Josh Wretlind

CSCI/MATH 440

Parallel Scientific Computing

**Assignment 4**

How I handled the assignment if as follows.

1. Initialize wMatrix(as a PxJ matrix), as well as p and j and a few other constants
2. Generate r(as a 1xj matrix). The seed used here is currentTime + 100\*myRank. The 100\*myRank part comes so that no host uses the same seed as another one. I used the uniform\_real\_distribution<double> type contained within <random> paired with just the default random number generator to generate all of the random numbers needed for the assignment.
3. All of the hosts other than the master send their r vector to master, and master collects these and inserts them into the wMatrix.
4. Initialize a 1xL vector, to store all of the normalized weights in(**w** in the assignment)
5. Loop through wMatrix, adding up all of the elements, and use those for S.
6. Loop through wMatrix, divide entries by the sum found in 5) and store them into the normalized vector
7. Add all of entries in the normalized vector, and check to see if it’s within .1% of 1(what we’d expect), and if it isn’t, abort.
8. Send the normalized vector back to all hosts.
9. Allocate X(as a JxL matrix), and fill the values in, using the same formula for the seed as in 2.
10. Allocate C(as a JxL for all cases except for J=2, P=4, otherwise we allocate C as a LxL matrix, but only in the master. All other hosts for J=2,P=4 allocate as a JxL)
11. Allocate a 1xJ vector to contain sampleMeans for the J rows of X that each host contains
12. In each host, calculate what the sum of the squares for the normalized vector is, store it in weightSum;
13. Allocate Y as a LxL matrix.
14. For the first J rows of Y, calculate the values of Y as follows:

Yij =

This is to make sure that C is simply YYT and can be calculated as such

1. Each host sends its own J rows of Y back to Master
2. Master collects all of the rows of Y, and then sends them back out to the hosts.
3. Each host calculates YT
4. Each host calculates the J rows of C that they need to, calculating each as YYT
5. Each host loops through their own JxL C matrix and finds Cmax ,imax,jmsx , as well as Cmin ,imin,jmin for their chunk of C
6. If we are running for J=2,P=4, have master collect the rows of C back from the hosts, and output them.
7. Have all of the hosts send back their min/max arrays back to master
8. Master collects all of the pairs back in a Px4 matrix, and then looks at the first entry of each record in that matrix. It finds the max of all of those, and prints out the C value,(i,j) pair, and the core number that found it.
9. Each host outputs how long it took between MPI::Init and MPI::Finallize
10. Clean up all allocated memory.

For J=2,P=4, this is the resulting C I got, with the minimum and maximum bolded

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 9.07263 | 5.36182 | 1.39671 | 1.41325 | 4.36812 | -3.0984 | 1.59212 | **-5.21176** |
| 5.36182 | 11.2321 | 1.44929 | 3.5555 | 6.13477 | 9.76123 | 1.34745 | 7.67079 |
| 1.67229 | 2.85702 | 3.05561 | -4.41193 | 1.4867 | -1.98591 | 0 | 4.29442e-319 |
| 2.85702 | 7.57428 | 5.48574 | -3.08822 | 2.53082 | -0.729092 | 0 | 0 2.27809e-319 |
| 7.36257 | -4.27902 | 3.10837 | -0.947266 | 0 | 4.00243e-320 | -2.04018 | 1.15422 |
| -4.27902 | **41.7035** | -3.95151 | 24.9003 | 0 | -2.59083e-319 | 2.01506 | 4.49036 |
| 1.65326 | -2.10939 | 0 | 4.0375e-319 | -1.32835 | 0.516266 | -0.880799 | -1.92039 |
| -2.10939 | 20.388 | 0 | -2.63215e-318 | 1.35958 | -1.47545 | -1.88263 | -3.19014 |

Cmax: 41.7035 CoreMax: 2 i\_max: 5 j\_max 1

Cmin: -5.21176 CoreMin: 0 i\_min: 0 j\_min 7

All indexes are counted starting with 0 being the first index, and similarly for the core numbers. By looking at the matrix, you can tell that the values for min and max are correct. For the values that are 0, I haven’t figured out a reason why that happens quite yet. The fact that the numbers in X(and hence, Y and C) are randomly generated doesn’t help with figuring that out.

For the other values of J and P, this is what my results were:

J = 100, P = 16

Cmax: 1.77567e+09 CoreMax: 12 i\_max: 1299 j\_max 99

Cmin: -2.01368e-314 CoreMin: 9 i\_min: 996 j\_min 701

J = 50, P = 32

Cmax: 4.51114e+08 CoreMax: 8 i\_max: 449 j\_max 49

Cmin: -1.00796e-314 CoreMin: 17 i\_min: 896 j\_min 751

J = 25, P = 64

Cmax: 1.11711e+08 CoreMax: 22 i\_max: 574 j\_max 24

Cmin: -4.87794e-315 CoreMin: 27 i\_min: 698 j\_min 926

Given that all of these numbers are more or less in the same range, the fact that they all produce similar Cmax and Cmin is to be expected. The sum of the squares of the normalized vector get closer to 1 as J\*P grows, so the denominator of Y shrinks, and hence, the values of Cmax/Cmin grow.

As far as time to go from MPI::Init and MPI::Finalize goes, one trend that was noticed from the output is that the master always takes the longest. This is to be expected, since the master has more to do for a given J,P pair. The master combines all of the random vectors, produces the normalized vector, combines the yMatrix together, as well as searches for Cmax and Cmin globally, none of which the other hosts have to perform. Additionally, the average time for a thread to complete went down as P increased, even though J\*P was the same for each value.