AGENT BASED PARALLELIZATION OF COMPUTATIONAL GEOMETRY ALGORITHMS

Saranya Gokulramkumar

Summary Report of work done as part of CSS 700: Master’s Thesis

Master of Science in Computer Science & Software Engineering

University of Washington, Bothell
Autumn, 2019

Committee:
Dr. Munehiro Fukuda, Chair
Dr. Clark Olson, Committee Member
Dr. Min Chen, Committee Member
Table of Contents

1. INTRODUCTION .................................................................................................................. 4

2. CLOSEST PAIR OF POINTS IMPLEMENTATION ............................................................ 4
   2.1 CLOSEST PAIR OF POINTS ALGORITHM ...................................................................... 4
   2.1.1 SPARK & MAPREDUCE ALGORITHM ...................................................................... 4
   2.1.2 MASS ALGORITHM .................................................................................................. 5
   2.2 RESULTS ....................................................................................................................... 6
   2.2.1 PROGRAMMABILITY ............................................................................................... 6
   2.2.2 EXECUTION PERFORMANCE .................................................................................. 7

3. CONVEX HULL IMPLEMENTATION .................................................................................... 9
   3.1 CONVEX HULL ALGORITHM ...................................................................................... 9
   3.1.1 SPARK ALGORITHM ................................................................................................ 9
   3.2 RESULTS ....................................................................................................................... 11

4. VORONOI DIAGRAM IMPLEMENTATION .................................................................... 11
   4.1 VORONOI DIAGRAM ALGORITHM .......................................................................... 11
   4.1.1 SPARK ALGORITHM .............................................................................................. 11
   4.1.2 MASS ALGORITHM ................................................................................................ 13
   4.2 RESULTS ....................................................................................................................... 13

5. FUTURE IMPROVEMENTS ............................................................................................... 15
   5.1 CLOSEST PAIR OF POINTS ...................................................................................... 15
   5.2 VORONOI DIAGRAM ................................................................................................. 15

6. CURRENT STATUS ............................................................................................................. 16

7. PROJECT PACKAGE .......................................................................................................... 16

8. CONCLUSION .................................................................................................................... 16
AGENT BASED PARALLELIZATION OF COMPUTATIONAL GEOMETRY ALGORITHMS

1. Introduction

Computational complexity is of significant importance in computational geometry. In practical scenarios very large datasets come into play, like in the fields of biology, computational physics, spatial cognition, etc. For such datasets even a slightest decrease in computational complexity could mean a significant practical gain in terms of time taken to solve the problem. More importantly, achieving spatial scalability is a primary focus because, these large datasets can be fit into memory for data analysis. Problems in computational geometry such as Closest Pair of Points, Voronoi diagram, Delaunay Triangulation, Minimum Spanning Tree and Convex Hull are useful in measuring data distances and categorizing data in biology, spatial cognition and even geo/multimedia/environmental databases. Hence, parallelization of algorithms to solve these problems is important in order to achieve a speed up with respect to computation time as well as to build scalable solutions for larger datasets. Furthermore, the end users in these fields are from a non-computing background which gives us a motivation to build intuitive parallelized algorithms.

MASS (Multi-Agent Spatial Simulation) is a parallel computing library based on mobile agent-based simulation of applications in physical, biological, social, and strategic domains in a given virtual space. The library was developed by Dr. Munehiro Fukuda and his research group at the Distributed Systems Laboratory (DSL), University of Washington, Bothell. The MASS library provides multiple-language support which include C++, Java, and CUDA. MASS is an Agent-based modeling (ABM) library, which supports efficient analysis of structured datasets and hence will be better suited to certain computational geometry algorithms as opposed to other parallel programming frameworks such as Spark, and MapReduce which follow the parallel data-streaming approach.

The Agent based parallelization of computational geometry algorithms project is aimed at adding parallel implementations of certain widely used algorithms in computational geometry to the MASS Java library. As part of my CSS600 Independent Study course and CSS700 Master’s thesis, select computational geometry algorithms were implemented in MASS, Spark and MapReduce. The following sections discuss the implementations and results I have so far, in detail.

2. Closest Pair of Points Implementation

2.1 Closest Pair of Points Algorithm

The implementation of the Closest Pair of Points (CPP) problem is based on a divide and conquer algorithm. A high-level description of the algorithm in each of the three parallel programming models is outlined below.

2.1.1 Spark & MapReduce Algorithm

Algorithm 1: Closest Pair of Points (Spark & MapReduce)
Closest-Pair (input_file)

- Read the input points from input_file
- Sort the points in increasing order of x co-ordinate
- <Stripe, [Points]> <- Map 3 points at a time to stripes

For each <Stripe, [Points]> do:
- Compute pair-wise distances between the points and record the local closest pair of points
- Delta <- minimum distance
- Point1, Point2 <- closest pair of points within the current stripe
- L <- point with the largest x-coordinate value within the stripe

EndFor

While (number of stripes != 1) do:
- L <- point with the largest x-coordinate value in stripe 1
- Delta1 <- minimum distance between closest pair of points in stripe 1
- Delta2 <- minimum distance between closest pair of points in stripe 2
- Delta <- Min(Delta1, Delta2)
- Point1, Point2 <- closest pair of points with distance Delta
- Sy <- points in stripe 2 within distance of Delta from L
- Construct Sy (O(n) time)
- For each point s ∈ Sy, compute distance from s to each of next 7 points in Sy
- Let s, s′ be pair achieving minimum of these distances (O(n) time)

If (d(s, s′) < Delta) then:
   closest pair of points of the combined stripe are s, s′
Else:
   closest pair of points of the combined stripe are Point1, Point2
EndIf

EndWhile

2.1.2 MASS Algorithm

Algorithm 1: Closest Pair of Points (MASS)

Closest-Pair (input_file)

- Read input points from input_file
- Rows, Cols <- number of rows and columns of Places matrix required to house all the points. (max value of x and y values)

- Create Places matrix of order Rows * Cols
- Map points to appropriate indices in the Places matrix
- Create Agents at locations in the Places matrix only where points are mapped
- ClosestPairFound <- False

**While (!ClosestPairFound) do:**
- Spawn children in neighboring cells of each agent where the same point is not mapped already and is within the boundary of Places matrix
- Map each spawned agent with the point that was assigned to the Place where it originated from
- Collect agents in each Place and check for agent collision

**If |agents| > 1 in any Place do:**
- The points mapped to colliding agents constitute the closest pair of points
- ClosestPairFound <- True

EndIf

EndWhile

---

### 2.2 Results

#### 2.2.1 Programmability

Programmability of the solutions using the three different parallel programming models is measured by computing the following metrics:
- Boiler plate code – number of lines of code required to setup the parallel programming environment
- LOC – total number of lines of code
- Number of instances – number of class instances created or number of RDDs (Resilient Distributed Dataset) for Spark that are created during the entire duration of the program.

Table below captures these metrics for MASS, Spark and MapReduce.

<table>
<thead>
<tr>
<th>Parallel Framework</th>
<th>Boiler plate code</th>
<th>LOC</th>
<th># of instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASS</td>
<td>6 (1.08%)</td>
<td>551</td>
<td>5</td>
</tr>
<tr>
<td>Spark</td>
<td>3 (1.22%)</td>
<td>245</td>
<td>8</td>
</tr>
<tr>
<td>MapReduce</td>
<td>35 (5%)</td>
<td>628</td>
<td>9</td>
</tr>
</tbody>
</table>

*Table 1. Programmability metrics for CPP implementation*

We can observe from the table that MASS and Spark implementations require very few lines of boiler plate code compared to MapReduce implementation. The lines of boiler plate code required for MapReduce implementation comes mainly from having to setup the job configuration, output key and value class, mapper and reducer class, input and output format, input and output paths etc. for each of the map reduce jobs. The current implementation for the CPP algorithm using MapReduce consists of three different mappers and reducers applied in order and thus increasing the amount of boiler plate code. It appears that both Spark and MASS are better programming models for this problem in terms of amount of boiler plate code required compared to MapReduce.
The total number of lines of code required for the CPP algorithm is clearly very high in the MapReduce implementation compared to MASS and Spark. The reason being the requirement for multiple mappers and reducers to be implemented for the CPP. Due to this reason, code for intermediate file reads and formatting the input to objects in the program adds to the lines of code. While LOC for the MASS implementation is higher than Spark implementation, it is still less compared to MapReduce. It is evident that Spark is the better programming model for this problem in terms of lines of code required for the implementation.

The number of class instances required by MASS implementation is fewer compared to that in Spark and MapReduce. This reduces the need to store multiple objects in memory during execution of the program thereby occupying less memory.

2.2.2 Execution Performance

**Spark.** Screenshot below shows the execution performance of the Spark implementation with an 8-node configuration. An input size of 8192 was used for this execution.

![Screenshot of execution time taken by Spark CPP implementation with an 8-node configuration.](image1)

**Figure 1.** Screenshot of execution time taken by Spark CPP implementation with an 8-node configuration.

**MapReduce.** Screenshot below shows the execution performance of the MapReduce implementation with 1 slave node. This implementation was tested only on one slave node configuration due to issues in Hadoop MapReduce configuration with multiple slave nodes. The input consisted of 10,000 randomly generated 2D points with co-ordinates in the range 1-100.

![Screenshot of execution time taken by MapReduce implementation of CPP with a single slave node configuration.](image2)

**Figure 2.** Screenshot of execution time taken by MapReduce implementation of CPP with a single slave node configuration.
**MASS.** Screenshot below shows the execution performance of the MASS implementation with an 8-node configuration. An input size of $8192 = 2^{13}$ was used for this execution.

![Screenshot of execution time taken by MASS CPP implementation with an 8 node configuration.](image)

**Performance Comparison.** Due to setup issues with MapReduce, the performance comparison graph shown in Figure 4 includes only the results from the executions of Spark and MASS implementations. I will be measuring the performance of MapReduce implementation and update the results by the start of Winter quarter, once I resolve the setup issues for multi-node execution. As seen from the graph in Figure 4, with fewer nodes on the cluster the MASS implementation of CPP takes more time for the input compared to the Spark implementation. But we can see that the trend decreases with the increase in number of nodes for MASS. This input size was chosen because, with the availability of only 8 nodes on the current parallel programming cluster for MASS, increase in the input size takes longer to execute. We can infer from the trend that increasing the number of nodes on the cluster would have a positive impact on the performance of the MASS implementation.
3. Convex Hull Implementation

3.1 Convex Hull Algorithm

The implementation of the Convex Hull problem is based on a divide and conquer algorithm. In this quarter I was able to complete implementation for convex hull in Spark. I will work on MASS and MapReduce implementations in the Winter quarter. A high-level description of the Spark algorithm is outlined below.

3.1.1 Spark Algorithm

**Algorithm 1: Convex Hull**

ConvexHull (input_file)

- Read input points from input_file
- Sort the points in increasing order of x co-ordinate
- <Stripe, [Points]> <- Map 2 points at a time to each stripe

For each <Stripe, [Points]> do:
- Compute Convex Hull for individual stripes
- Convex_hull <- all points in current stripe
  For each <Point> in convex_hull do:
  - Point.cw_next <- clockwise neighbor of the point in the convex hull
  - Point.ccw_next <- counter-clockwise neighbor of the point in the convex hull
  EndFor
EndFor

Figure 4. Spark and MASS execution time with increase in number of worker nodes for an input size of $4096 = 2^{12}$.
While (number of stripes != 1) do:
  - upperTangent, lowerTangent <- MergeHulls(ch1, ch2) // merge hulls of two consecutive stripes
  - point <- upperTangent.p1
    While (point.cw_next != p1) do:
      - Unified_Convex_hull += point.cw_next
      - Point <- point.cw_next
    EndWhile
EndWhile

MergeHulls(ch1, ch2)

  - p1 <- right most point on ch1
  - p2 <- left most point on ch2
  - upperTangent <- line(p1, p2)

While (upperTangent crosses ch1 or ch2) do:
  While (upperTangent crosses ch1) do:
    p1 <- p1.ccw_next
  EndWhile
  While (upperTangent crosses ch2) do:
    p2 <- p2.cw_next
  EndWhile
  - upperTangent <- line(p1, p2)
  - p1.cw_next <- p2
  - p2.ccw_next <- p1
EndWhile

  - p3 <- right most point on ch1
  - p4 <- left most point on ch2
  - lowerTangent <- line(p3, p4)

While (lowerTangent crosses ch1 or ch2) do:
  While (lowerTangent crosses ch1) do:
    p3 <- p3.cw_next
  EndWhile
  While (upperTangent crosses ch1) do:
    p4 <- p2.ccw_next
  EndWhile
  - lowerTangent <- line(p3, p4)
  - p3.ccw_next <- p4
  - p4.cw_next <- p3
11

EndWhile

Return upperTangent, lowerTangent

3.2 Results

Since I was able to implement Convex Hull algorithm only using Spark this quarter, I am including the output screenshots of the execution of this algorithm. Performance metrics will be included in the report once I complete MapReduce and MASS implementations of Convex Hull in the Winter quarter.

*Spark.* Figure 5 shows the output for Spark Convex Hull implementation in a single node configuration with 8 and 1000 input points.

![Output Convex Hulls for 8 and 1000 input points for Spark implementation in a single node configuration.](image)

4. Voronoi Diagram Implementation

4.1 Voronoi Diagram Algorithm

The implementation of the Voronoi problem is based on a divide and conquer algorithm in Spark. In this quarter I was able to complete implementation for Voronoi diagram in Spark and MASS. A high-level description of the Spark and MASS algorithm is outlined below.

4.1.1 Spark Algorithm

**Algorithm 1: Voronoi Diagram**

VoronoiDiagram (input_file)

- Read input points from input_file
- Sort the points in increasing order of x co-ordinate
- `<Stripe, [Points]> <- Map 2 points at a time to each stripe`

**For each <Stripe, [Points]> do:**
- Compute Convex Hull for individual stripes
- Compute Voronoi Diagram for individual stripes
- `Convex_hull <- all points in current stripe`
- `Voronoi_diagram <- <Point, [Voronoi_edge]> // edge representing the two half-planes formed by two points in each stripe`

**EndFor**

**While (number of stripes != 1) do:**
- `upperTangent, lowerTangent <- MergeHulls(ch1, ch2) // merge hulls of two consecutive stripes`
- `Unified_Voronoi_Diagram <- MergeVoronoiDiagrams(vd1, vd2) // merge Voronoi diagrams of two consecutive stripes`

**EndWhile**

**MergeVoronoiDiagrams(vd1, vd2)**
- `sL <- upperTangent.p1`
- `sR <- upperTangent.p2`
- `e <- the perpendicular bisector of sites sL belongs to stripe1 and sR belongs to stripe2`
- `eL <- first edge on open boundary of Voronoi_diagram(sL)`
- `eR <- first edge on open boundary of Voronoi_diagram(sR)`

**While (sL != lowerTangent.p1 and sR != lowerTangent.p2) do:**
  **While (edges e and eL do not intersect) do:**
  - `eL <- eL.NEXT`
  **EndWhile**

  **While (edges e and eR do not intersect) do:**
  - `eR <- eR.NEXT`
  **EndWhile**

  **If (IntersectionPoint(e,eL) closer to e.p1 than IntersectionPoint(e,eR)) then:**
  - `sL <- point of input data set that is on the other side of eL`
  - `e <- bisector of sL and sR`
  - `eL <- reverse of eL (the new eL is an edge of Voronoi_diagram(sL))`
  **Else do:**
  - `sR <- point of input data set that is on the other side of eR`
  - `e <- bisector of sL and sR`
  - `eR <- reverse of eR (the new eR is an edge of Voronoi_diagram(sR))`
  **EndIf**

**EndWhile**
4.1.2 MASS Algorithm

Algorithm 1: Voronoi Diagram

VoronoiDiagram (input_file)

- Read input points from input_file
- places_rows <- max co-ordinate in input in x direction * 10
- places_cols <- max co-ordinate in input in y direction * 10
- Create places of dimension places_rows * places_cols
- Map input points to their appropriate place location by multiplying each co-ordinate by the mesh interval default, which is 10
- Create Agents at places with points mapped to them

step <- 0
While number of agents > 0 do:
  If step % 2 equals 0 then:
    Spawn 4 children of each agent in a place and migrate them to Von Neumann neighborhood
  Else:
    Spawn 8 children of each agent in a place and migrate them to Moore neighborhood
  Kill parents of each of the newly spawned agents, since they won’t be part of a ripple collision yet
  Update agent_footprints list at each Place with the point mapped to all new agents that migrated to this Place
  Check for ripple collision at each Place
    1. If two agents meet at a Place which maps to the co-ordinates of the mid-point of the two points mapped to these agents, migrate the agents to their source Place and add a new Voronoi edge
    2. If agents encountered other agents or agent footprints which are mapped to points other than the one mapped to the current agent, check if the perpendicular bisectors of each pair of points intersect. If they do and if the intersection point maps to the current place, this is a Voronoi vertex. Migrate agents to their source locations and update the edges with their new origins.
  3. Terminate the agent once the source hashtable is updated.
EndWhile

Collect all hashtables from Places that have a input point mapped, which form the final Voronoi diagram

4.2 Results

Since I was able to implement Voronoi algorithm only using Spark and MASS this quarter, I am including the output screenshots of the execution of these algorithms. Performance metrics will be included in the report once I complete MapReduce implementation of Voronoi Diagram
during the first 3 weeks of Winter quarter. Also, the input size considered for the execution is currently 8 points. Beyond this, my implementations have few bugs which I am currently trying to fix. I am hoping to complete debugging before the start of Winter quarter.

**Spark.** Figure 6 shows the output for Spark Voronoi Diagram implementation in a single node configuration with 8 input points.

![Spark Voronoi Diagram](image)

*Figure 6. Output for Spark Voronoi Diagram implementation with single node configuration.*

**MASS.** Figure 7 shows the output for MASS Voronoi Diagram implementation in a single node configuration with 8 input points. The MASS implementation is not able to recognize the right most Voronoi vertex in the output below because, the Places matrix is bounded to the maximum x and y coordinates in the input data set. Since the Voronoi vertex lies outside this bound, the agents cannot migrate to this location in the implementation and hence the vertex is not recognized. I will look for a possible and efficient solution for this edge case and update the implementation.

![MASS Voronoi Diagram](image)

*Figure 7. Output for MASS Voronoi Diagram implementation with single node configuration.*
5. Future Improvements

5.1 Closest Pair of Points

*Spark.* Grouping the input data points into stripes of three points each forms the crux of the divide and conquer implementation of CPP in Spark. This step requires a sequential pass through all the input points that have been sorted in increasing order of x co-ordinate and hence cannot be done parallelly using the RDD data structure provided by Spark because a global state needs to maintained among different partitions in order to get the final sort order, which is not straightforward. Thus, requiring the entire data to be brought into memory on the master node which proves to be a bottleneck for the execution performance. To overcome this bottleneck, the input data points can be sorted and grouped into stripes of three and written to the input file before starting the Spark implementation execution.

*MapReduce.* Similar problem as mentioned above exists with the MapReduce implementation as well. The solution mentioned above can also be adopted for this implementation to reduce the bottleneck for the execution performance. One more problem with the MapReduce implementation is that multiple mappers and reducers are required to implement the CPP algorithm which means that multiple file reads and writes need to be performed throughout the program thereby increasing the execution time of the program. One way to mitigate this would be to use the above solution itself by sorting and grouping the points into stripes of three which will reduce an additional file write and read and can reduce the execution time of the program.

*MASS.* Due to the creation of multiple *Place* objects in the current MASS implementation performance will be hindered because of object serialization over network. Also, the places matrix will be sparse because not all matrix locations will be occupied by the input points. To overcome this, the implementation can be tweaked to create a dense matrix of places with each place consisting of a hash table that records information about the interval between the current *Place* and its neighbors. The hash table can also contain information about the agents mapped to each place in order to detect collision and thereby compute the closest pair of points.

5.2 Voronoi Diagram

*MASS.* The current Spark implementation of Voronoi Diagram works only for input points that are integers and not real numbers. This is due to the fact that a fine-grained mesh needs to be created with an interval of at least 10 between two consecutive co-ordinates in both X and Y axes in order to be able to track the mid-point of Voronoi sites. With the current implementation, the dimension of Places matrix will be huge and with an availability of 8 nodes in the parallel programming cluster, the program cannot be tested with large input sizes. Hence, the input data is limited to only integers. In future, if more computing nodes are available, an even more fine-grained Places matrix can be created to house input points that have co-ordinates as real numbers.
6. Current Status

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MASS</th>
<th>Spark</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closest pair of Points</td>
<td>Completed, tested</td>
<td>Completed, tested</td>
<td>Completed, tested</td>
</tr>
<tr>
<td>Voronoi Diagram</td>
<td>Completed, testing</td>
<td>Completed, testing</td>
<td>Planned for Winter’20</td>
</tr>
<tr>
<td>Convex Hull</td>
<td>Planned for Winter’20</td>
<td>Completed, tested</td>
<td>Planned for Winter’20</td>
</tr>
<tr>
<td>Delaunay Triangulation</td>
<td>Planned for Spring’20</td>
<td>Planned for Spring’20</td>
<td>Planned for Spring’20</td>
</tr>
</tbody>
</table>

*Table 2. Current progress of algorithm implementations.*

Table 2 shows the current status of the algorithms implemented so far as part of my thesis. I am currently working on testing the MASS and Spark Voronoi implementations and hoping to have all the bugs fixed by start of Winter quarter. Once I complete that I will be left with 6 implementations which I will be working on in Winter and Spring quarters as shown in the table.

7. Project Package

All implementations have been checked into a CSS600_saranyag of the bitbucket mass_java_appl repository at:

https://bitbucket.org/mass_application_developers/mass_java_appl/branch/CSS600_saranyag

8. Conclusion

The research work done so far has helped me understand the different parallel programming frameworks: Spark, MapReduce and MASS and how to parallelize algorithms using them. From programmability perspective, Spark and MASS excel compared to MapReduce which requires boiler plate code that constitutes 5% of the entire implementation for the CPP problem. MASS implementation only requires 1.08% of boiler plate code to setup the parallel programming environment. MASS implementation also proves to be better by not requiring multiple file read and write operations. Once the data points are read from the file, the data is distributed among Place elements over the network and all operations are performed using method calls on places and agents. This provides significant performance gain in terms of execution time.

From my observation during the research work as part of my thesis, I saw that there is a steep learning curve for Spark and MapReduce compared to that of MASS. Also, Spark and MapReduce programming models require complex divide-and-conquer algorithms to parallelize algorithms which proved to be time consuming. On the other hand, the agent-based simulation approach provided by MASS library allows users to build more intuitive parallel algorithms and doesn’t require much time in terms of designing the algorithm and coding it up. However, the documentation for MASS Java library is sparse and could use more details on how to use certain
APIs provided by the library. Also, some of the design decisions like limiting the number of agents that can be spawn in a single node, could be included in the documentation to help developers be aware of them.

All of the three frameworks abstract the underpinnings of the network communication between nodes on the cluster and the splitting of data among the nodes which makes it easier for the programmer by letting them focus only on the program logic. Although Spark and MapReduce have their own advantages as parallel programming frameworks, the MASS Java library is better in terms of the learning curve associated with using the library to parallelize applications and also in terms of providing an intuitive way of programming which the other two frameworks lack on. With addition of more computing nodes to the parallel programming cluster, MASS could achieve similar performance as that of its counterparts – Spark and MapReduce.

9. References


