PARALLELIZATION OF CLOSEST PAIR OF POINTS PROBLEM

by Sidhant Bansal

Harshit Rajvaidya

Kanika Saraswat

CSS 534: Parallel Programming in Grid and Cloud

Dr. Munehiro Fukuda

University of Washington Bothell

Bothell, WA

12/18/2021

Table of Contents

Introduction **3**

Solutions **3**

Message Passing Interface (MPI) 4

Execution Snapshots 4

Performance Analysis 5

Code Snippets 6

Map-Reduce 7

Execution Snapshots 8

Code Snippets 10

Execution Analysis 10

Multi-Agent Spatial Simulation (MASS) 10

Code Snippets 12

Execution Analysis 12

Spark 12

Execution Snapshots 13

Code Snippet 15

Performance Analysis 16

Programmability Analysis 17

MapReduce 18

Spark 18

MPI 18

Multi-Agent Spatial Simulation (MASS) 19

References **19**

I. Introduction

It is a problem of computational geometry. We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in several applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a collision.

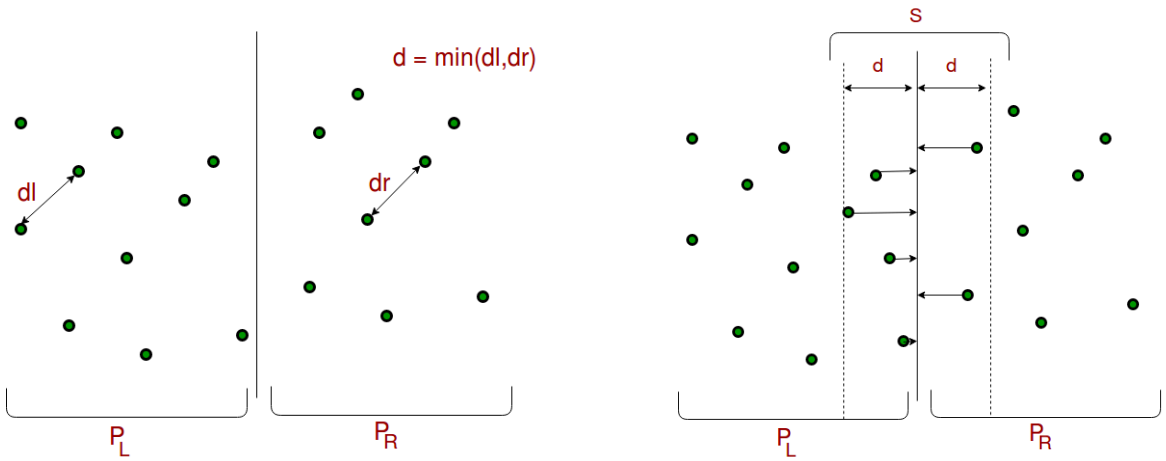
II. Solutions

There are 2 approaches to solve this problem:

1) Brute Force: We compute the distance of each pair and return the smallest distance.

2) Divide and Conquer: Initially the points are sorted according to the x coordinates. The points are recursively divided into two halves and for every recursive call, minimum distance is calculated. From the last step, we have an upper bound of minimum distance. We consider the pairs within the minimum distance in such a way that one point is from the left half and the other one is from the right half.

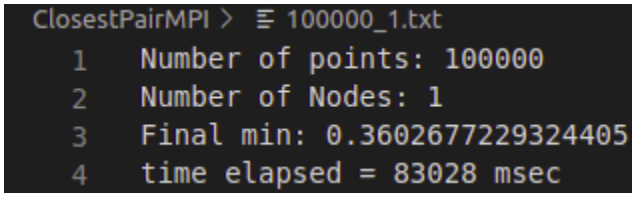
[1]

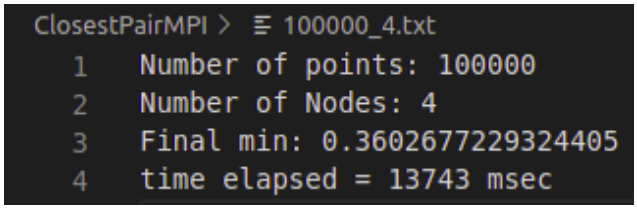


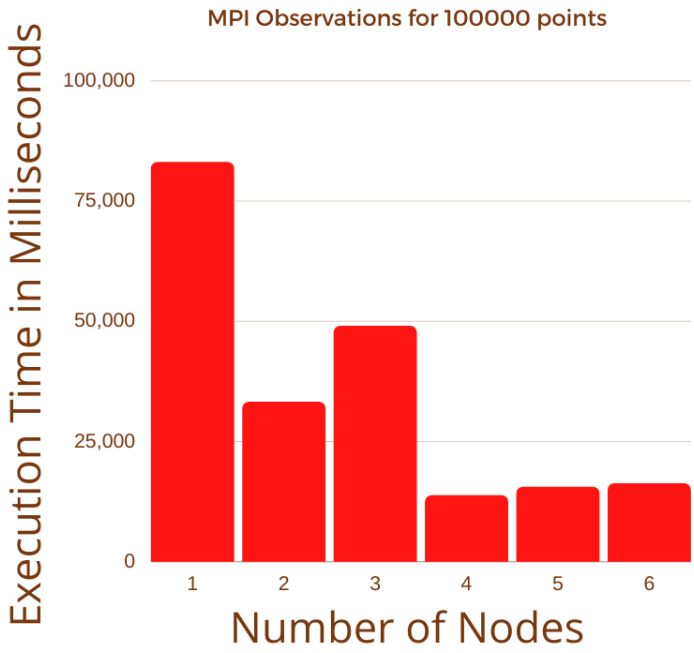
Message Passing Interface (MPI)

In the Message Passing Interface (MPI) approach, the list of x sorted points was divided according to the number of ranks. Each rank used the approach of divide and conquer to find the closest pair of points for their set after that each rank sent its minimum to the rank 0 which in turn compared all the minimums to determine the global minima. Once the global minima for all respective ranks were found then it was passed to all the ranks once again and all the adjacent ranks with the help of communication between them then took care of the case in which the closest pair of points might have gone to 2 different ranks. For example, if rank 0 had a point of the closest pair and rank 1 had the other point then in that case rank 0 would create an array of all the points that are within the minimum and send it to rank 1 then similarly rank 1 would also create an array of points within the minimum, combine these 2 arrays and again find the minimum distance for those points in the array.

Execution Snapshots







Performance Analysis

Observations for 100000 points:

Number of Nodes	Shortest Distance	Time Taken (msec)	Performance Improvement
1	0.3602677229324405	83028	1.0
2	0.3602677229324405	33206	2.5
3	0.3602677229324405	48980	1.7
4	0.3602677229324405	13743	6.0
5	0.3602677229324405	15490	5.3
6	0.3602677229324405	16227	5.1

Code Snippets

```
for (int rank = 0; rank < nprocs; rank++ ) {
    rows[0] = ( rank < extra ) ? averows + 1 : averows;
    if(rank == 0){ // This is useful only when rank 0 has to take care of the remainder rows.
        receivedXSorted = Arrays.copyOfRange(xSortedPoints,0, rows[0]);
    }
    if(rank != 0){
        MPI.COMM_WORLD.Send(offset, 0, 1, MPI.INT, rank, mtype);
        MPI.COMM_WORLD.Send(rows, 0, 1, MPI.INT, rank, mtype );
        MPI.COMM_WORLD.Send(xSortedPoints, offset[0], rows[0], MPI.OBJECT, rank, mtype);
    }
    offset[0] += rows[0];
}

ySortedPoints = receivedXSorted.clone();

masterMinimum = GetMinDistance(receivedXSorted, ySortedPoints, 0, receivedXSorted.length-1);//Rank 0 caluclating
the minimum distance for their set of points.

Double tempMin = 0.0;
for ( int source = 1; source < nprocs; source++ ) {
    MPI.COMM_WORLD.Recv(tempMin,0, 1, MPI.DOUBLE, source, mtype); // Rank 0 receiving minimum from every rank to
    determine the global minimum for all ranks
    masterMinimum = Math.min(masterMinimum, tempMin);
}
```

```
else{
    mtype = TAG_FROM_MASTER;
    MPI.COMM_WORLD.Recv(offset, 0, 1, MPI.INT, MASTER, mtype );
    MPI.COMM_WORLD.Recv(rows, 0, 1, MPI.INT, MASTER, mtype );
    receivedXSorted = new PointsGrabber[rows[0]];
    MPI.COMM_WORLD.Recv(receivedXSorted, 0, rows[0], MPI.OBJECT, MASTER, mtype); // Respective rank receiving their
    part of the data.

    ySortedPoints = receivedXSorted.clone();

    rankMinimum = GetMinDistance(receivedXSorted, ySortedPoints, 0, receivedXSorted.length-1); //Ranks caluclating
    minimum distance for their set of points.

    MPI.COMM_WORLD.Send(rankMinimum, 0, 1, MPI.DOUBLE, MASTER, mtype);// Ranks sending their respective minimum to
    rank 0

}
if(myrank == 0){
    return masterMinimum; // Rank 0 returning the minimum
}
```

```
int nextBorderIndex = borderIndex + rows;//Calculating the border index
double x = points[0].getX();
List<Points> pointsList = new ArrayList<>();
for (int i = 0; i < points.length; i++) {
    if (points[i].getX() > x + minimum) {
        break;
    }
    if (points[i].getX() <= x + minimum) {
        pointsList.add(points[i]); //Points within the minimum are added in
        the array list that are then converted into array before sending
    }
}

sizeOfPointsList[0] = pointsList.size();
MPI.COMM_WORLD.Send(sizeOfPointsList, 0, 1, MPI.INT, rank - 1,
TAG_FROM_SLAVE);//Rank sending the size of points to the adjacent rank
if (!pointsList.isEmpty()) {
    pointsArr = new Points[pointsList.size()];
    pointsArr = pointsList.toArray(pointsArr);
    MPI.COMM_WORLD.Send(pointsArr, offset[0], pointsArr.length, MPI.OBJECT,
    rank - 1, TAG_FROM_SLAVE);// Rank sending the array object to the
    adjacent rank
}
```

```
MPI.COMM_WORLD.Recv(sizeOfPointsList, 0, 1, MPI.INT, rank + 1,
TAG_FROM_SLAVE);//Receiving the size of the points from the adjacent rank
if (sizeOfPointsList[0] != 0) {
    pointsArr = new Points[sizeOfPointsList[0]];
    MPI.COMM_WORLD.Recv(pointsArr, 0, sizeOfPointsList[0], MPI.OBJECT, rank
    + 1, TAG_FROM_SLAVE);//Receiving the array object from the adjacent rank

    List<Points> pList = new ArrayList<>();
    Double x = points[0].getX();
    for(int i = pointsArr.length - 1; i >= 0; i--) {
        if (pointsArr[i].getX() > x + minimum) {
            break;
        }
        if (pointsArr[i].getX() <= x + minimum) {
            pList.add(pointsArr[i]);
        }
    }
```

```
//Calculating the minimum distance of points after combing the two arrays of the adjacent ranks.
for (int i = 0; i < pointsArr.length; i++ ) {
    for(int j = i+1; j < Math.min(pointsArr.length , i+7) ; j++) {
        minimum = Double.min(minimum, EuclideanDistance(pointsArr[i].getX(), pointsArr[i].getY(),
        pointsArr[j].getX(), pointsArr[j].getY()));
    }
}
```

Map-Reduce

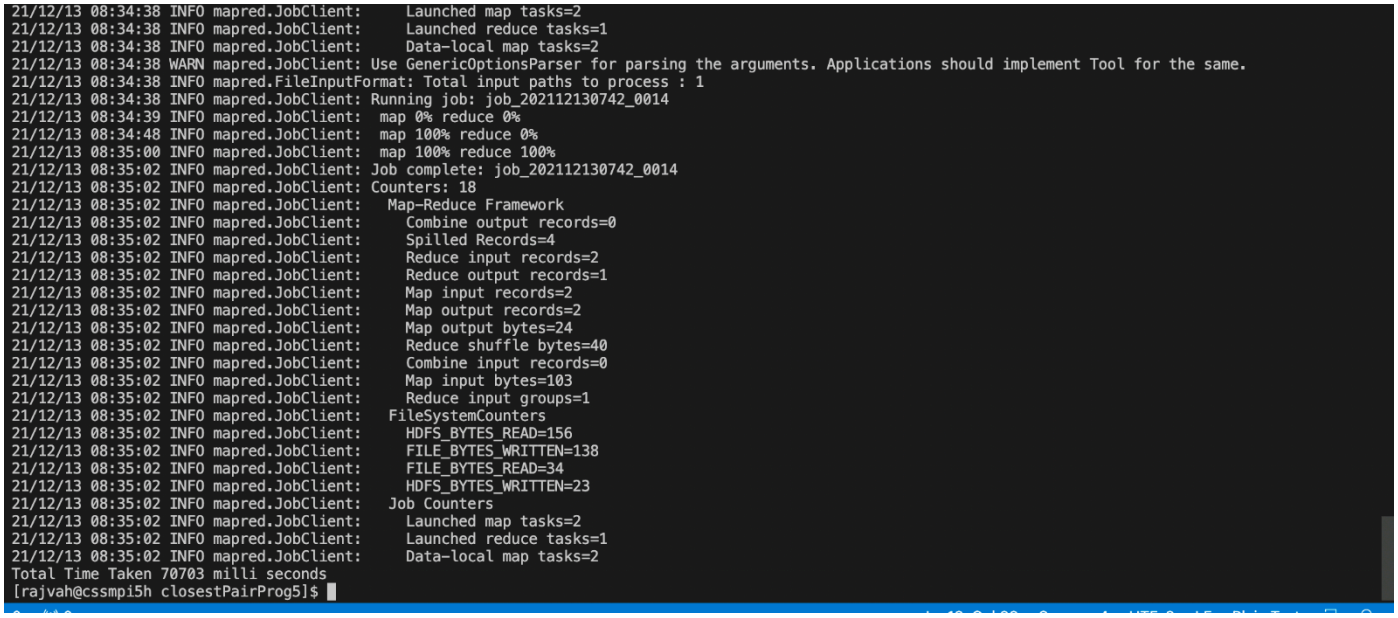
In the map-reduce approach we have followed a divide and conquer approach with multiple map-reduce jobs.

In the first job, we are making use of the default sort available for map-reduce jobs to sort the coordinates along the x-axis. In the second job, we are mapping the points as a group with the key as the input split id. The more the input splits the greater number of subgroups it will form. In the reduce section we are trying to calculate the local minima for all the subgroups (splits) it has created.

In the third map-reduce job we are combining the splits to finally calculate the global minimum.

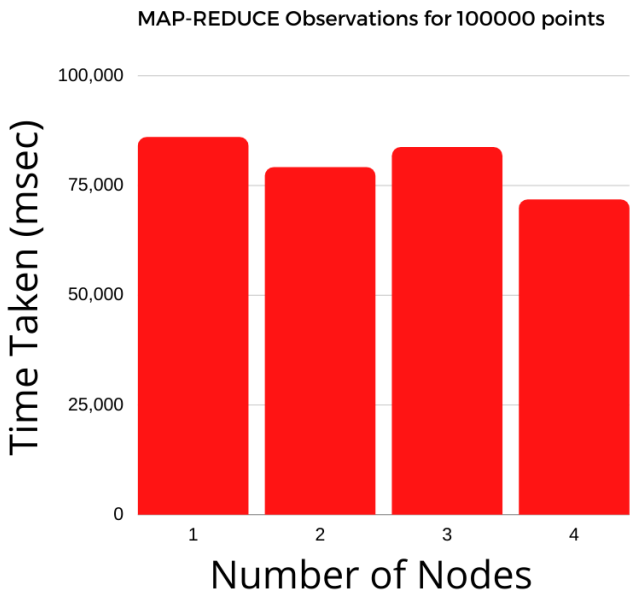
Execution Snapshots

100,000 data points:



10000 data points:

```
21/12/13 08:39:32 INFO mapred.JobClient: Counters: 19
21/12/13 08:39:32 INFO mapred.JobClient:   Map-Reduce Framework
21/12/13 08:39:32 INFO mapred.JobClient:     Combine output records=0
21/12/13 08:39:32 INFO mapred.JobClient:     Spilled Records=4
21/12/13 08:39:32 INFO mapred.JobClient:     Reduce input records=2
21/12/13 08:39:32 INFO mapred.JobClient:     Reduce output records=1
21/12/13 08:39:32 INFO mapred.JobClient:     Map input records=2
21/12/13 08:39:32 INFO mapred.JobClient:     Map output records=2
21/12/13 08:39:32 INFO mapred.JobClient:     Map output bytes=24
21/12/13 08:39:32 INFO mapred.JobClient:     Reduce shuffle bytes=0
21/12/13 08:39:32 INFO mapred.JobClient:     Combine input records=0
21/12/13 08:39:32 INFO mapred.JobClient:     Map input bytes=102
21/12/13 08:39:32 INFO mapred.JobClient:     Reduce input groups=1
21/12/13 08:39:32 INFO mapred.JobClient:   FileSystemCounters
21/12/13 08:39:32 INFO mapred.JobClient:     HDFS_BYTES_READ=154
21/12/13 08:39:32 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=138
21/12/13 08:39:32 INFO mapred.JobClient:     FILE_BYTES_READ=34
21/12/13 08:39:32 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=23
21/12/13 08:39:32 INFO mapred.JobClient:   Job Counters
21/12/13 08:39:32 INFO mapred.JobClient:     Launched map tasks=2
21/12/13 08:39:32 INFO mapred.JobClient:     Launched reduce tasks=1
21/12/13 08:39:32 INFO mapred.JobClient:     Rack-local map tasks=1
21/12/13 08:39:32 INFO mapred.JobClient:     Data-local map tasks=1
Total Time Taken 71754 milli seconds
[rajvah@cssmpi5h closestPairProg5]$
```



Code Snippets

```
public static class PointsReducer extends MapReduceBase implements Reducer<Text, Text, Text, DoubleWritable> {
    public void reduce(Text key, Iterator<Text> values, OutputCollector<Text, DoubleWritable> output, Reporter reporter) thr
        ArrayList<ArrayList<Double>> points = new ArrayList<ArrayList<Double>>();

        while (values.hasNext()) {
            ArrayList<Double> point = new ArrayList<>();
            String[] pointStr = values.next().toString().split("\\s+");
            point.add(Double.parseDouble(pointStr[0]));
            point.add(Double.parseDouble(pointStr[1]));

            points.add(point);
        }
        Double min = Double.MAX_VALUE;
        for( int i =0 ; i < points.size(); i++){
            for(int j=i+1 ; j < points.size(); j++){
                min = Double.min(EuclideanDistance(points.get(i), points.get(j)), min);
            }
        }

        output.collect(key, new DoubleWritable(min));
    }
}
```

Execution Analysis

Number of Nodes	Shortest Distance	Time Taken (msec)	Performance Improvement
1	0.3602677229324405	85980	< 1.0
2	0.3602677229324405	79090	1.08
3	0.3602677229324405	83667	~1
4	0.3602677229324405	71754	1.19

Multi-Agent Spatial Simulation (MASS)

In Multi-Agent Spatial Simulation (MASS) the parallelization technique was a simple one. We spawned 1 agent for each of our points in the input file and then a callAll for each agent to try and find their closest pair. The approach was to go around in a circle (x- a) ^2 + (y - b)^2

for every point in the place.

Then we would compute the distance that is minimum along all the values reported by the agent. Since the execution gets each agent to compute their Neighbour independently, the more the number of point/agents the higher the chance of collision between the points. However, the performance impact has not been great in this implementation. I believe this is due to a lack of insight on my end regarding how the collision of agents can be considered.
Once that is implemented the agents would stop checking any other point as soon as the first one hits.



Code Snippets

```
MASS.setLoggingLevel( LogLevel.DEBUG );

MASS.init();
MASS.getLogger().debug("MASS started");

int size = inputPoints.size();

Places places = new Places( 1, ClosestPairPlaces.class.getName(), null, 100, 100);

ArgsToAgents args2agents = new ArgsToAgents( inputPoints );

Agents agents = new Agents(2, ClosestPairAgent.class.getName(), (Object) args2agents, places, size);

agents.callAll(ClosestPairAgent.INIT);
Object resp = agents.callAll(ClosestPairAgent.CHECK, null);

Double min = Double.MAX_VALUE;

for (Object ob : (Object[]) resp) {

    if (ob != null) {
        List<Double> mins = (List< Double>) ob;

        for( Double x : mins){
            min = Double.min(min, x);
        }
    }
}
```

Execution Analysis

Number of Nodes	Time Taken (msec)	Performance Improvement
2	426782	< 1
3	309873	< 1
4	279801	< 1

Spark

In Spark we are making a Cartesian product of the JavaRDD object of the Coordinates list with itself. Then we get a JavaPairRDD object in which each coordinate is mapped with all

the other points in the plane. Then we calculate the Euclidean distance between each coordinate and store it in the JavaDoubleRDD object, which now contains a list of Euclidean distances between all points. Then we simply use JavaDoubleRDD.min() to print the minimum. We will most definitely encounter cases where the point is mapped with itself. The Euclidean distance of a point with itself is zero. This result could hamper our calculations. To resolve this problem, we stored the distance as Double.MAX_VALUE(). By doing this, we are able to omit the distance of the point with itself when it is being stored in JavaDoubleRDD.

This code works well for a small dataset i.e., 100 points. The performance gets exponentially slower with a larger dataset since every point is mapped with every other point which is unnecessary since calculating the smallest distance between points does not need to have a very subset. In MPI implementation, we are mapping a subset of points with itself, so the computation overhead is much lower. If given more time, we could look for ways to complete our calculations with a smaller subset parallelly.

Execution Snapshots

100 points, 4 nodes 8 cores:

```
2021-12-13 12:17:49 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:53) finished in 2.540
2021-12-13 12:17:49 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 2.591001 s
Minimum distance: 0.6360752942851988
Total time elapsed: 2713
```

100 points, 4 nodes 6 cores:

```
2021-12-13 12:18:00 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:53) finished in 2.64
2021-12-13 12:18:00 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 2.705127 s
Minimum distance: 0.6360752942851988
Total time elapsed: 2799
2021-12-13 12:18:00 INFO  AbstractConnector:318 - Stopped Spark@5c77053b{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

100 points, 4 nodes 4 cores:

```
2021-12-13 12:21:07 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:53) finished in 2.193 s
2021-12-13 12:21:07 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 2.243180 s
Minimum distance: 0.6360752942851988
Total time elapsed: 2328
2021-12-13 12:21:07 INFO  AbstractConnector:318 - Stopped Spark@56356e9e{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

100 points, 4 nodes 2 cores:

```
2021-12-13 12:22:21 INFO  TaskSchedulerImpl:54 - Removed TaskSet 0.0, whose tasks have all completed, from pool
2021-12-13 12:22:21 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:53) finished in 2.364
2021-12-13 12:22:21 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 2.413571 s
Minimum distance: 0.6360752942851988
Total time elapsed: 2482
2021-12-13 12:22:21 INFO  AbstractConnector:318 - Stopped Spark@432d8459{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

100 points, 4 nodes 1 core:

```
2021-12-13 12:23:43 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:53) finished in 2.036 s
2021-12-13 12:23:43 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 2.082310 s
Minimum distance: 0.6360752942851988
Total time elapsed: 2154
2021-12-13 12:23:43 INFO  AbstractConnector:318 - Stopped Spark@310ee053{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

100k points, 4 nodes 4 cores

```
2021-12-13 09:44:34 INFO  TaskSchedulerImpl:54 - Removed TaskSet 0.0, whose tasks have all completed, from pool
2021-12-13 09:44:34 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:53) finished in 4304.969 s
2021-12-13 09:44:34 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 4305.023701 s
Minimum distance: 0.3602677229324405
Total time elapsed: 4305113
2021-12-13 09:44:34 INFO  AbstractConnector:318 - Stopped Spark@287f94b1{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
2021-12-13 09:44:34 INFO  SparkUI:54 - Stopped Spark web UI at http://cssmpi7h.uwb.edu:4040
2021-12-13 09:44:34 INFO  StandaloneSchedulerBackend:54 - Shutting down all executors
2021-12-13 09:44:34 INFO  CoarseGrainedSchedulerBackend$DriverEndpoint:54 - Asking each executor to shut down
2021-12-13 09:44:34 INFO  MapOutputTrackerMasterEndpoint:54 - MapOutputTrackerMasterEndpoint stopped!
```

100k points, 4 nodes 6 cores

```
2021-12-13 12:09:59 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 6847.292617 s
Minimum distance: 0.3602677229324405
Total time elapsed: 6847390
2021-12-13 12:09:59 INFO  AbstractConnector:318 - Stopped Spark@5c77053b{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

100k points 4 nodes 8 cores

```
2021-12-13 12:06:33 INFO  TaskSetManager:54 - Finished task 3.0 in stage 0.0 (TID 3) in 6587506 ms on 10.158.82.96 (ex
2021-12-13 12:06:33 INFO  TaskSchedulerImpl:54 - Removed TaskSet 0.0, whose tasks have all completed, from pool
2021-12-13 12:06:33 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:53) finished in 6588.261 s
2021-12-13 12:06:33 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:53, took 6588.470528 s
Minimum distance: 0.3602677229324405
Total time elapsed: 6588648
2021-12-13 12:06:33 INFO  AbstractConnector:318 - Stopped Spark@30b34287{HTTP/1.1,[http/1.1]}{0.0.0.0:4041}
2021-12-13 12:06:33 INFO  SparkUI:54 - Stopped Spark web UI at http://cssmpi7h.uwb.edu:4041
2021-12-13 12:06:33 INFO  StandaloneSchedulerBackend:54 - Shutting down all executors
```

100k points 6 nodes 6 cores

```
 10.158.82.65 (executor 3) (4/4)
2021-12-13 00:59:20 INFO  DAGScheduler:54 - ResultStage 0 (min at ClosestPairOfPoints.java:51) finish
ed in 6371.768 s
2021-12-13 00:59:20 INFO  TaskSchedulerImpl:54 - Removed TaskSet 0.0, whose tasks have all completed,
 from pool
2021-12-13 00:59:20 INFO  DAGScheduler:54 - Job 0 finished: min at ClosestPairOfPoints.java:51, took
6371.914920 s
Minimum distance: 0.3602677229324405
Total time elapsed: 6372056
2021-12-13 00:59:20 INFO  AbstractConnector:318 - Stopped Spark@525d79f0{HTTP/1.1,[http/1.1]}{0.0.0.0
:4042}
2021-12-13 00:59:20 INFO  SparkUI:54 - Stopped Spark web UI at http://cssmpi1h.uwb.edu:4042
2021-12-13 00:59:20 INFO  StandaloneSchedulerBackend:54 - Shutting down all executors
2021-12-13 00:59:20 INFO  CoarseGrainedSchedulerBackend$DriverEndpoint:54 - Asking each executor to s
hut down
2021-12-13 00:59:20 INFO  MapOutputTrackerMasterEndpoint:54 - MapOutputTrackerMasterEndpoint stopped!
2021-12-13 00:59:20 INFO  MemoryStore:54 - MemoryStore cleared
2021-12-13 00:59:20 INFO  BlockManager:54 - BlockManager stopped
```

Code Snippet

```
JavaPairRDD<Coordinates,Coordinates> pointsPairRDD = points.cartesian(points);
//creating a pair RDD of the coordinates, each point is a pair with every other point.


JavaDoubleRDD distances = pointsPairRDD.mapToDouble(new
DoubleFunction<Tuple2<Coordinates, Coordinates>>() { //storing the distances in
JavaDoubleRDD
    @Override
    public double call(Tuple2<Coordinates, Coordinates> t) throws Exception {
      double distance = euclideanDistance(t._1, t._2);
```

```
            if(distance == 0) {
                return Double.MAX_VALUE; //in case the point calculates the distance with itself, the distance will come as zero. So turning it into
                                        // MIX_VALUE of double so that it will not be considered in finding the minimum distance
            } else {
                return distance;
            }
        }

    });
```

Performance Analysis

Number of Nodes: 4

Number of Cores	Dataset (number of points)	Execution Time (ms)	Performance Improvement
1	100000	8798824	1
2	100000	8278111	0.94
4	100000	4305113	2.04
6	100000	6847390	1.28

8	100000	6588648	1.33
1	100	2154	1
2	100	2482	0.86
4	100	2328	0.92
6	100	2799	0.76
8	100	2713	0.79

Programmability Analysis

	MPI	Spark	MapReduce	MASS
Number of lines boilerplate code	14	3	40	12
Ease of writing (1 easy - 10 hard)	5.5	8	7	10

Ease of readability (1 easy - 10 hard)	3.5	9	3	3

MapReduce

Understanding the implementation of Map-reduce was a slightly tricky task as we had to employ the use of multiple map-reduce jobs. However, writing the code was a fairly simple task as map-reduce is quite straightforward in terms of what the mapper should be like and what the reducer should work with. Getting input splits and working with the divided sorted points was slightly challenging initially.

Spark

This code was easier to write, the entire implementation was completed in 3 lines. The test cases took longer to run since the code is not optimum enough to handle a large dataset.

MPI

The border point calculation was difficult to implement and resolve, but the code ran quite quickly and had the best performance out of all of the other parallelization strategies.

Multi-Agent Spatial Simulation (MASS)

The MASS implementation was a difficult one to crack. To go through the concept of places and agents and how the agents behave in a place, how the agents are mapped, how each agent would interact with other agents was tricky. However, the programming portion of MASS was fairly easy once you understand what each particular object is supposed to work like and how each agent would go on in the place.

III. References

[1] GeeksforGeeks:

https://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/