Parallel Programming HW3

Aashray Arora : SBU ID : 109940382, NETID : aaarora

Ajay Lakshminaryanrao SBU ID: 109898256, NETID: alakshminara

1. [120 Points] Distributed-Memory Matrix Multiplication.

    a. We have implemented cannon's algorithm. The code can be found in the file *mpi_matrix_replace.cpp.* We use the sendrecv_replace MPI call. We also tried the send and recv MPI calls only which can be found in *mpi_matrix_blocking.cpp,* but these take too long and hence we did not use them for reporting out performance.

    b. 1b) one process per node, s = seconds.

L/K	**10**	**11**	**12**	**13**	**14**
0	0.4s	3.37s	34.53s	354.07s	2643.64s
1	0.1s	0.81s	6.79s	75.76s	601.37s
2	0.04s	0.22s	1.61s	13.89s	150.43s

    c. 16 processes per node for stampede, s= seconds

L/K	**10**	**11**	**12**	**13**	**14**
0	0.03s	0.22s	3.01s	25.20s	200.24s
1	0.01s	0.07s	0.43s	6.03s	50.41s
2	0.02s	0.04s	0.14s	0.88s	12.11s

    d. We implemented this using the gather and scatter mpi calls. processor 0 scatters the submatrices to each processor. In the end processors return individual results to processor 0 using gather. The code can be found in *mpi_matrix_master_slave.cpp*

e. one process per node

L/K	10	11	12	13	14
0	0.43s	3.47s	35.08s	288.97	2298.30s
1	0.15s	1s	7.61s	76.73s	586.23s
2	0.08s	0.41s	2.38s	16.99s	154.65s

f. 16 processes per node

L/K	10	11	12	13	14
0	0.06s	0.33s	3.45s	25.99s	NA
1	0.07s	0.26s	1.19s	9.06s	NA
2	0.15s	0.35s	1.02s	3.97s	NA

g. Parts 1-b,c and 1-e,f clearly show that the performance is better since we are utilizing all the cores of each node by running t=16 processes per node. When comparing these two implementations of cannon's algorithm we observe that 1-e,f takes longer than its corresponding 1-b,c (no master processor). This is due to the added communication delay. In theory, having a master processor increases the computation complexity by a factor of root(p). This is clearly observed. Note: For k=14, we got errors from stampede and could not debug them further since stampede is not running jobs from our queues again.

2. [80 Points] Distributed-Shared-Memory Matrix Multiplication.

a. We used the ITER_MM_IKJ with the two outer loops (i and k) in parallel, which was the fastest in homework 1. The codes can be found in *mpi_matrix_replace_cilk.cpp* and *mpi_matrix_master_slave_cilk.cpp.*

b. Repeat part 1(b) with the two algorithms from part 2(a).

mpi_matrix_replace_cilk.cpp

L/K	**10**	**11**	**12**	**13**	**14**
0	0.64s	4.56s	49.78	470.95	3994.11
1	0.23s	1.21s	9.58s	106.78s	980.7s
2	0.3s	0.46s	2.71s	19.19s	204.7s

mpi_matrix_master_slave_cilk.cpp

L/K	**10**	**11**	**12**	**13**	**14**
0	0.65s	4.58s	49.37	469.93	3683.42
1	0.32s	1.60s	10.15s	105.50s	920.78s
2	0.41s	0.61s	3.37s	22.38s	222.56

c. We observe that using Distributed-Shared-Memory Matrix Multiplication, we did not observe much better performances if it all any (very minute differences), as we can see from the tables above. This may be due to the added communication delay. The running time would be dominated by the MPI communication that parallelizing within each node does not make a great difference. Since we are using blocking calls. Although, we could not re confirm our results as stampede is blocked are our jobs remain in the queue without running.