CS426 - Parallel Computing
Project 01 - Report
Aldo Tali
ID: 2150097
Date: 10/03/2019

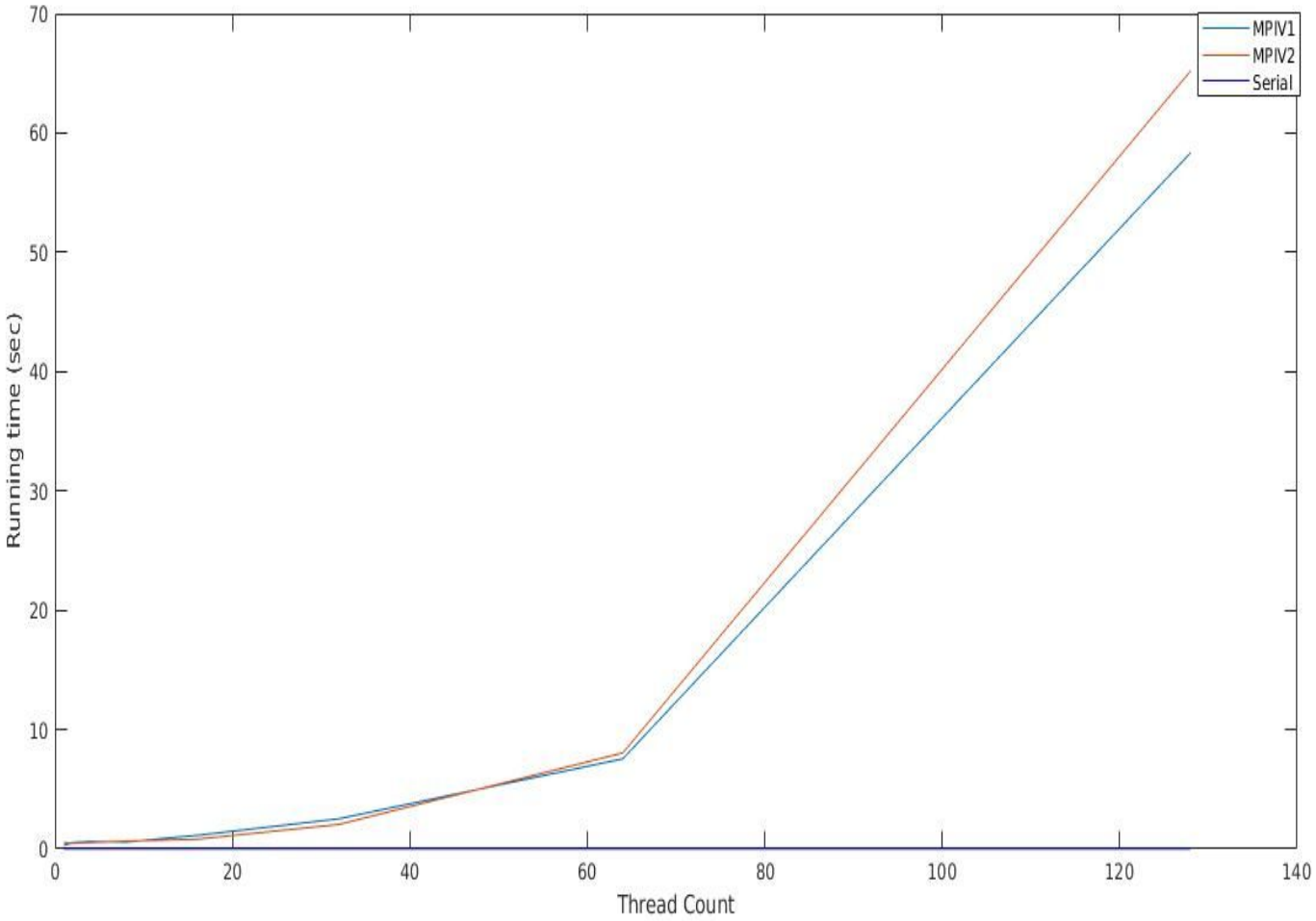
Q1) Explain the implementation and design choices

In part A the serial implementation constitutes of two steps, first being the reading of the numbers into an array and then summing up the array. The other implementation design would have been to simply sum up the values as the script was reading through the file. In here the choice is not space efficient as in the latter case we would only need to store one integer variable not the whole list of numbers in memory. However this type of implementation was chosen in order to make the code more reusable and granular for the other parts. In the second section of Part A,MPI_SEND and MPI_RECV were used for point-to-point communication. Using these instead of MPI_Isend for example makes the code simpler to manage as MPI_SEND and MPI_RECV are by themselves blocking calls, meaning that there is lowered risk of accessing wrong memory addresses. In MPIv1 the master process is sending 2 types of information to each processor: the size of the chunk and the buffered chunk. Each slave receives them,sums up the elements in the array and sends the local result back. In MPIv2 the exact implementation to MPIv1 is used with the exception that at the end, the master process uses MPI_Bcast() to send the sum to all processes. The reason for this choice is that at the time the homework was announced I was not fully confident with MPI_Allreduce() as it came later in the lectures and also due to the fact that the first 2 parts were already written in certain granularity that it made it easy to implement MPI_Bcast() in almost no time.

In Part B the serial implementation simply reads the two files and allocates two 2D arrays for the matrices. As in part A, I chose to keep a resulting matrix in memory instead of writing it to a file as I progressed computing the entries for the same reasons mentioned above. Please note that this implementation reads the second matrix in a column major fashion for easiness in the entry multiplications. Then in MPI version the processes are thought of as a grid with a grid dimension X. Assuming the dimension of the matrices is Y then each of them would contain Y/X such grids. So for each processor i in the grid only Y/X rows and Y/X columns are sent as data buffers, then each processor permutes the vector multiplication between the given rows and columns and sends the indices and results back to master. Please not that the only reason that we are certain for this to work is that the message passing is blocking by nature and if it wasn't then it would be possible to have misleading indices in the overall computation. In here the only problem I faced was the fact that I could not used different buffers in sending multiple buffered data from master to slaves (the columns and rows of process i) so I had to perform deep copy of the row data so as not to lose them. In the case when the input is 1 processor then the solution diverges to the serial case as it is an exceptional case.

Q2) Plot a graph with various thread numbers indicating the performance of your implementation. Use sequential implementation as a baseline.

Below the report presents the run time for threads in multiples of 2's starting from 1 until 128 threads.



The lower baseline deptics the serial execution whereas the upper two the MPIV1 and MPI V2 codes. Please note that this graph does **not** include running times for part B, as in my implementation I was able to get correct results for the example matrix 3x3 and 6x6, but when scaling up the messages sent also scale and memory addresses get confused. As such in higher order matrices currently the implementation reaches segmentation faults due to accessing the wrong addresses which I was not capable to fully debug.

Q3) Your observations about the performance of your implementation.

When talking about the performance of these implementations, I must note down that the computer at which the submitted implementation was tested is a Dual Core, 3.3GB Ram Ubuntu device. As a result, as seen from the graph, for part A of the assignment the serial implementation and the parallel ones come close in execution times for small number of threads but they quickly diverge for large number of processes. In theory this shouldn't be the case and there should be at least some improvement in the parallel implementation, however since there were only 2 processors (hardware) in the testing period, that essentially means that no matter how many threads we create, at most my machine can execute only 2 at the same time. Given that the system does not have only the threads created by MPI but also its own previously running ones the improvement chances decrease as well. Another thing to consider are the overheads, for example running the serial program proves to be 1/10th of a second faster than running the parallel process with a specified number of processes of 1. Although in here the same number of threads runs, the computation is slower as the system also has to account for the overheads of dealing with MPI initialization. The improvement however can be seen in the first few threads of execution where the running time, has small downhills, showing that increasing the concurrency can lead to some improvement if the concurrency number is chosen reasonably and carefully.