CS426 - Parallel Computing
Project 02 - Report
Aldo Tali
ID: 21500097
Date: 07/04/2019

Part 1) Explanation of kreduce function

Below the full implementation of the kreduce function for this project is given. All processors call the same kreduce function after they have received their respective batches of documents that they will need to process (the distribution of the initial data). The overall idea is that each processor computes its own local k-length array of least k similar ids for the documents that are present in that processor. In order to proceed with the necessary computations for finding the local most k similar ids `computeSimilaritiesSorted` function is used. In the function for each computation, the processor takes the batch of documents and computes the similarity value for each of them with respects to the given query. As the computation is being done all the resulting values are arranged such that the ids and vals represent local min heaps. So each insertion takes log(k) time into the local heap. At the end all processors have their own heaps with least similar document on the top. At this point getting least k similar documents means that all we have to do is extract the top value of the heap and rearrange the heap structure k times (giving the local least k elements). This is done by the call to the `findleastkDocs` which returns the local least values array. Then the function goes into two simple cases. Case 1 is when the number of processors available in the system is only one (The program needs to emulate serial execution completely). It is important to distinguish this case because it may be the case that in specific arrangements, the send and receive calls (being null) could lead to deadlock, even though the chances for this to happen are low. The second case is when we have more than one processor available. In this case all processors other than master send both their ids array and their leastk values array to the master. The master on the other hand receives 2 arrays from each other processor and uses an algorithm similar to the merge of MergeSort algorithm to join the local least similar documents together. This is done in the call to `merge2KResults` After merging with the values of all processors, then the master simply outputs the time measurements and the final least k values of the least k similar documents.

```
//implements the kreduce
void kreduce(int * leastk, int * myids, int * myvals, int k, int
world_size, int my_rank){
   int  *myleast;
   int *ids;
   int *idReceiver = (int*) malloc(k * sizeof(int));
   double parall;
   leastk = (int*) malloc(k * sizeof(int));
   //compute partial result for this processor
   computeSimilaritiesSorted(&pv,1);       //use a min heap to compute the
similarities
   myleast = findleastkDocs(&pv,&ids,k);   //extract the min from heap k
times

   //1 proc means serialized version
   if (world_size == 1){
      /* end = MPI_Wtime();
       parall = end-start;
       printf("Sequential Part: %f ms\n",sequential*1000);
       printf("Parallel Part: %f ms\n",(parall - sequential)*1000);
       printf("Total Time: %f ms\n",parall*1000);
       if (pv.noDocs < k){
           resultPrint(pv.noDocs,ids);
       } else {
           resultPrint(k,ids);
       }*/

       pv.ids = ids;
   } else{
       if (my_rank == 0){
           int i;
           //accumulate partial results
         for (i = 1; i<world_size; i++){
          //get the least k from a proc
          MPI_Recv(leastk,k,MPI_INT,i,LEAST_TAG,MPI_COMM_WORLD,NULL);
          MPI_Recv(idReceiver,k,MPI_INT,i,IDS2_TAG,
                      MPI_COMM_WORLD,NULL);      //get ids ids
             //merge with your own info
             merge2KResults(&myleast, leastk,k, &ids,idReceiver);
          }
          /* end = MPI_Wtime();
           parall = end-start;
           printf("Sequential Part: %f ms\n",sequential*1000);
           printf("Parallel Part: %f ms\n",(parall -
           sequential)*1000);
           printf("Total Time: %f ms\n",parall*1000);
           resultPrint(k,ids);*/

          pv.ids = ids;
       }else {
           //send partial results
           MPI_Send(myleast,k,MPI_INT,0,LEAST_TAG,MPI_COMM_WORLD);
           MPI_Send(ids,k,MPI_INT,0,IDS2_TAG,MPI_COMM_WORLD);
       }
   }
}
```

Part 2) Explanation of your main program.

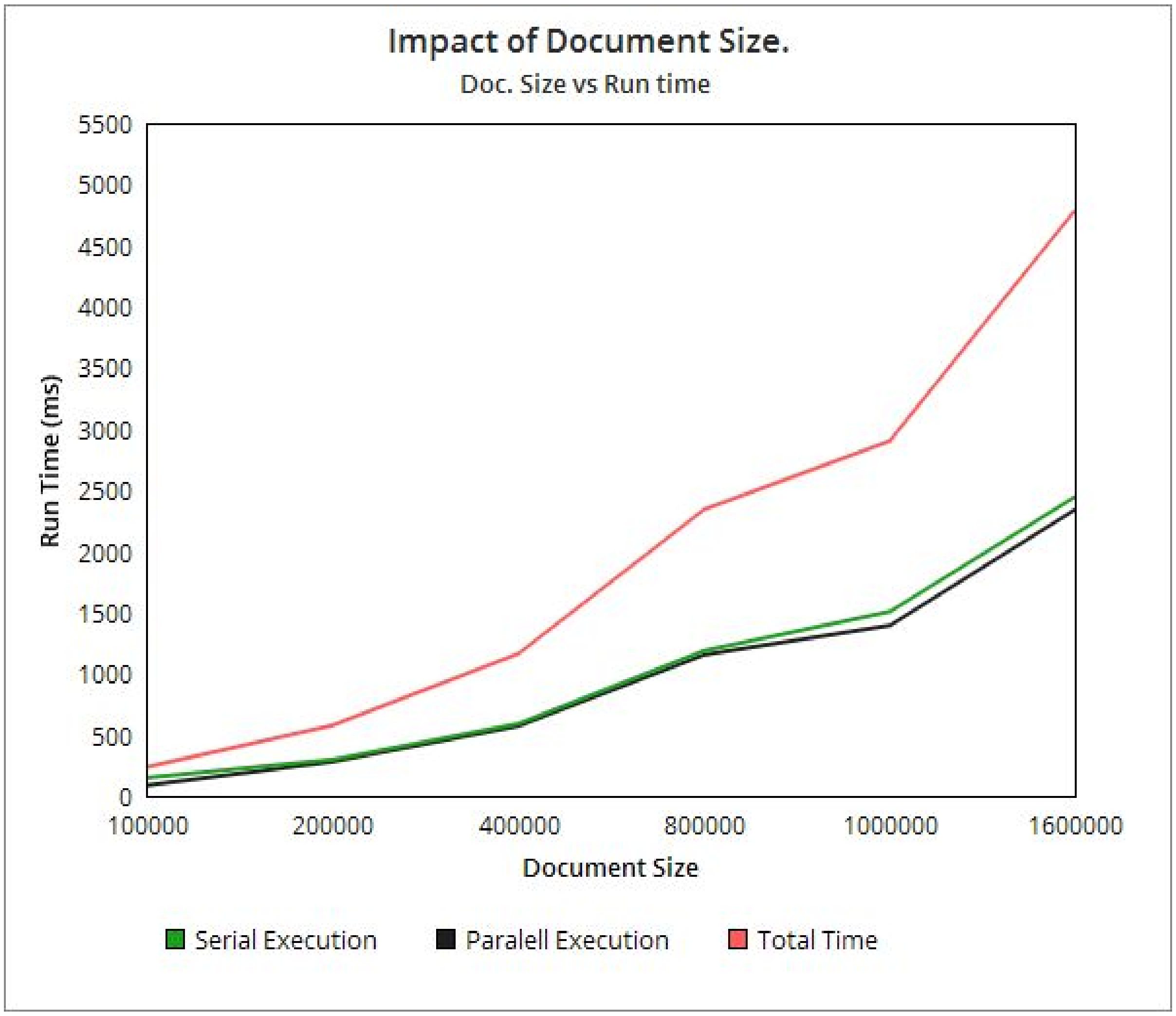
Given that for the explanation of the main a fully detailed explanation was not required the actual code piece was not included in this report, but it can nonetheless be accessed with the other submission files. The main program follows a simple logic. Initially the master node/process checks if all the command line input is done properly and if correct it will read all the documents and query in the given file. While reading it will automatically create documents array, which holds the documents that are needed by each processor. This is useful since distributing while reading the files enables us not to read the documents a second time. Plase note that it is assumed that up until this point is what is supposed to be called “Serial Execution” or serial time and the rest of what follows is given as parallel execution time. Then master will send the documents that are meant to be present at processor i to processor i. Processor i will block until it received the buffered data. Once the distribution of the initial documents is done and all the processors have their own local documents, all of them will call kreduce which will collectively be able to compute the global k least similar ids as explained in part 1.

Part 3) Plot several graphs for chosen parameters.

The graphs given below follow the datasets of the outputs that are given in the “outputs” folder of this submission. In order to make a meaningful analysis of the output results four parameters were chosen to test the performance of the kreduce implementation in this project. All the tests have been conducted on a Dual-Core AMD processor and each of the following graph keeps three values at constant in order to depict the fourth parameter. The graphs are represented here and the analysis follows in the subsequent task.

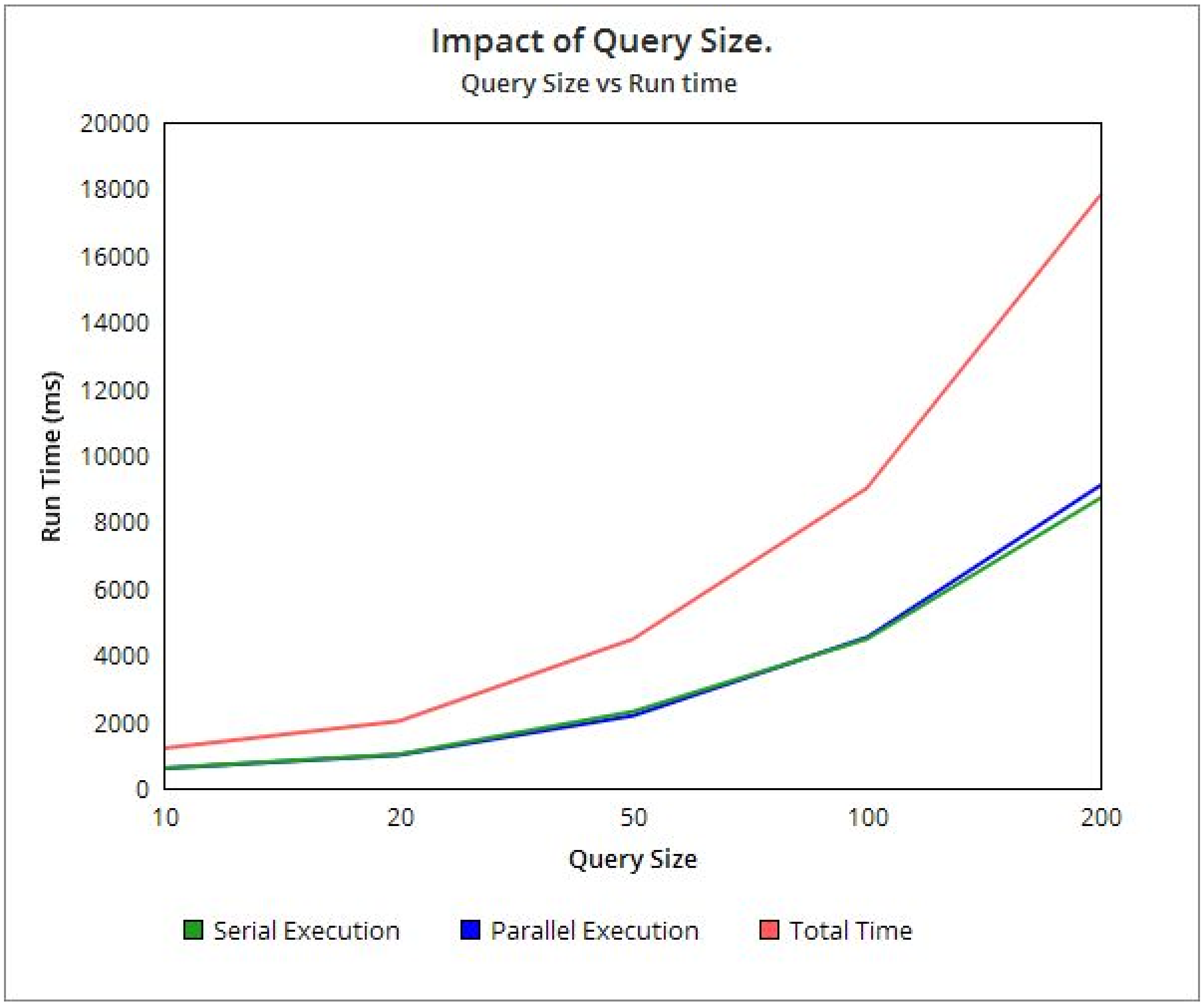
Graph 1 - Impact of Document Size in running time.

In testing this part several document sizes were given as an input to the running script. The number of processes was fixed to 2, the dictionary size was fixed to 10 and k value was fixed to 10.



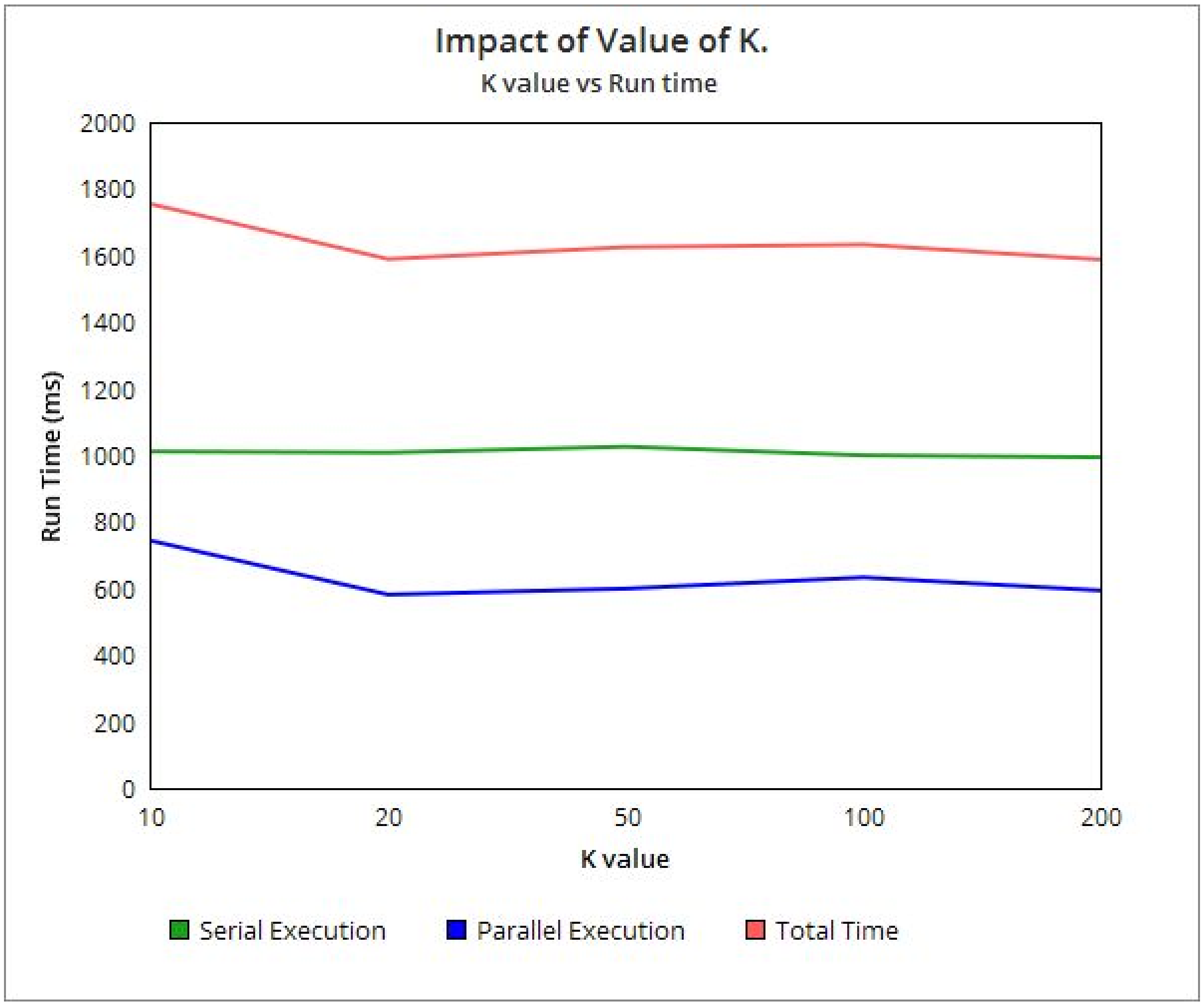
Graph 2 - Impact of Query Size in running time.

In testing this part several query sizes were given as an input to the running script. The number of processes was fixed to 2, the document size was fixed to 400000 and k value was fixed to 10.



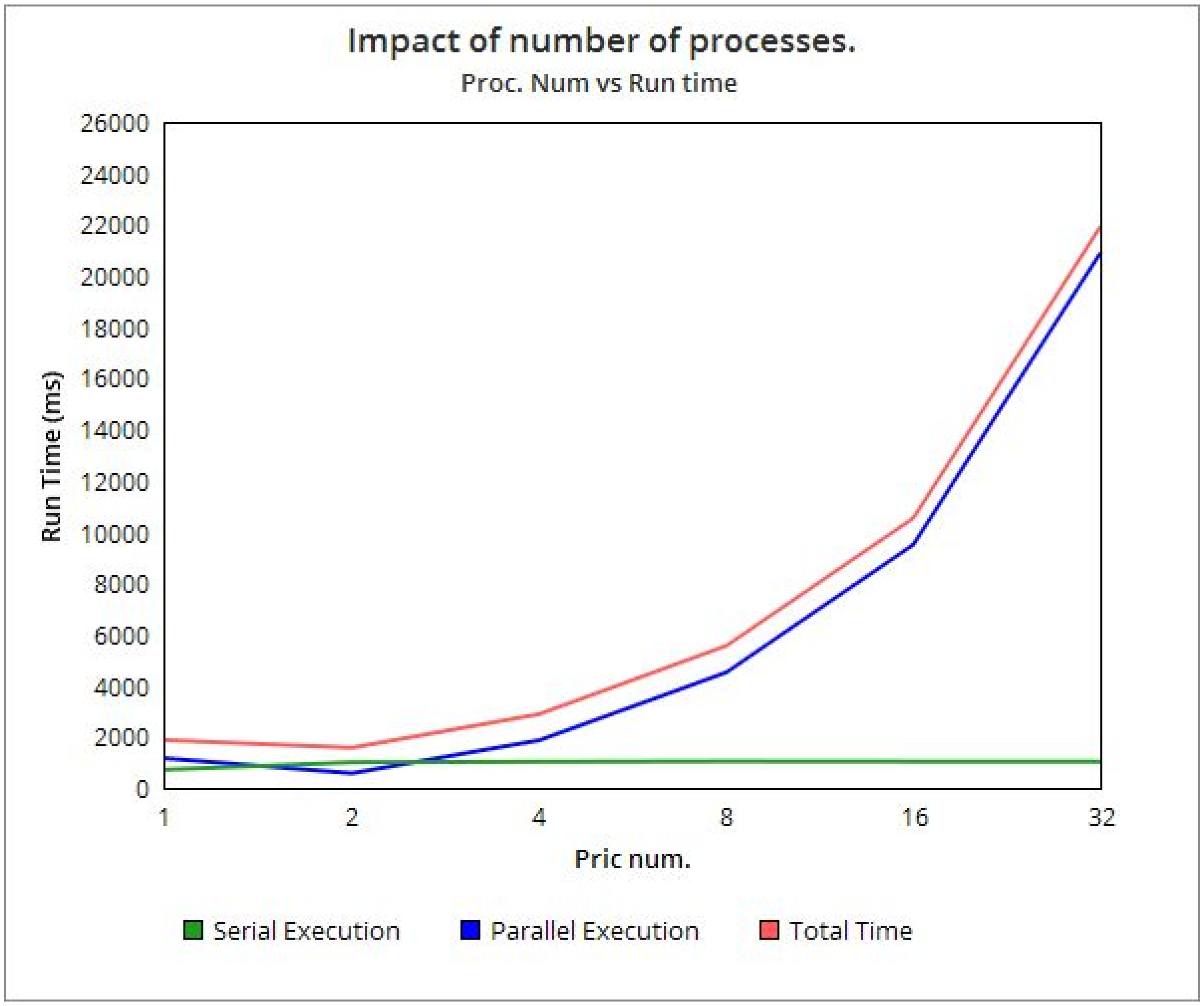
Graph 3 - Impact of Value of K in running time.

In testing this part several values of k were given as an input to the running script. The number of processes was fixed to 2, the document size was fixed to 400000 and dictionary size was fixed to 20.



Graph 4 - Impact of Value of Number of Processes in running time.

In testing this part several number of processes were given as an input to the running script. The document size was fixed to 400000 , dictionary size was fixed to 20 and k value was fixed to 10.



Part 4) Discussion of selected parameters and and performance results.

The parameters given above were considered worthwhile to consider since when analyzing them locally one might draw out certain benefits and drawbacks to the parallel implementation. For example in Graph 1 we can see that there is a monotonically increasing relation between the document size increase and the time taken by the serial and parallel part of the project. Of course this is to be expected since essentially in the serial part more documents means the array of documents has to be bigger and the time taken by the file to read and assign the documents to the respective array is larger. Same argument is valid to the parallel part of the execution since more documents will indeed require more computational power and time to compute the similarities.

Taking into consideration graph 2 it looks like the same argument is also valid for query size, however we need to be careful here since in the sequential part the query is read only once (there is only one line) and it should not overthrow a large overhead. Of course the values tested 10, 20, 50, 100, 200 are relatively small to be able clearly see that the overhead is not comparatively large to the rest of the execution.

In graph number 3 , one could make inferences on the efficiency of the algorithm itself. Given that all the other factors are kept constant, the serial time stays the same (expected since the same documents are read from run to run). Also the parallel time seem to not fluctuate, which is a good indication since even if the user may choose to get a large subset of least similar values, this will not affect the run time of the program. This comes mostly because of the heap implementation of the storage of these documents, values and ids.

In the last graph the impact of the number of processes is given. Since the device at which the script was run has only 2 cores, the point of diminishing returns comes relatively near, since after running with two processors the time needed to pass the documents and ids to the ith processor for all i’s present outweighs the benefit of the overall execution. However these results would of course change if the k-value lets say would also be increased. Nevertheless the shape of the graph is bound to stay the same. We can however see the improvement of the parallel execution if we compare the execution with 1 process and the one with 2.