Q1

(a) For the given problem we notice that the output (line numbers and positions of words) for a given portion/chunk of the file is independent of what is there in the other chunks of the file.

- This means we can identify the tasks beforehand (*static task generation)* and use some kind of data decomposition technique / recursive decomposition technique.
  - Recursive decomposition is of no use here since we are not performing any kind of divide-and-conquer strategy or solving sub-problems.
- We can use **input data partitioning** where each task handles a given chunk of the file.
  - In each task we compute the relative line numbers and positions of occurrences of the input words in the allotted chunk. All these tasks take the same amount of time since the input is distributed uniformly.
- In the end, all the results are combined at after which the relative line numbers are modified to the actual line numbers and outputted. This is a separate task on its own.

On testing for np=6, it was found that the average time taken for a process to compute the relative line numbers for its chunk was 0.52223sec, while the average time taken for the last task of combining the results was 0.00536sec.

Using the above result, the weights for the first level of nodes (representing the first set of tasks to compute the relative line numbers) can be taken as 97 while the weight for the last node (representing the last task) can be taken as 1.

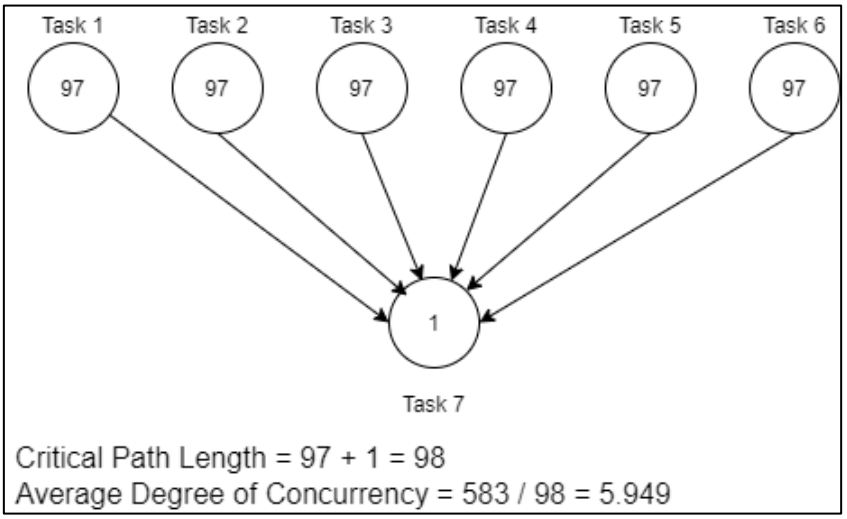

Figure 1: Task Dependency Graph

The only interactions that happen are between task 7 and each of tasks 1-6 (for passing over the relative line numbers so that they can be combined at task 7).

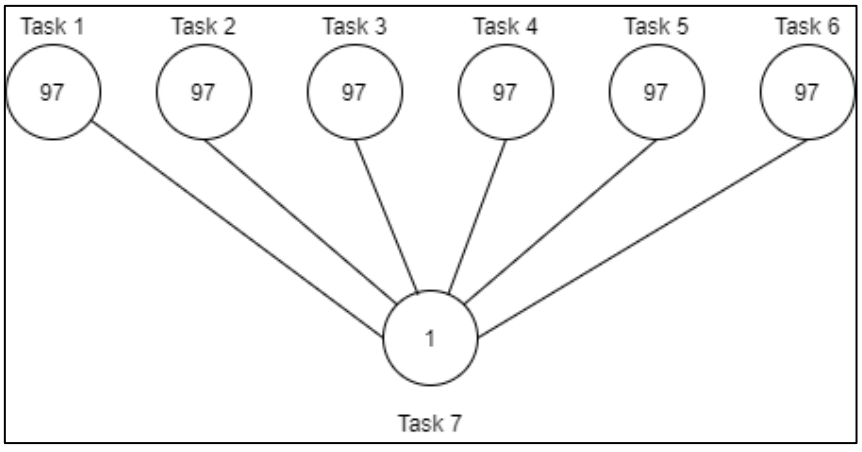

Figure 2: Task Interaction Graph

It may be noted here that the above interactions are optimized by using the MPI_Gather() and MPI_Gatherv() collective operations. The manner in which these interactions are optimized is abstracted from the user. An example of such optimized interaction would be :

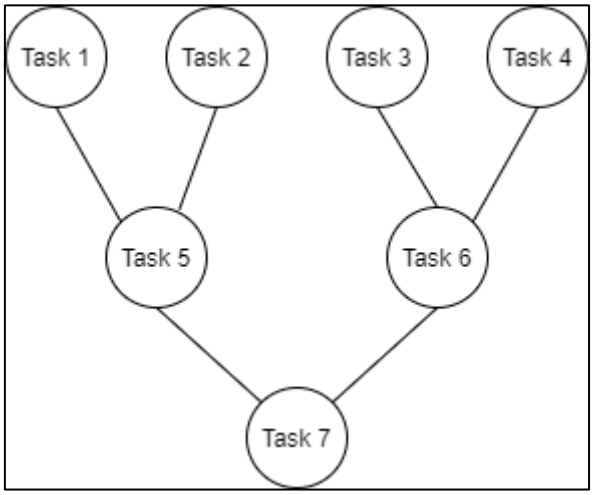

Figure 3: Optimized task-interaction graph

(b) Since the tasks are statically generated and their sizes are known beforehand, we can go for some kind of *static mapping.*

- The tasks are already partitioned based on input-data, so the mapping can also be based on this same data partitioning.

We observe that the first level of tasks is the bottleneck here, since they very heavy compared to the last task. Thus the issue of efficient load balancing is of much higher priority compared to minimized idling.

- This means task at the first level should be mapped to a single process so that the load is balanced evenly.
- The last task is dependent on the completion of above tasks, and so can be taken up by any process (the rank-0 process in this case).

Idling is present during the execution of the last task but is very *insignificant* compared to the total execution time since the last task's weight is very low.

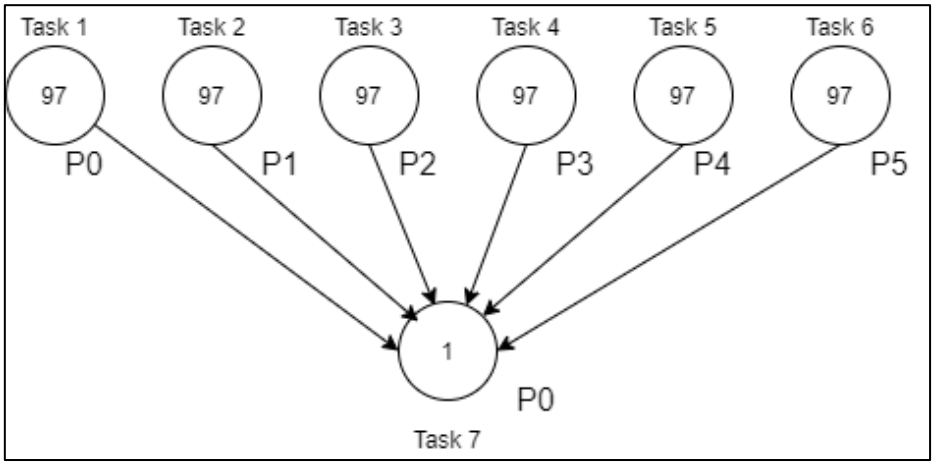

Figure 4: Task-Process mapping

(c) The most efficient sequential algorithm for the given problem makes use of a *trie* data structure where all the query words are inserted into a trie and the input file is scanned sequentially. This allows for linear time searching of all the query words.
The parallel algorithm is no different since we only partition the input data and run the same sequential algorithm parallelly on all the processes.

Every process in our algorithm does the following:

- Insert all the query words into a trie.
- Calculate the starting and ending position of its chunk based on its rank and filesize.
    - For a case where a line happens to get split across 2 processes, the process with lower rank takes the entire line while the other process drops that line.
- Traverse this chunk line by line and simultaneously update the trie state, and note down any matches along with their relative line numbers (with respect to current chunk) and positions.
- Finally, all results are gathered at rank-0 process (using MPI_Gather) and the absolute line numbers are outputted.

(d), (e) The efficiency, speedup and cost were plotted for a given input file and the no_of_processes were varied from n=1 to n=4. The graphs are shown below:

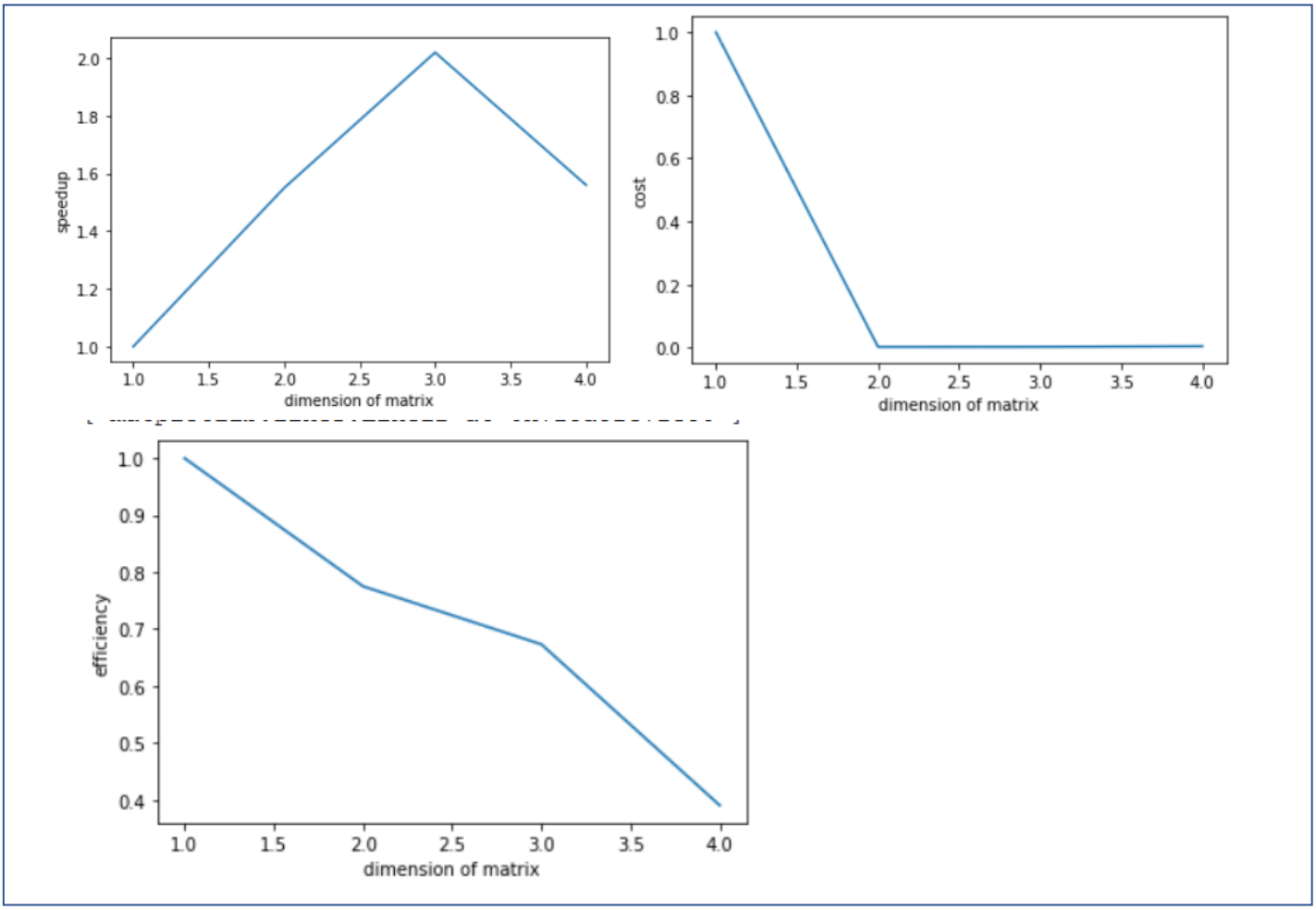

Figure 5: Speedup, Cost, Efficiency vs no of processes