B17100
Rohit Agarwal

CS508 - Introduction to Heterogeneous Computing
Assignment 2

Problem 1

Quick Sort Algorithm

Input file: 'prob1-input.text'
Output file: **'**prob1-output.text'

Assumption: Number of processors << number of elements

Function **writeNumbers** is used to generate n (user's input) random numbers in a file. Time to perform this operation is not included (as this operation will take the same time in both sequential and parallel programs).

The number of processors (value is n) may not be a divisor of size. So, **MPI_Scatter** cannot be used. **MPI_Bcast** is used for broadcasting the input array.

Note 1: Functions **quickSort**, **partition** is handled in the respective individual processors itself over a range of iterations (low to high) to **sort the sub-arrays.**

```
low = (size/numProcess)*rank;
high = (size/numProcess)*(rank+1);
```

numProcess = number of processor
Note 2: Processor (rank = numProcess-1) may have different number of elements.

MPI_Gatherv is used to gather the sorted sub-arrays into one array in the root processor. (but key point is that the whole array is still not sorted).
So, **Merge method** is used to merge the K sorted sub-arrays in the root processor.
Finally, Results are written back into the **output file**.

Output: Output numbers are sorted numbers (same as sequential output) for same input file.
Performance:
Sequential - 0.049 s Parallel - 0.059 s

Time taken by the parallel program is slightly higher than the sequential program because there will be overhead while broadcasting the input array to all the processors and then gather the sorted sub-arrays into the root processor to sort the whole array.

Problem 2

Z Score Normalization

Input file: 'iris.data'
Output file: **'**iris-Normalised.data'

Input-Output is handled in the root(rank =0) processor.
Number of processors (value is n) may not be divisor of size. So, **MPI_Scatter** cannot be used. **MPI_Bcast** is used for broadcasting the column vectors. and **range of iterations (**different for each processor) is used for accessing the column vector in different processors (Alternative method of this approach is **scatterv function**).

Also, For each column vector, **MPI_Allreduce** is used for calculating mean, standard deviation, and passing it back to each processor.
Calculating Z score takes a constant amount of time so it is handled by the respective individual processor itself.

Finally, **MPI_Gatherv** is used for gathering the data(Z-score Normalised column vectors) back into the root processor.
When all the column vectors are normalized and processors reach the same final state(done by using **MPI_Barrier)** then, results are written back into the root processor.

Note: Functions **getSum, getSquareSum** are handled in the respective individual processors itself over a range of iterations (low to high).

```
low = (size/numProcess)*rank;
```

```
high = (size/numProcess)*(rank+1);
```

Output: Normalised column vectors (same as sequential processing)
Performance:
Sequential - 004s
Parallel - 0.000435 s (p = 5)
Parallel - 0.160301 s (p = 10)
Time taken by the parallel program is slightly lower than the sequential program when number of processors was 5. As soon as we increase the value of p(number of processors), data

handled by each processor reduces to a very few. So, time increase as overhead for message passing between the processor's increases.

Problem 3

Note: Value of cluster (K) is set in macros in code to get the appropriated time.

MPI_Bcast is used for broadcasting the value of K(**cluster size**) to all processors.
Number of processors (value is n) may not be a divisor of size. So, **MPI_Scatter** cannot be used. **MPI_Bcast** is used for broadcasting the 4-D vectors. and **range of iterations (**different for each processor) is used for accessing these vectors to find the closest cluster in different processors (Alternative method of this approach is **scatterv function**).

Cluster centers, Cluster Size are synchronized with respect to all the processors and they are accessible to each of them.

Assumption: Number of clusters are very few as compared to the size of data points. So, functions like **getClosestCluster, and getEuclidDistance** are not parallelized individually as their time complexity is **O(K).**
O(K) << O(Size)

Note: Our main goal is to focus on parallelizing a for loop over a data vector with **len=size**. This will obviously reduce the time that a sequential for loop would have taken.

The variable **noChange** is used to decide '**Where to Stop**' and it is synchronized with respect to all the processors.
When all the cluster centers are calculated and processors reach the same final state (**MPI_Barrier)** then they are prompt out on the Stdout.

Output: K Cluster centers
Performance:
When K = 5,
Sequential - 0.006 s

Parallel - 0.000543 s (p=5)
Parallel - 0.0012 s (p=8)

Time taken by the parallel program is slightly lower than the sequential program when the number of processors was 5. As soon as we increase the value of p(number of processors), data handled by each processor reduces to a very few. So, time increase as overhead for message passing between the processor's increases.

New Concepts/Uses Learned during this Assignment:
- MPI_IN_PLACE
- MPI_scatterv/MPI_Gatherv
- MPI_Allreduce
- MPI_Barrrier