Alper Şahıstan
21501207

CS426-Parallel Computing Project 1 Report

1. Implementation and Design Choices

1.1. Sum Programs

Serial implementation of the summation program was very fundamental and straigth-forward. Simply, program read the given file into a self-implemented vector struct and summed the read numbers into a variable for printing.
MPI-ppv1 implementation utilized Send and Receive functions of the Open MPI. Program uses process with ID 0 as master process while threatening others as workers. Master reads the values from the specified file and divides the work among it-self and other worker processes(last worker process may take +1 number if the given amount of numbers are not perfectly divisible by number of processes).Again aforementioned vector struct and it's util functions are used to store the read values. Master first sends the number of integers that every worker will sum as the first message so that workers can allocate an appropriate sized array. Then master sends the "sub-arrays" to each worker as the second message. Then each process computes their sum (including master). Then worker processes send their partial sums to master which sums those up to print the final result. The program is implemented to reduce the communication cost as much as possible.
MPI-ppv2 implementation was very similar to ppv1 implementation however instead of workers sending their part to master via Send/Receive calls MPI_Allreduce with MPI_SUM parameter is utilized allowing every process to get the final result. Although assignment mentions Broadcast calls for this program they were not made use of since broadcasting whole array to all processes was deemed inefficient.

1.2. Matrix Multiplication Programs

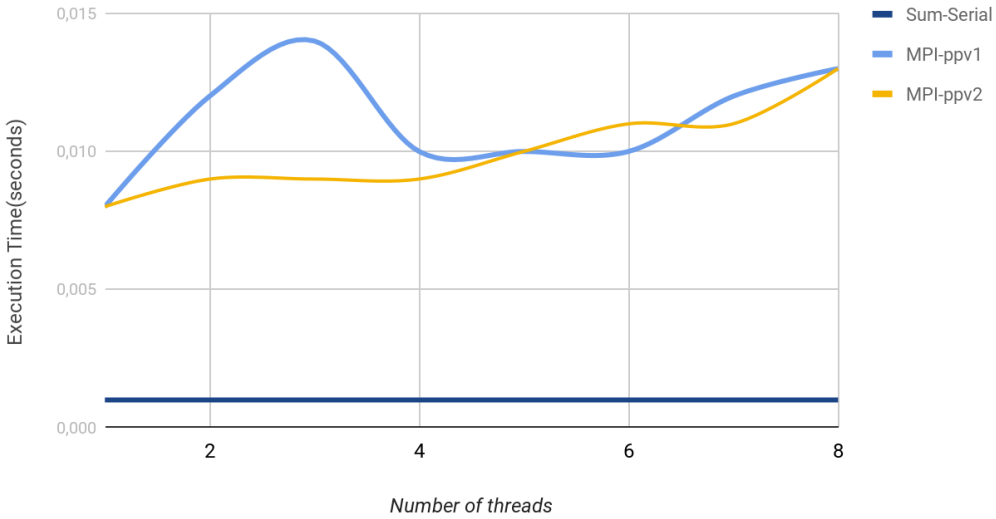
Serial implementation of the matrix multiplication code is again very straight forward. Matrices are read into 2-D arrays and applied multiplication. At every step of multiplication result is written into a file. This implementation can be improved performance-wise however it will be used as a base-line so it is left as naive as it is.
MPI implementation of matrix multiplication again was made as efficient as possible where the heavy work is being done. Firstly, 2 matrices are read into 2-D arrays and 2nd matrix is transposed. Then both of the matrices are "flattened" into 1-D arrays. These step are done to exploit the caching in-terms of locality by storing sequentially accessed items physically close and possibly in the same blocks. After preprocessing the matrices, the work load is distributed evenly among processes by the master via Send/Receive calls of MPI. Every process calculates a "tile" of a result matrix. After master and the workers are done workers send their parts to master via Send/Receive calls of MPI. Then master combines them into an 2-D array. Finally, that array is written into a file.

2. Benchmark Results

To eliminate noise all the of the data points are taken as the best(min-time) of 50 executions. Serial executions are used as a base-line, they were not executed with multiple threads however they are placed in the same graph with the mpi executions to illustrate the performance comparisons. Sum programs are tested with 1000 number ASCII file and matrix programs are tested with 30x30 matrices.

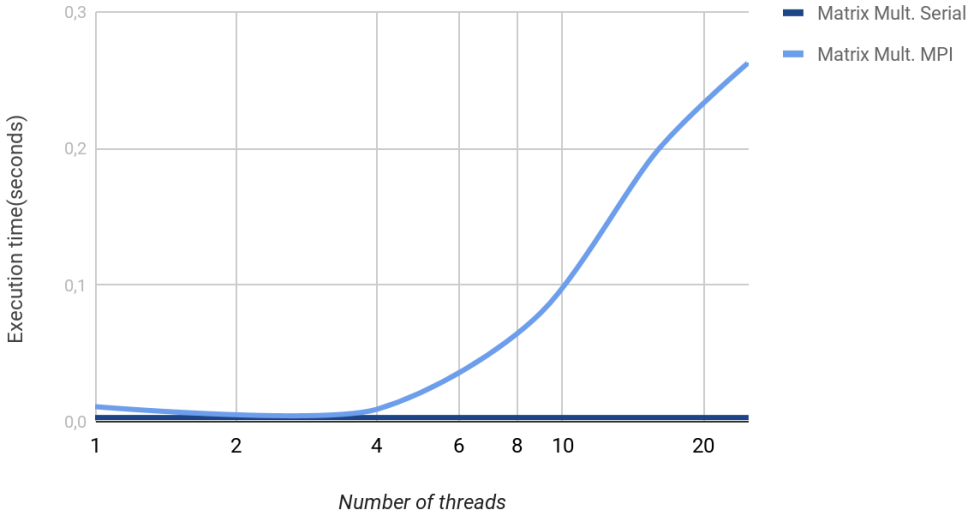
2.1. Sum Programs

Sum Programs Execution time vs. Number of threads



2.2. Matrix Multiplication Programs

Matrix Multiplication Programs Execution time vs Number of threads



Due to nature of the program linearly increasing thread numbers cannot be applied. Thus to illustrate the data clearly data obtained via an exponentially increasing number of threads(1,4,9,25...) is plotted over an x-axis logarithmic scale graph.

3. Comments and Observations

For the first part of the assignment it's clear that serial execution out-performs the mpi implementations by far. However if one compares the mpi implementations among each other one can observe that MPI ppv1 reaches optimal points where thread numbers are 1, 4 and 5 where as MPI ppv2 execution time keeps growing as the thread number increases. The explanation for ppv1 implementation results can be that the machine that ran the tests has 4 physical CPU cores which implies 4 threads can be ran truly parallel. So every thread more than 4 will be scheduled by the OS resulting in context switches. These context switches are costly comparatively to truly uninterrupted runs. However, in the case of ppv2 the optimal point is between 1-4 which implies similar results however we do not see the peak at 3 which we see in ppv1's case. The reason for this can be that the extra communication cost of copying the results to master process and master doing the aggregation of partial sums is reduced due to collective communication.

For the second part of the assignment one can observe that serial execution again out performs the mpi implementation. And peak performance of the mpi implementation is at 4 threads. The reason for this result is the same as before, the number of CPU cores in the machine.

To sum up, MPI may not be the ideal API to execute parallel programs on one machine. That note aside, when compared among each other the MPI programs results indicate that 1-to-1 correspondence between number of CPU cores and threads usually brings the optimal performance. However, some applications are better left serial, since the overhead of dividing the labor may not be amortized by the power of parallel execution.