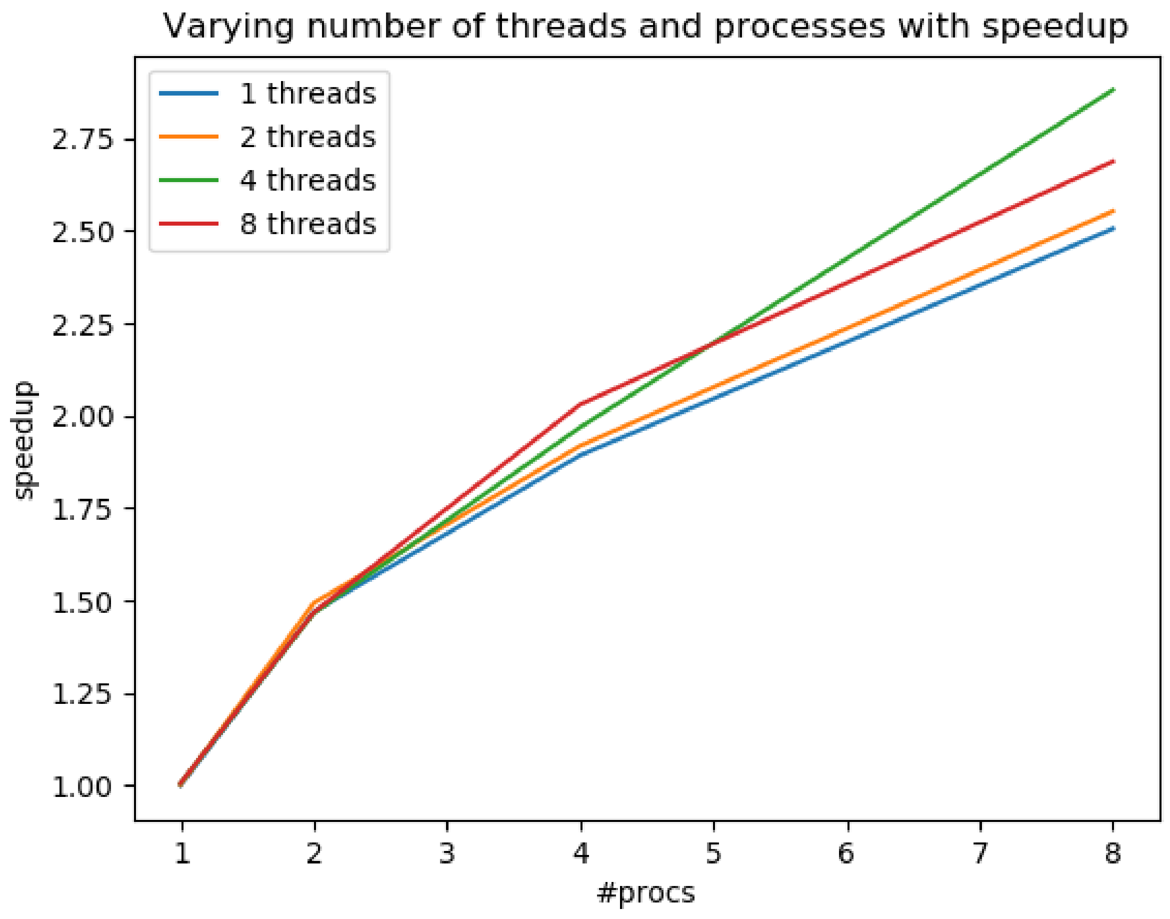

Figure 2.1: Speedup for task 1 on test3

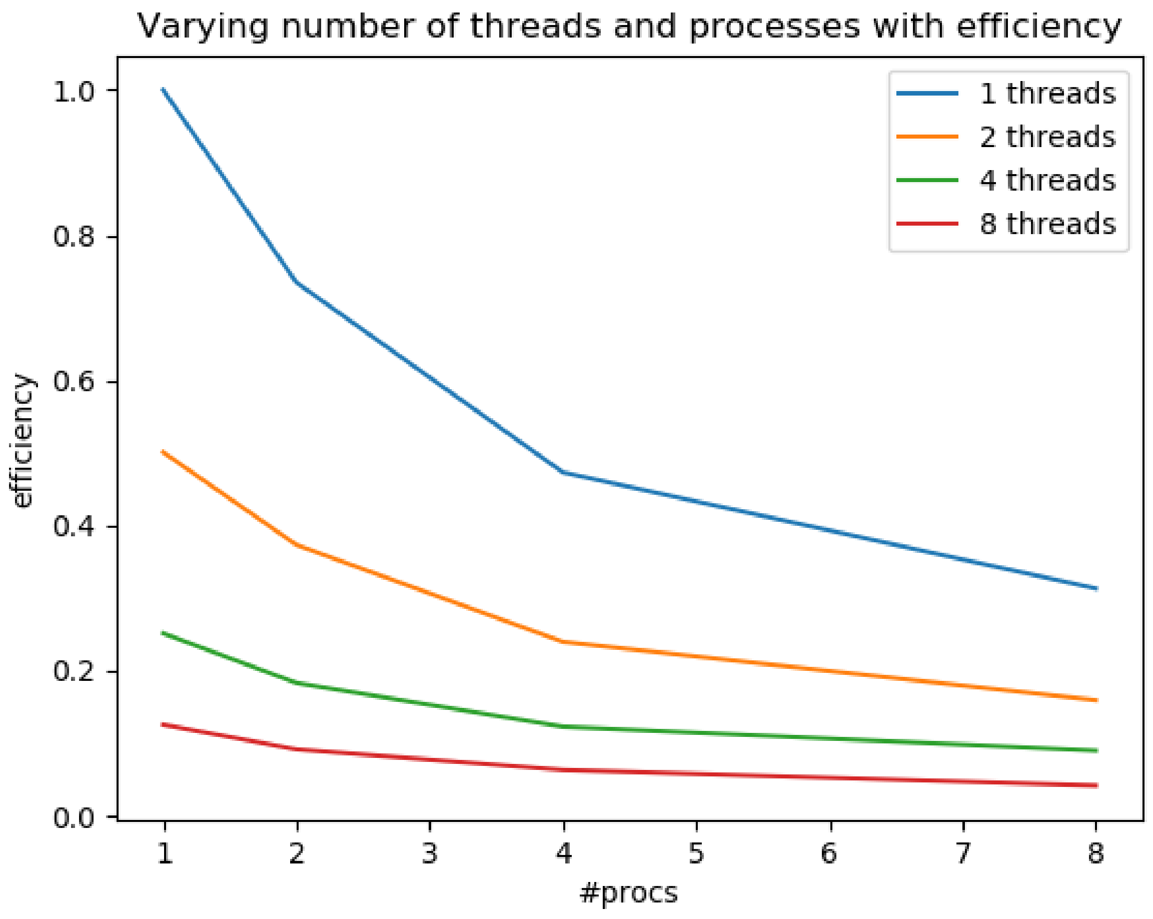

Figure 2.2: Efficiency for task 1 on test3

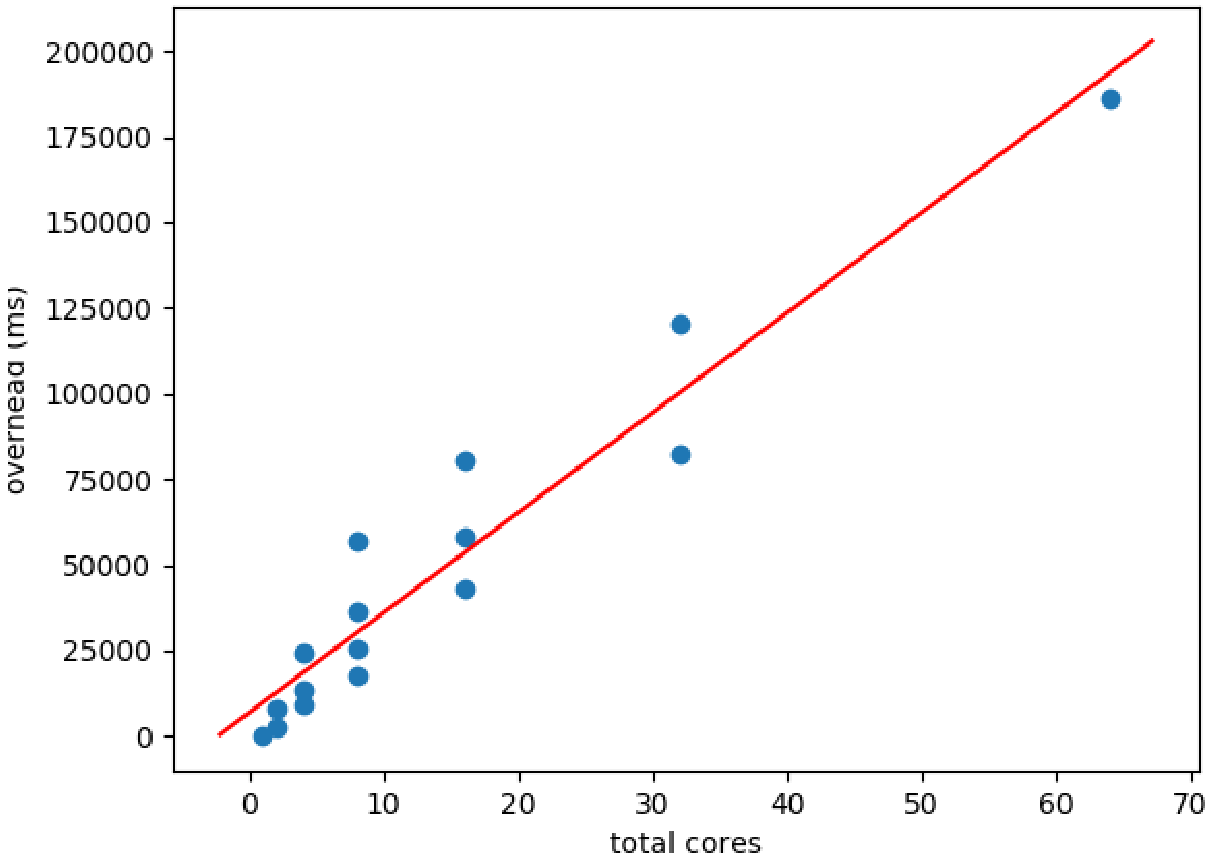

Figure 2.3: Overhead/Isoefficiency for task 1 on test3

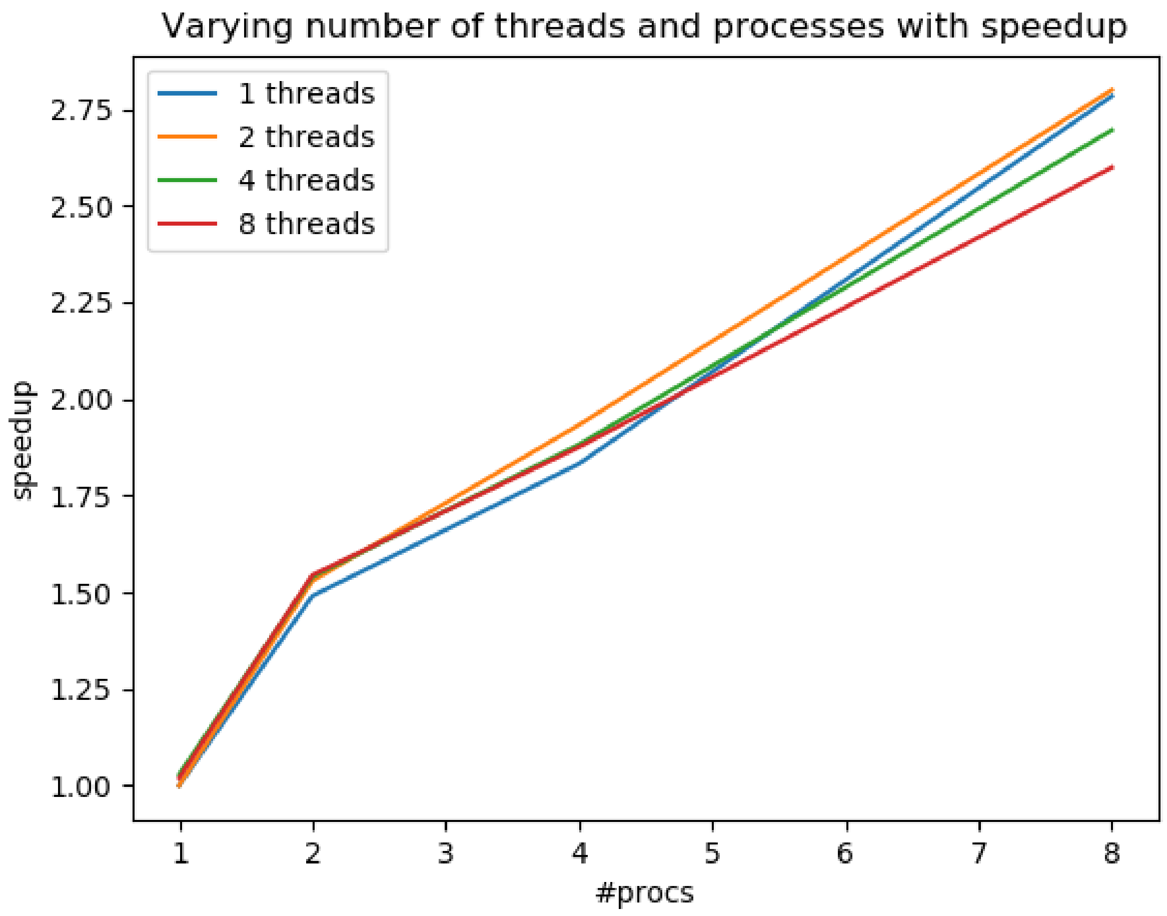

Figure 2.4: Speedup for task 2 on test3

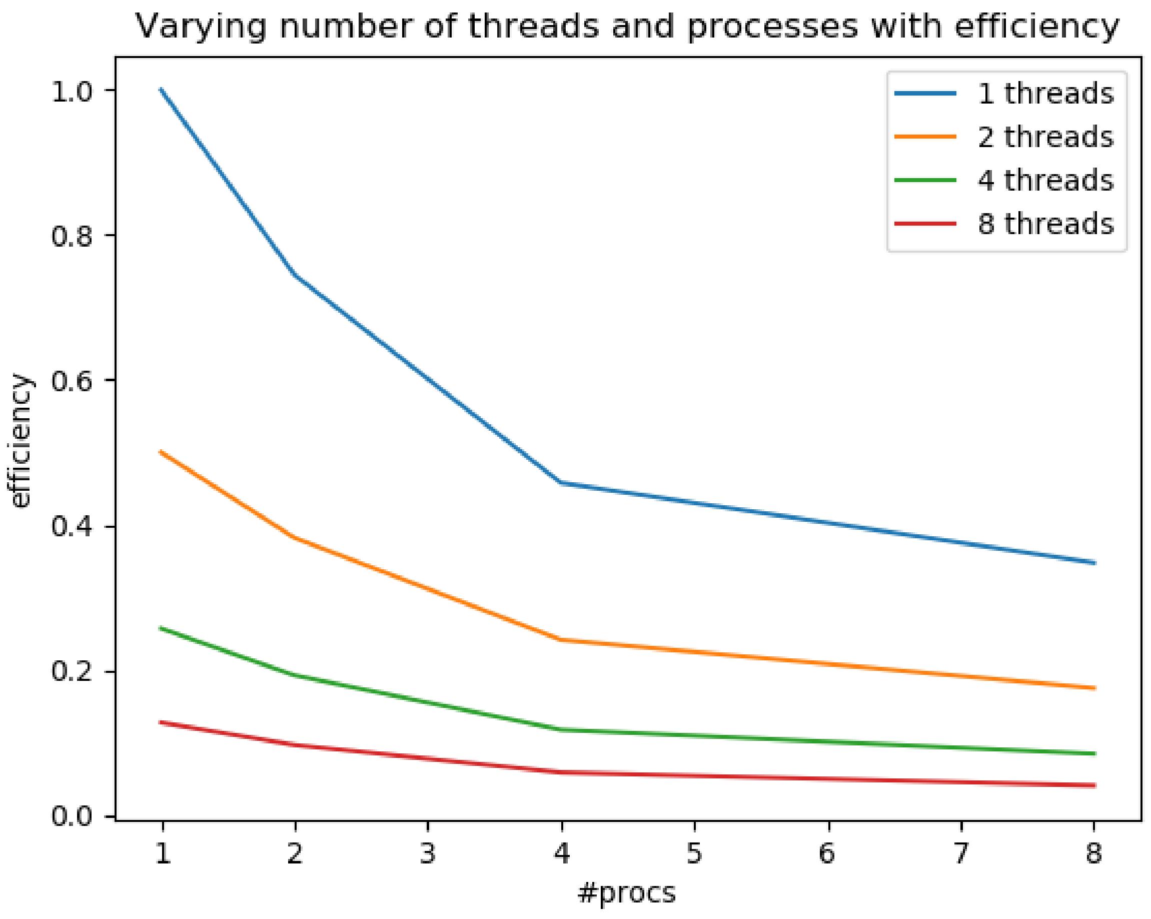

Figure 2.5: Efficiency for task 2 on test3

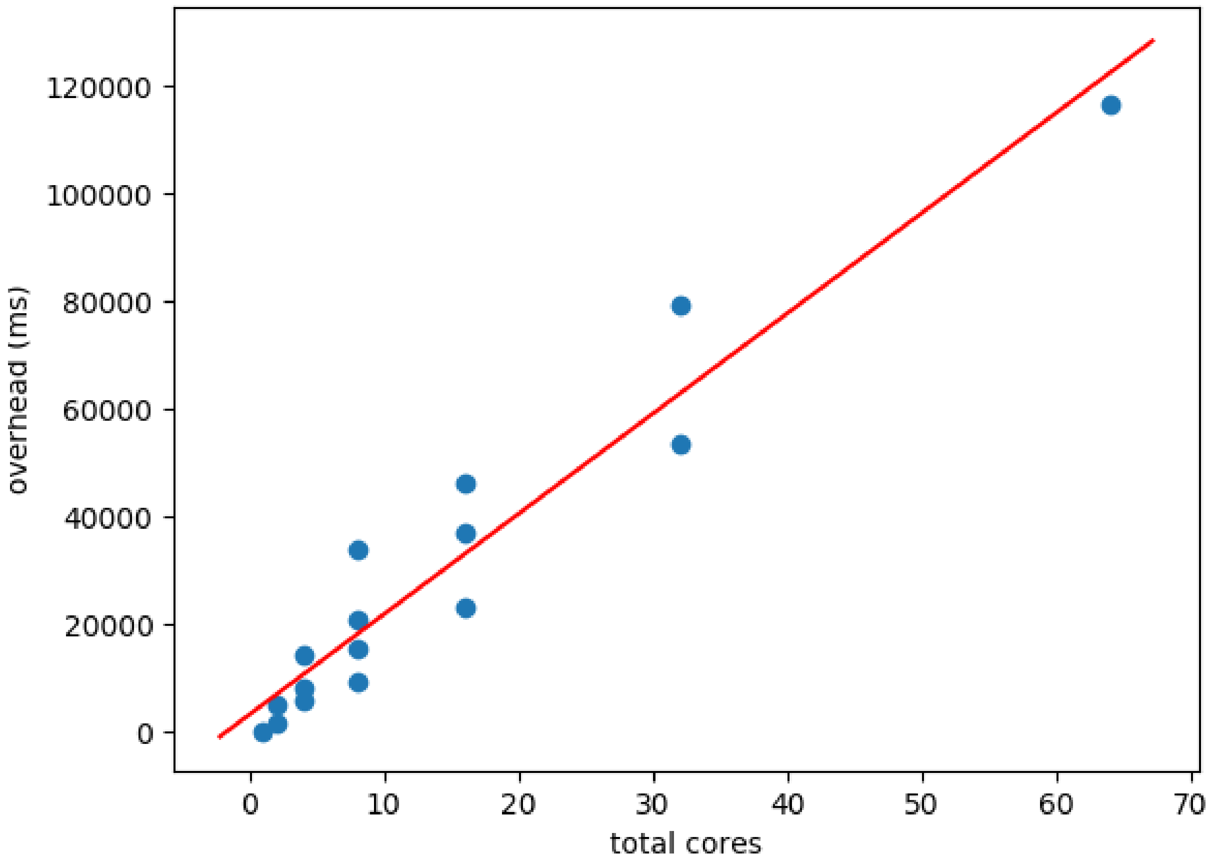

Figure 2.6: Overhead/Isoefficiency for task 2 on test3