CS 525 Project 1 Report

Name: Ali Necat Karakuloğlu
ID: 22101543

Part A)

Implementation:
In serial implementation, the code reads the input file (given as an argument) then finds the maximum and minimum elements in the input array. After a pass, code prints the range (i.e. max-min). Parallel code uses the same algorithm. In parallel case however, master node partitions the work among other processes and sends a partition of workload to other nodes (w.r.t. their ranks). They execute the same range determination algorithm and send the max and min values they found. Master process compares all max and min values. Finally, master node finds global max and min values and prints the overall range. Difference between parallel v1 and v2 is parallel v1 use MPI_Recv and MPI_Send methods to deliver local minimum and maximum values. In parallel version, MPI_Allreduce method is used. Note that, with the allreduce method, requirement for delivering a result to a node is eliminated. The method combines the minimum and maximum values into a global variable.

Performance:
The results for small sized, medium sized and large sized data are provided in Figures 1-3, respectively. Small data contains 10K, middle data contains 1M and large data contains 100M integers. In all cases, serial has the best performance. I think in parallel versions, the time is mosly spent on communication of nodes. As can be seen from the code, algorithm doesn't have a large complexity (O(n)). Thus, the execution time of the algorithm is small compared to the communication of nodes (especially, sending array partition).

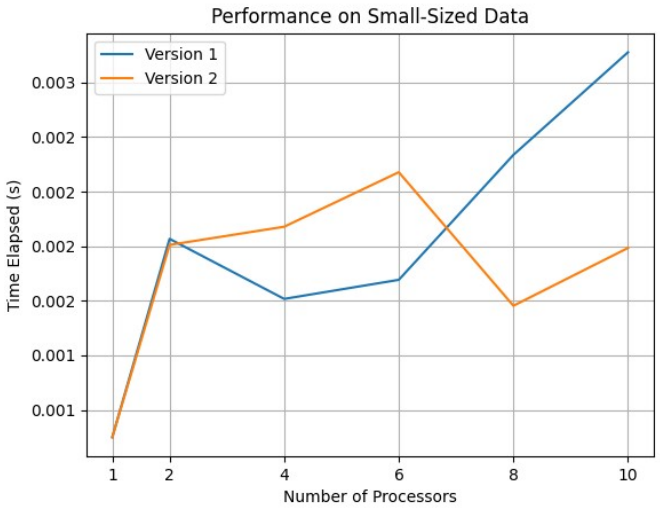

Figure 1: Performance on Small Data (1 Processor is the Serial Code)

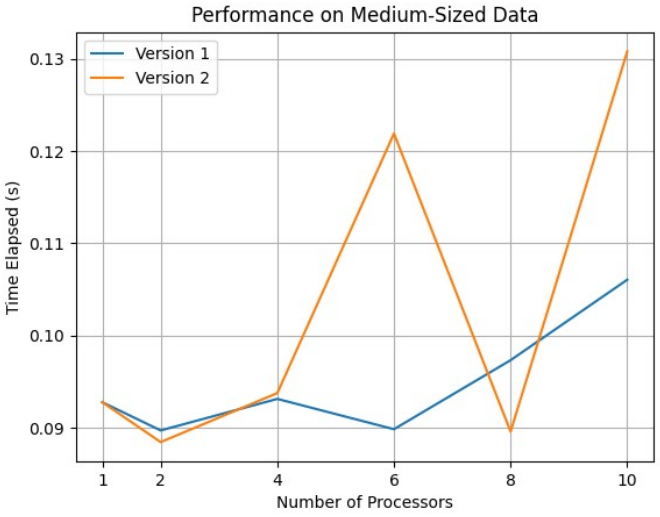

Figure 2: Performance on Medium Data (1 Processor is the Serial Code)

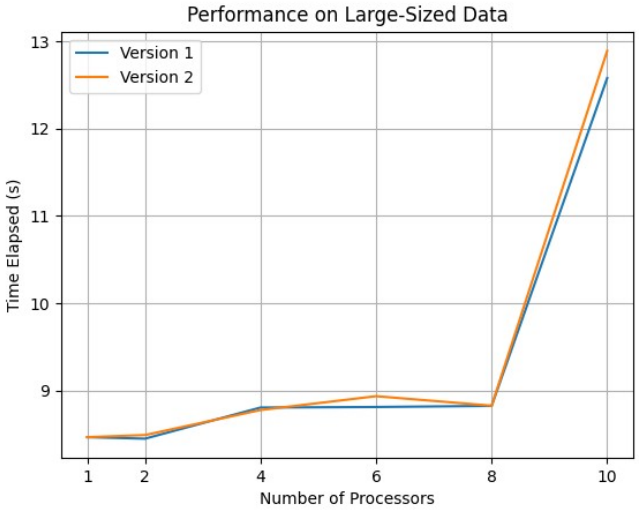

Figure 3: Performance on Large Data (1 Processor is the Serial Code)

Part B)

Implementation:

In serial implementation, the code reads input file. Then, sorts the kpArray by using bucket sort and another sorting algorithm (sort() from algorithm library). Then, it finds the maximum consecutive difference in kpArray. The algorithm to find the difference is as follows. First, check whether consecutive two elements are the same. If so, find the difference between the second keys. If the difference is greater than the maximum difference found, assign the new maximum difference. If the two first keys don't match, print the maximum consecutive difference of that bucket. Parallel algorithm works the same way. However, in parallel code, the sorting algorithm is modified to write the result to another key pair array. The sorted key pairs are copied into a fraction of original key pair array. Note that, the master node reads input and sends it to the other nodes as a whole. The original key pair array is read by other processors and corresponding buckets are copied into partial key pair arrays. In other words, original one is not altered but, each process makes a copy for itself. The division of workload is determined by the master node. Master node sends two information to the other ones. First one is start bucket id and the other one is end bucket id. A process is responsible from the buckets in this range. After processes find consecutive differences, they send it back to master node as an array. Master node prints the results in an output file whose name is obtained as an argument from the command line.

Performance:

The results for small sized, medium sized and large sized data are provided in Figures 4-6, respectively. Each data contains 360 buckets. In small sized data, each bucket contains 10 integers. In middle sized data, buckets consists of 1K integers and in large sized data, there are 100K integer in each bucket. Similar to part A, increasing parallelism increases the execution time of the program. Since the data is small, communication overhead may be increasing the time when number of processors are increased. In middle size data, making the program parallel increases the performance. There are certain points (number of processors) that decreases the compilation time. Before and after those parts, the time increases. I think such behaviour is related to the architecture of CPU and the distribution of the workload by compiler. If the number of processors are increased, execution time increases in large sized and middle sized data. This result can be explained by the work distribution on physical cores. Physical cores are limited. If we go beyond this point, execution time may increase. However, in large and middle sized data, there is a significant benefit coming from parallel execution.

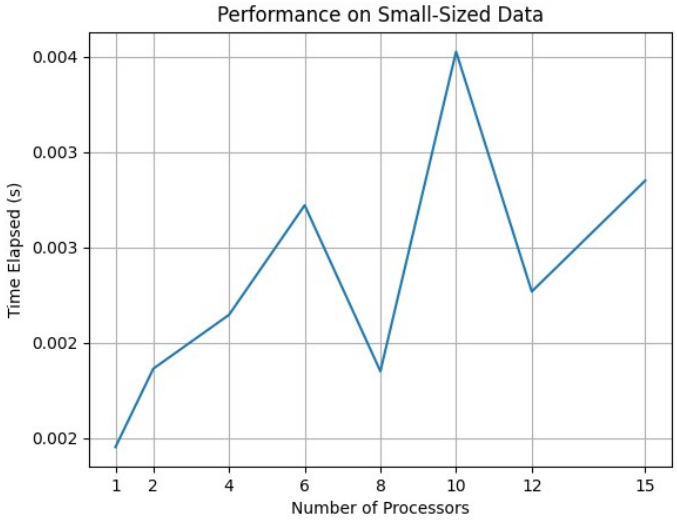

Figure 4: Performance on Small Data, Part B (1 Processor is the Serial Code)

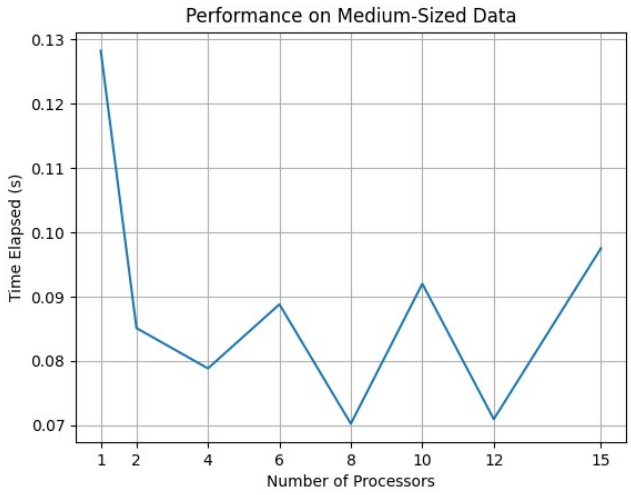

Figure 5: Performance on Medium Sized Data, Part B (1 Processor is the Serial Code

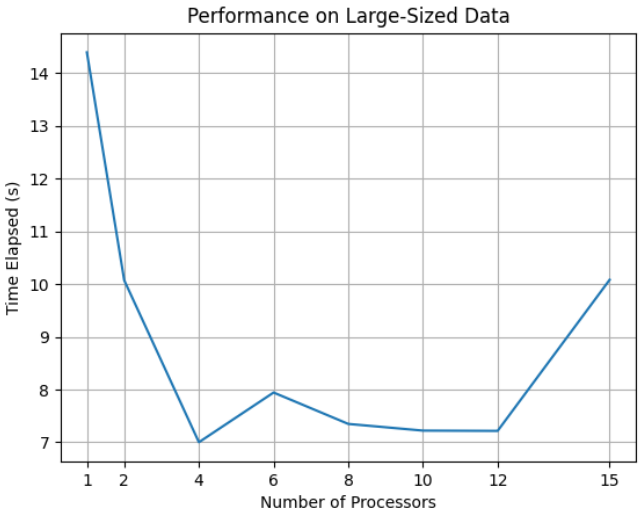

Figure 6: Performance on Large Data, Part B (1 Processor is the Serial Code

Platform Specifications:

◦ Operating System: Ubuntu 20.04.3 LTS
◦ CPU: AMD Ryzen 7 3700X 8-Core Processor
◦ Number of Logical Cores: 16
◦ RAM: Corsair Vengeance 8 GB 3200 MHz DDR4, x2
◦ HDD/SSD: SanDisk SSD PLUS 120GB
◦ Compiler: g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
-I couldn’t compile with gcc or mpicc, I used g++ and mpiCC