CS 426 Parallel Computing

Project 1 Report

Alp Ege Baştürk

21501267

10.03.2019

Design Choices

All implementations were based on 1D arrays to prevent flattening matrices later.

- **sum-serial.c & matmult-serial.c**

  Implemented with basic for loops.

- **sum-mpi-ppv1.c**

  Implemented using MPI_Send to distribute parts from master to clients. Every process calculates its partial sum and sends to the master using MPI_Send. Master receives using MPI_receive. Partitions are calculated by dividing to equal numbers and adding one for remaining part if division has remainder.

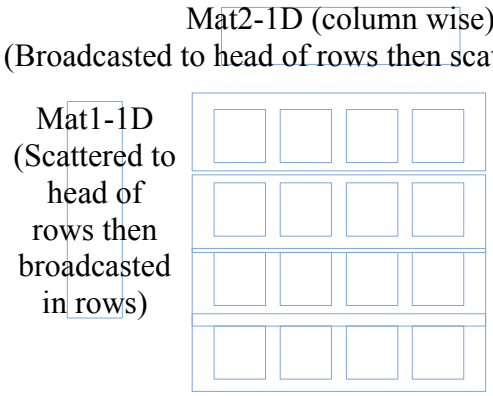
- **sum-mpi-ppv2.c**

  Implemented using MPI_Scatterv. Initially master broadcasts the array length, so that clients can also make simple calculations themselves. Then it creates a new MPI_Comm. This is required for an edge case when number of inputs are less than the number of inputs. In this case input number of processes will get pieces of length 1 and remaining won't get anything and block the communicator. So these idle processes are calculated at the start and removed in the new communicator. Later, a remainder calculation similar to the one in first part is used, but this time part lengths are saved in arrays. These arrays are passed to MPI_Scatterv, which can distribute non-uniform length parts using offsets. After partition, all processes calculate their partial sum and call MPI_Allreduce to get the whole sum from others. Use of scatter makes the implementation more efficient than using broadcast. Also unnecessary processes are removed at the start.

- **matmult-mpi-1d.c**

  Used groups and communicators to form grid like structure. Then the process calculating the grid called the serial matrix matrix multiplication to calculate the grid. MPI_Gather was used to collect partial grids at the master. MPI_COMM_WORLD was splitted using MPI_Comm_split according to colors which were calculated based on grid size. This provided rows. Also first processes of each row is grouped under master_world_group. Master reads both matrices, then scatters first matrix and broadcast second matrix to the master world. Each master broadcasts the part of the first matrix they received to the all others in their row communicator. Also they scatter the second matrix they received by the

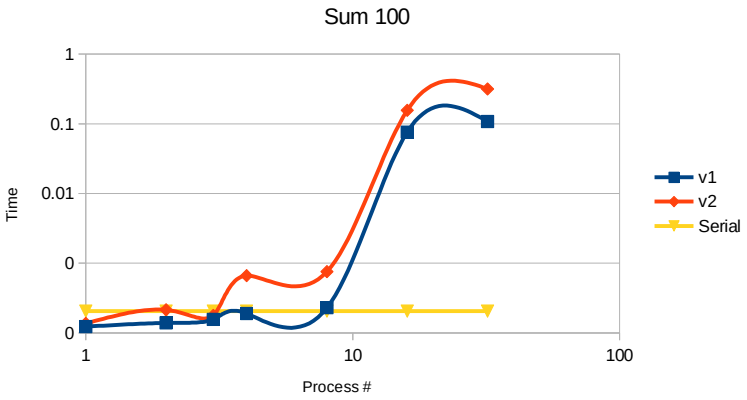
Broadcast to the ones in their row communicator. It was designed made the implementation more robust and efficient than using sends/receives and/or broadcasting the arrays to all processes. Furthermore, if each row group can work in parallel, then broadcast and scatters in different row communicators won't affect each other.



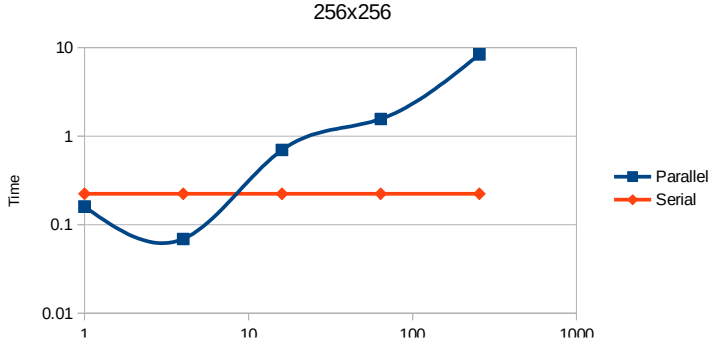
Data

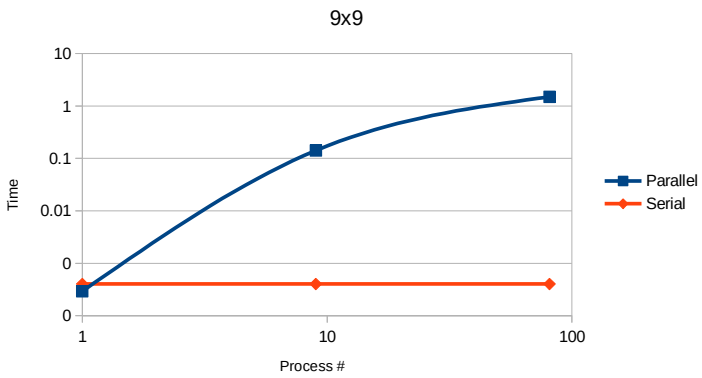
Following graphs are drawn in log scale.

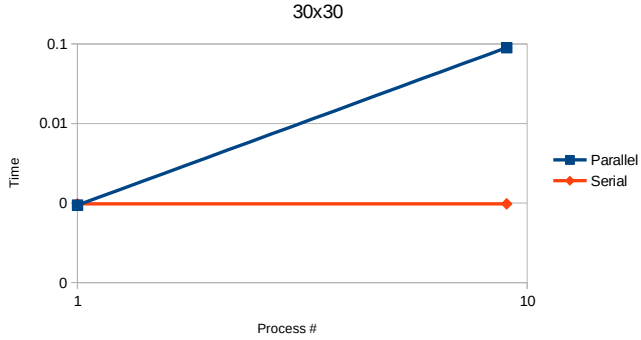
Process #	v1	v2	Serial
1	0.000123	0.000137	0.000206
2	0.000139	0.000214	0.000206
3	0.000156	0.000178	0.000206
4	0.00019	0.000661	0.000206
8	0.000229	0.000754	0.000206
16	0.075992	0.156059	0.000206
32	0.107986	0.315982	0.000206



Mat Mult	256x256		30x30			9x9		
Process #	Parallel	Serial	Process #	Parallel	Serial	Process #	Parallel	Serial
1	0.160202	0.223456	1	0.00094	0.000977	1	0.000292	0.000402
4	0.069045	0.223456	9	0.089825	0.000977	9	0.141852	0.000402
16	0.697896	0.223456				81	1.491988	0.000402
64	1.55982	0.223456						
256	8.36802	0.223456						







Observations

- **Sum**

As it can be seen from the graph, the serial implementation gives the same result as the parallel one for single thread. This trend doesn't change up to 4 threads. Also sometimes parallel implementations are better than the serial one such as the 3-thread run of the both mpi sum implementations. However parallel implementations perform poorly after 4 threads. This might be caused by the test machine, which is a Intel™ i7-4720HQ with 4 cores at 2.6 GHz. HT was disabled during the tests. To conclude, the parallel implementations perform better for large input sizes if they can run in parallel. However they perform poorly if they cannot run in parallel because they have more overhead and significant communication costs.

- **Matrix Multiplication**

Observations about the sum part can be said for this part too. Furthermore graph of 256x256 matrix multiplication supports that the parallel code performs better if most of the processes manage to run in parallel. It looks like all 4 available cores on the test machine were able to run in this implementation.