B17100 **Date: 25 July**
Rohit Agarwal

Heterogenous Computing - CS508
Assignment 1 Report

Answer 1:

Loop 1: In writeNumbers (line 16 to 19)

```
        for(int i=0;i<n;i++){

            fprintf(ptr, "%d ",rand());
        }
```

I/O Streams are sequential programs, and cannot be parallelized.

Loop 2:In readNumbers (Line 29 to 32)

```
    while(fscanf(ptr, "%d ", &num)!=EOF){
        Array[i] = num;
        i++;
    }
```

I/O Streams are sequential programs, and cannot be parallelized.
There is also an interloop dependency.

Loop : Line 42 to 50

```
    for(j=j;j<high;j++){
        if (Array[j]<pivot){
            i++;
            //swap
            int temp = Array[i];
            Array[i] = Array[j];
            Array[j] = temp;
        }
    }
```

Each iteration depends on the value of **I** from the previous iteration. So, I think, this loop is not parallelizable.

```
void quickSort(int low, int high, int Array[]){                 //Line 1
   if (low<high){                                               //Line 2
       int partitionElement = partition(Array, low, high);        //Line 3
       quickSort(low, partitionElement-1, Array);               //Line 4
       quickSort(partitionElement+1, high, Array);                //Line 5
   }                                                            //Line 6
}                                                               //Line 7
```

Here, Using **data level parallelism,** we can execute the line 4 and line 5 independently on other processors parallelly. Line 4 and line 5 are not dependent on each other. But here if we noticed then there is a loop carried dependency since each recursive call depends on the low and high values of the previous call.
Note: Each processor can call two other processors to execute the sub-process. Then Merge can happen at the end.

Loop : line 69 to 70

```
       for(int i=0;i<n;i++)
           printf("%d ", Array[i]);
```

I/O Streams are sequential programs, and cannot be parallelized.

Loop : Line 86 to 88

```
for(int i=0;i<n;i++){
       fprintf(ptr, "%d ", Array[i]);
   }
```

I/O Streams are sequential programs, and cannot be parallelized.

Problem 2:
Loop 1: In getMean function (Line 7 to 9)

```
   for(int j=0;j<n;j++){
       mean = mean + Array[j];
   }
```

The above loop has a loop carried dependency. So, each iteration cannot be processed independently but by using expansion and reduction techniques (**data level parallelism)** we can find the sum of elements of each subarray parallelly. Then we will sum the result obtained from each sub-array.

Loop 2: In getStdDev function (Line 17 to 19)

```
  for(int j=0;j<n;j++){
      stdDevSum = stdDevSum + (Array[j]-mean)*(Array[j]-mean);
  }
```

The above loop has loop carried dependency which can be avoided to some extent by using two for loops.

```
int getMeanDiff[n]; int stdDevSum = 0;
For (int j=0; j<n; j++){                                    //completely parallelizable
        getMeanDiff[j] = (Array[j] - mean)*(Array[j] - mean)
}
for (int j=0; j<n;j++){                                     //partial parallelizable
        stdDevSum = stdDevSum + getMeanDiff[j];
}
```

The first loop is completely parallelizable (So, each instruction can be processed independently) whereas the second loop can be partially parallelized using **expansion and reduction technique (data level parallelism).**

Loop3: Line 37 to 44

```
  while(fgets(str,1024, ptr) != NULL){
      sscanf(str,"%lf,%lf,%lf,%lf,%s\n",&col1, &col2, &col3, &col4,str);
      firstCol[i]=col1;
      secondCol[i]=col2;
      thirdCol[i]=col3;
      fourthCol[i]=col4;
      i++;
  }
```

This loop can be converted into two loops (one for I/O, one for assignments/computations).

```
Char *str[150];
Char line[1024]; int i=0;
while(fgets(line, 1024, ptr)!=NULL){                //I/O Stream - not parallelizable
        str[i] = line;
        i++;
}
```

```
For (int i=0;i<150;i++){                                                    //Parallelizable
        sscanf(str[i], "%lf,%lf,%lf,%lf,%s\n", &col1, &col2, &col3, &col4, str);       //Line 1
        firstCol[i] = col1;                                                             //Line 2
        secondCol[i] = col2;                                                            //Line 3
        thirdCol[i] = col3;                                                             //Line 4
        fourthCol[i] = col4;                                                            //Line 5
}
```

This loop is parallelizable as it is not having a loop carried dependency. So, each iteration can be separately processed. but, it has an interloop dependency. Line 2, 3, 4, and 5 are dependent on line 1 but they themselves are independent of each other. So, After execution of line 1, line 2, 3, 4 & 5 can be parallelly executed.

Loop 4: Line 66 to

```
while(i!=size){
        col1 = getZScore(firstCol[i], meanCol1, stdCol1);
        col2 = getZScore(secondCol[i], meanCol2, stdCol2);
        col3 = getZScore(thirdCol[i], meanCol3, stdCol3);
        col4 = getZScore(fourthCol[i], meanCol4, stdCol4);
        snprintf(str,1024, "%.2f,%.2f,%.2f,%.2f\n", col1, col2, col3,
col4);
        fputs(str, ptr);
        i++;
    }
```

This loop can also be divided into two loops (One is parallelizable whereas the second one is not).

```
Char *str[150];
Char line[1024];
int i=0;
while(i!=size){                                                         //Parallelizable
        col1 = getZScore(firstCol[i], meanCol1, stdCol1);              //Line 1
        col2 = getZScore(secondCol[i], meanCol2, stdCol2);             //Line 2
        col3 = getZScore(thirdCol[i], meanCol3, stdCol3);              //LIne 3
        col4 = getZScore(fourthCol[i], meanCol4, stdCol4);             //Line 4
        snprintf(line, 1024, "%.2f,%.2f,%.2f,%.2f\n", col1, col2, col3, col4);  //Line 5
        str[i] = line;
        i++;
}
```

This loop is completely parallelizable because of no loop carried dependency. In inter loop, Line 1, 2,3 & 4 can be executed parallelly in each iteration. Line 5 has to be executed after the execution of these lines as it directly depends on them.

```
i =0;
while(i!=size){                                    //Sequential - Not parallelizable
        fputs(str[i], ptr)
        i++;
}
```

Problem 3:

Loop 1: in getEuclidDistance (Line 17 to line 19)

```
   for (int i=0;i<n;i++){
       distance = distance +
(point[i]-centroid[i])*(point[i]-centroid[i]);
   }
```

```
Int Array[n]; int distance = 0;
For (int i=0; i<n; i++){                                //complete parallelizable
        Array[i] = (point[i]-centroid[i])*(point[i]-centroid[i]);
}


For (int i=0; i<n; i++){                                //partial parallelizable
        distance = distance + Array[i];
}
```

The first loop is completely parallelizable (So, each instruction can be processed independently) whereas the second loop can be partially parallelized using **expansion and reduction technique (data level parallelism).**

Loop 2: In getClosestCluster (Line 27 to 33)

```
   for(int j=0;j<K;j++){
       float temp = getEuclidDistance(numCols, centroids[j], point);
       if (temp<distance){
           distance = temp;
           cluster = j;
       }
   }
```

Int distance[K];

```
for(int j=0; j<K; j++){                                              //Completely Parallelizable
        distance[j] = getEuclidDistance(numCols, centroids[j], point);
}

Int minDistance = INT_MAX, cluster = 0;
for(int j=0; j<K; j++){                                              //Partial Parallelizable
        If (distance[j] < minDistance){
                minDistance = distance[j];
                cluster = j;
        }
}
```

The first loop is completely parallelizable(No loop carried/level dependency), whereas the second loop is partially parallelizable and it can be parallelized using expansion, reduction technique.

Loop 3: From line 61 to 67

```
//Initialization of cluster centers
   for(int i=0;i<K;i++){
       clusterCenters[i][0] = clusterPoints[i][0];
       clusterCenters[i][1] = clusterPoints[i][1];
       clusterCenters[i][2] = clusterPoints[i][2];
       clusterCenters[i][3] = clusterPoints[i][3];
   }
```

This is already parallelizable. (No loop level and inter loop dependency)

Loop 4: Line 69 to 71

```
   //Initially size of each cluster 0
   for(int i=0;i<K;i++)
       clusterSize[i]=0;
```

This is already parallelizable.

Loop 5: While loop (Line 75)
This loop(outer while loop) cannot be parallelized because cluster means are updating in each of the iterations, then it is going to be used in the next iteration.

Loop 6: Line 77 to 83

```
for(int i=0;i<size;i++){
  int clusterIndex = getClosestCluster(K, clusterCenters,
clusterPoints[i]);
  if (clusterIndex!=(int)clusterPoints[i][4]){
           clusterPoints[i][4]= clusterIndex;
           noChange = 0;
  }
}
```

```
Int clusterIndexes[size];
for(int i=0; i<size; i++){                                   //Completely parallelizable
        clusterIndexes[i] = getClosestCluster(K, clusterCenters, clusterPoints[i]);
}

for(int i=0; i<size; i++){                                   //Completely parallelizable
        If (clusterIndexes[i] != (int)clusterPoints[i][4]){
                clusterPoints[i][4] = clusterIndexes[i];
                noChange = 0;
        }
}
```

Both the loops are completely parallelizable because they do not have loop level dependencies. Also, In first loop, function **getClosestCluster** is itself parallelizable.

Loop 7 & 8: line 87, line 91
These outer loops individually can be parallelized because they do not contain a loop carried dependency.
Note: I used two for loops for updating the cluster centers but it can also be done in one for loop. Since here our main aim is to figure it out where parallel processing can be done. I didn't pay attention the effects of two for loops on complexity.

Loo 9: Line 93 to 95

```
        for (int k=0;k<numCols;k++){
             newCentroid[k] = newCentroid[k] + clusterPoints[j][k];
        }
```

Here, numCols = 4. So, **loop unrolling can be used** to reduce the loop overhead.
They are parallelizable also.

newCentroid[0] = newCentroid[0] + clusterPoints[j][0];
newCentroid[1] = newCentroid[1] + clusterPoints[j][1];
newCentroid[2] = newCentroid[2] + clusterPoints[j][2];
newCentroid[3] = newCentroid[3] + clusterPoints[j][3];

Loop 10: Line 101 to 107

```
           //Update the cluster centers (centroids)
           for (int k=0;k<numCols;k++){
                    newCentroid[k] = newCentroid[k]/(float)pointsCount;
                   clusterCenters[i][k] = newCentroid[k];


           }
```

numCols = 4, Use the approach of loop 9(**Loop unrolling**).

Loop 11: 114 to 123

```
 //Prompt out clusters if exists
 for(int i=0;i<K;i++){
   if (clusterSize[i]!=0){
      printf("cluster %d: (%f,%f,%f,%f )\n",i, clusterCenters[i][0],
clusterCenters[i][1], clusterCenters[i][2], clusterCenters[i][3]);
   }
   else{
       printf("cluster %d: does not exist (No vector is associated)\n",
i);
   }
 }
```

I/O stream is not parallelizable.