Alper Şahıstan
21501207

CS426-Parallel Computing Project 2 Report

1. Implementation of kreduce

This implementation takes two integer arrays that contains ids and similarity values of some documents stored across processes and finds the k smallest similarity values across these using a reduction scheme discussed below. Then all of the results are accumulated in one process that records the ids of k smallest values. Reduction scheme uses **recursive doubling** with **producer/consumer** scheme. All of the send and recieves are MPI_Send and MPI_Recv's **no collective communications** are used whatsoever.

Reduction scheme is as follows; firstly all of the active processes are grouped into two halves where first half receives k values from their partners and compares all of those values with respect to their k values to find their k smallest values and second half sends their k values to first half. After this the second half becomes inactive since their data is reduced to their partners. This step is repeated until number of producers and consumers is smaller or equal to 4(since an average computer usually has minimum of 4 cores). Then one final reduction is applied from all remaining processes to master process. The reduction is similar to above; all of the remaining processes send their k values to master and master compares and reorders its least k values array with respect to its own k values and received.

There is one edge case that needs mentioning; if number of active processes are odd then one process must stay idle for that iteration since it does not have a partner to send/recieve data. This is illustrated in the 2nd iteration of Figure-1 along with an example execution.

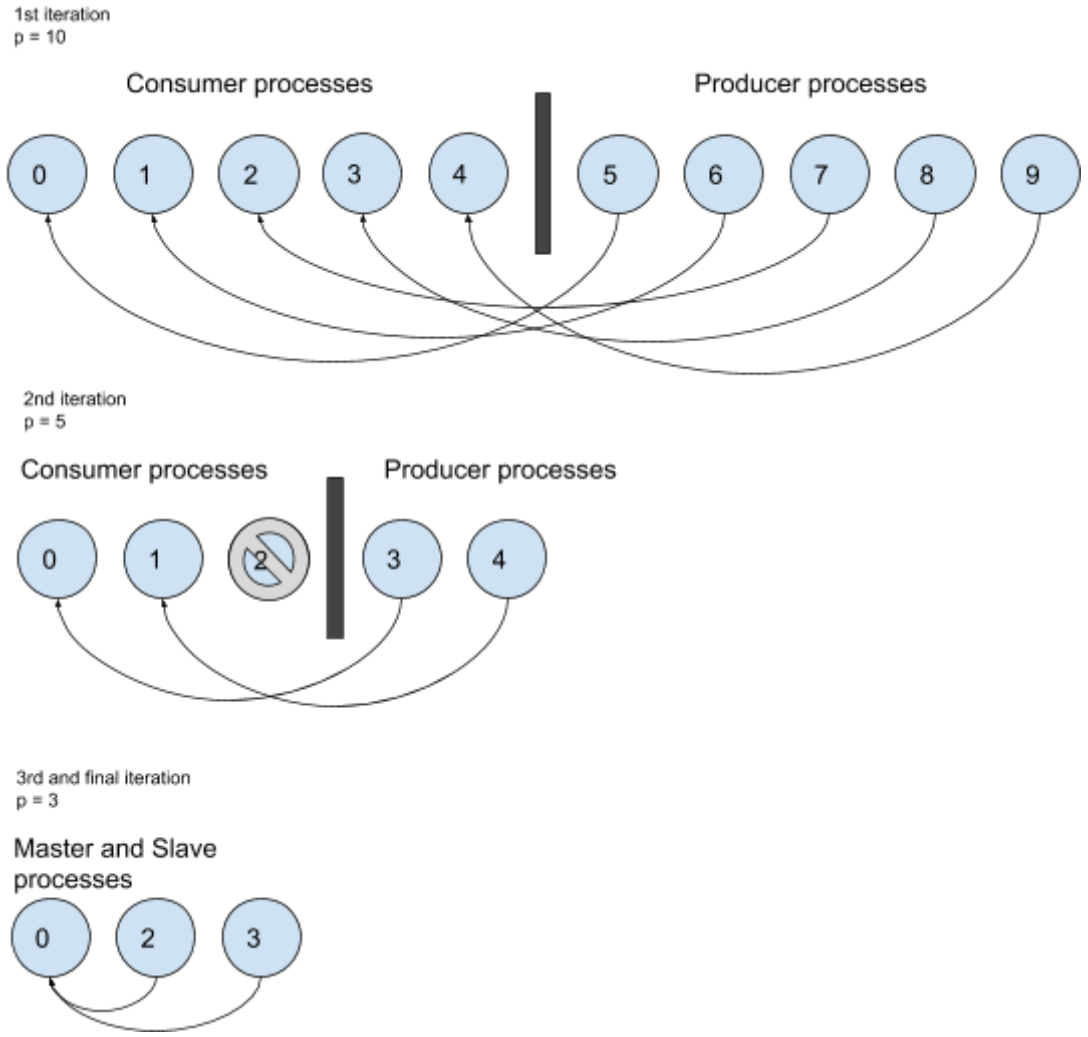


Figure-1: Example execution of kreduce

2. Implementation of Main Program

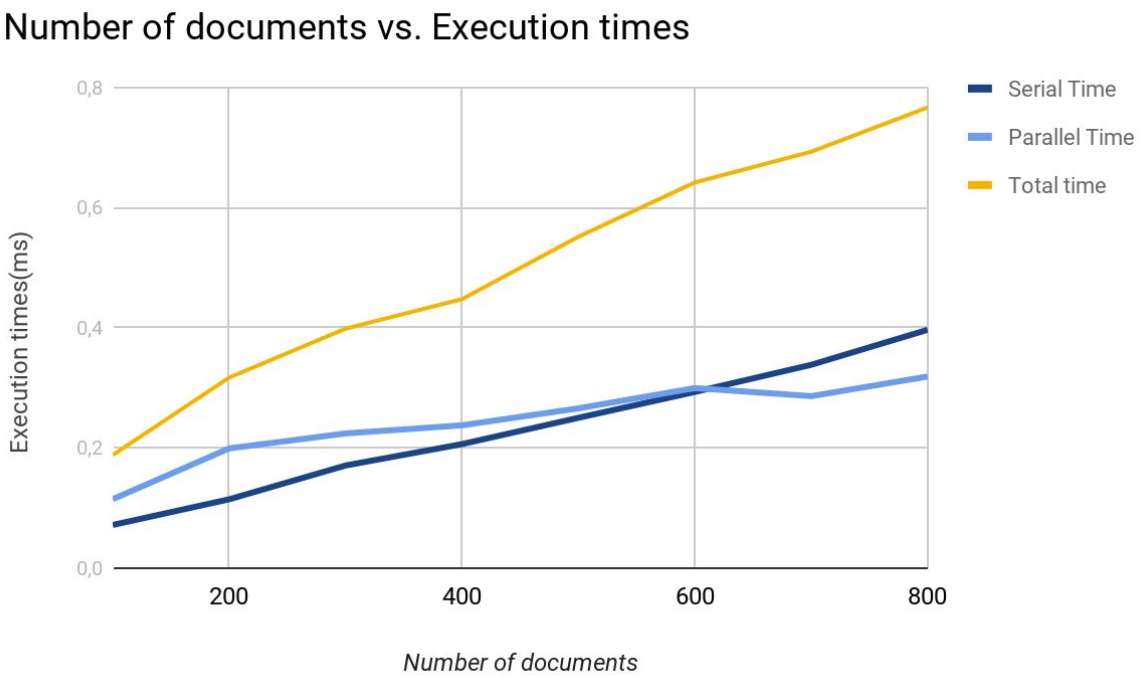
Main program takes paths to two text documents(one for query and one for document->weights) two integers k value and size of dictionaries. In the initial stage all processes read these parameters from arguments to main.c then Master process reads documents into a (document)struct array and reads query to an integer array. After that master process distributes documents to slave processes by evenly as possible(if number of documents is not divisible by the number of processes last slave process takes all the remainders).

After receiving other necessary data like query, number of documents per process, dictionary size and k slaves and master start calculating similarity values for their data. For each document similarity value is calculated as given in the assignment then compared with kth element in a values array(which has a size of k). If new calculated similarity is smaller than the kth value new similarity value and its id replaces kth value and it's id in the values and ids arrays respectfully. Next it runs one iteration of insertion sort over the values array for the kth value while preserving the correspondence relation between ids array. One iteration of insertion sort is to preserve the ascending order of the values array. This

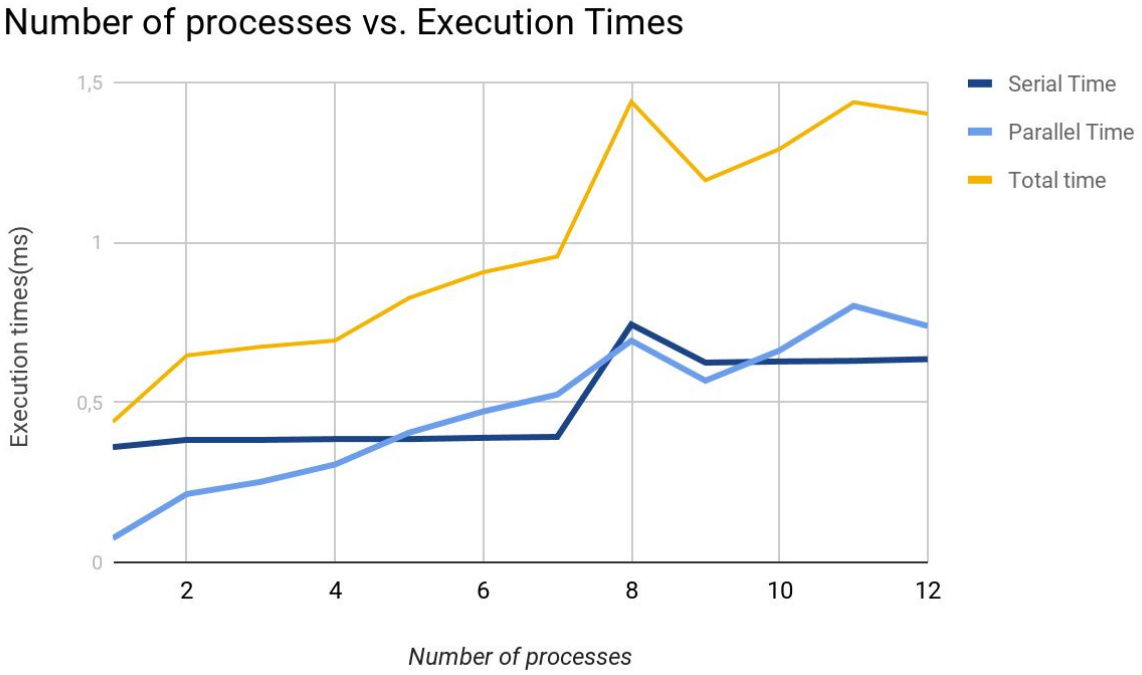
happens as follows; last inserted(kth) element is compared with the values in the array from end to start and swaps places with the larger values when encountered(ids are swapped as well). This preserves the ascending order since all the other elements in the array are already in the ascending order. After every process calculates their k smallest values they call kreduce.
Finally results of kreduce are printed along with the benchmark times.

3. Benchmark Results

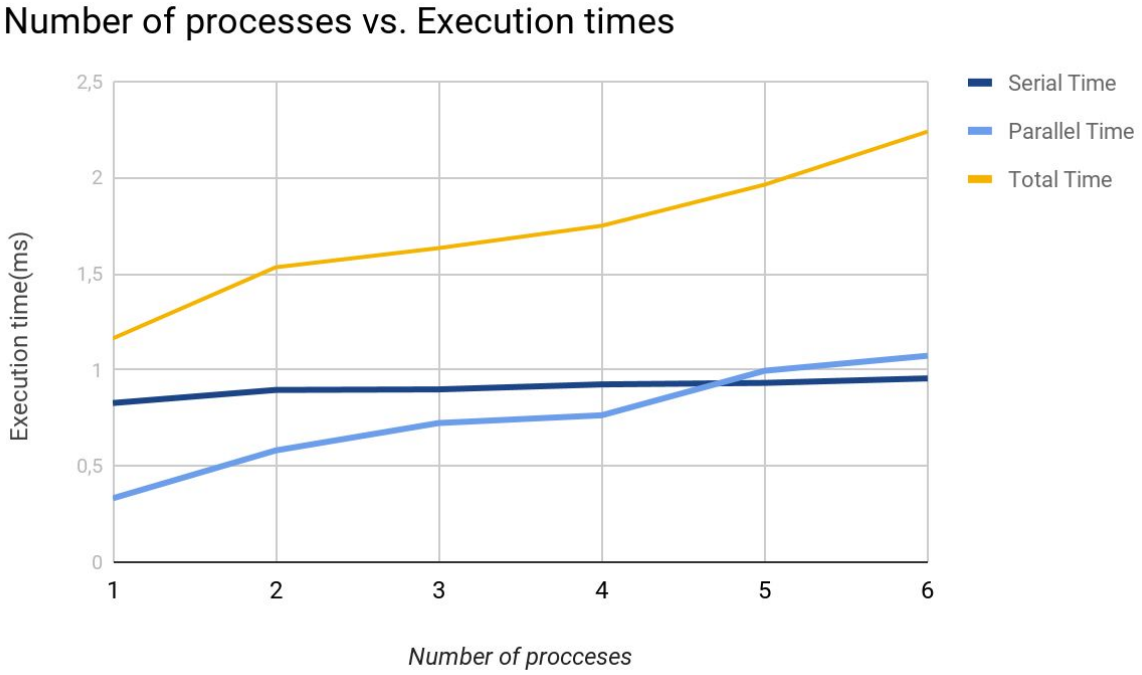
To eliminate noise all the of the data points are taken as the average of 50 executions.



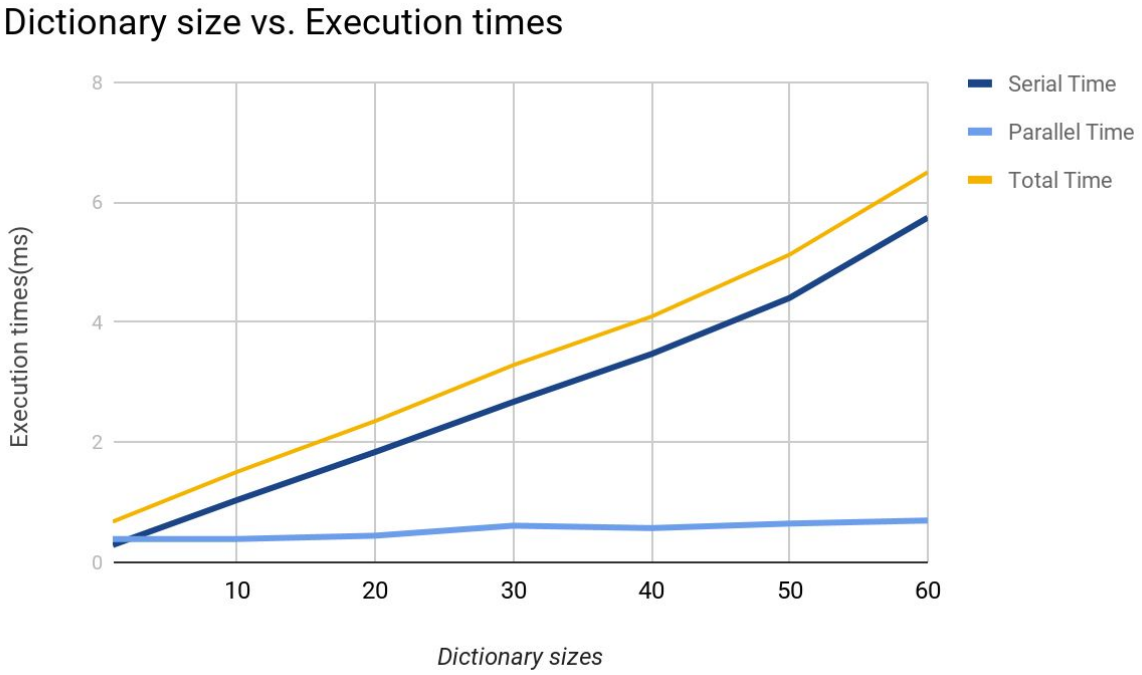
4 processes are used with dictionary size of 3 and k=3 for all of the data in this graph.



800 documents are used with dictionary size of 3 and **k=3** for all of the data in this graph.



2000 documents are used with dictionary size of 3 and **k=100** for all of the data in this graph.



1000 documents are used with **process count of 4** and **k=50** for all of the data in this graph.

4. Discussion

For this assignment it's clear that number of documents lengthen the serial time as well as the total time while parallel time gets affected on a minimal scale. However when we look t the process count vs. Execution time graphs it can be said that if the data is small and required operations are minimal parallel execution lengthens the total run time as well as the parallel execution time. The spikes in serial time of the first “Number of processes vs. Execution times” graph can be explained by increased context switches due to increased granularity. Although we read the files from one process MPI call initiates the code with n number of processes from the beginning causing context switches. However when one looks at the second “Number of processes vs. Execution times” graph it can be said that with the increased amount of data and tasks parallel execution time does not suffer as much as in the former graph. Therefore one can say that increasing the amount of computation and the parallelizing it is more cost effective. Yet despite all the increased computation size any increase in the process count increases the execution time especially the parallel execution time and by that total time. When we look at the last graph the increasing dictionary time does not affect the parallel execution time yet it greatly increases the total time as well as the serial time. This implies that expanding the dictionary size does not create any communication overhead. The increase in the serial time is due to master threads increased workload.
One can use Amdahl’s Law to calculate the speed up.

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

Where S is the speed up, P is the portion that is parallelizable and n is the number of processes.
Then an example execution average for **dictionary size of 50, k=3, # of processes = 4 and # of documents = 100000** yields the times:
Sequential Part: 465.232139 ms
Parallel Part: 47.572777 ms
Total Time: 512.804916 ms

Then,

S = 1 / (0.90723026337 + (0.09276973663 / 4))

S= 1.07478031507

From these one can conclude that MPI's communication cost is too high for given sizes of computations. A distributed scheme may be better suited for this API. Additionally one can add that main bottleneck of this application is the file reads. ~90% of this application time is actually serial execution therefore to achieve better parallel performance one can parallize the file reads.