**Analysis and Improvement of Current MASS-based GIS**

**Shahruz Mannan**

Capstone Term Report Wi24 submitted   
in partial fulfilment of the

requirements of the degree of

Master of Science in Computer Science & Software Engineering

## University of Washington 03/16/2024

**Project Committee:**

Dr. Munehiro Fukuda, Committee Chair

Dr. Michael Stiber, Committee Member

Dr. Dong Si, Committee Member

1. **Project Overview**

As geospatial data or geodata becomes more complex, the need for performance improvements and data handling enhancements in the geospatial information systems (GIS) becomes increasingly important. This could be achieved for example with the use of parallel computing techniques. The distribution of computations across multiple processors or computers can significantly enhance the performance and increase the speed of data processing of geodata. Multi-Agent Spatial Simulation (MASS) [1] is a parallel-computing library which is designed to support large-scale simulation models, such as agent-based models and cellular automata. MASS provides a scalable platform for developing and running models that involve large numbers of autonomous agents acting in a spatial context. This library allows these models to be parallelized across multiple processors, which significantly speeds up their execution. Figure 1 illustrates a typical parallel execution process using the MASS library. In this GIS context, performance is usually gauged by the effectiveness of the system's CPU scalability and the amount of geodata it can support. Thus, MASS can be a viable option for these GIS improvements.

This project will delve into two main areas. The first focus area is the improvement of the performance of the MASS-based GIS by exploring its CPU scalability. Current Geospace + MASS-Java implementations which previous students have implemented will be compared with Message Passing Interface (MPI) implementations. Depending on the outcomes of these comparisons, the project will either target the enhancement of CPU scalability for GIS parallelization or pivot towards spatial scalability. This shift in focus would assess the system's ability to handle larger GIS space with increased CPUs, as opposed to purely looking at speed improvements.

Given that the current implementation of MASS-based GIS predominantly focuses on vector data, this project additionally proposes to incorporate and analyze raster data to MASS Places. The objective is to parallelize GIS attribute/spatial queries using MASS Places and Agents, thereby broadening the system's data handling capabilities. The selection of these queries will be guided by two main criteria: computational intensity and relevance to real-world applications. Operations that are computationally intensive and could significantly benefit from parallelization will be prioritized. In addition, operations commonly used in real-world GIS applications will be considered to ensure the practical relevance of the project. For selecting GIS attribute/spatial queries, this project will reference Raghavendra's (2023) white paper on Agent-based GIS Queries."

By integrating these improvements, the project aims to enhance the computational performance of MASS-based GIS, while widening its scope to handle raster data as well. Through these efforts, this project will provide a significant contribution to the advancement of geospatial data handling and processing capabilities.

A diagram of a system

Description automatically generated  
*Figure 1. Parallel execution using MASS [2]*

1. **Goals & Criteria**
   1. **Goals**

The primary goal of this project is to enhance the computational performance and data handling capabilities of the MASS-based GIS. The main objectives are shown below:

* Goal 1: Develop a foundational understanding of the current MASS-based GIS performance by studying the existing Geospace + MASS-Java implementation. This analysis will serve as a benchmark for subsequent performance improvements. It will include an evaluation of the current CPU scalability and vector data handling capabilities, thereby establishing a starting point for performance and data handling enhancements.
* Goal 2: Compare the current MASS-based GIS implementation with MPI implementations to understand their differences in CPU scalability. This comparison will identify potential paths for performance improvement in MASS-based GIS.
* Goal 3: Depending on the analysis from Goal 2, if the MPI implementation shows CPU scalability, I will work on improving the CPU scalability of MASS-based GIS. If the implementation does not show CPU scalability, I will either try to continue to work on improving the CPU scalability by tackling the obstacles or shift the focus to spatial scalability. If spatial scalability becomes the focus, I will assess the system's capacity to support larger GIS spaces with more CPUs, aiming to enhance the efficiency of processing larger volumes of geospatial data.
* Goal 4: Integrate raster data handling into MASS-based GIS, currently focusing on vector data. This will involve mapping raster data to MASS Places and parallelizing some GIS attribute/spatial queries with MASS Places and Agents. This goal will extend the capabilities of the system to process and analyze a broader range of geospatial data.
* Goal 5: Conduct a comprehensive evaluation of MASS-based GIS after implementing the above improvements. This evaluation will examine the execution performance and programmability analysis of the improved system.

1. **This quarter’s achievements**
   1. **MPI Comparison**

In the Autumn 2023 quarter’s term report, I worked on implementing the Euclidean Shortest Path computational geometry problem with MPI Java to assess their CPU scalability and how well these problems are parallelizable. However, this implementation was not finished because of a logic-breaking bug where the edges in the visibility graph would pass through the obstacles. This bug is affected by not outputting the correct shortest path.

* + 1. ***Euclidean Shortest Path MPI***

In this quarter I fixed the logic-breaking bug and completed the implementation. To fix this issue I had to modify the visibility graph creation algorithm. Instead of using the visibility graph with a rotational sweep with Lee's Visibility Graph Algorithm [3], I decided to implement the naïve version [4]. This naïve algorithm compares every pair of points in the set of obstacles, and checks if they intersect with any edges of obstacles. If a pair of points does not intersect with any edge, and the pair is external to the obstacles, they are visibility edges. Even though this naïve algorithm is O(N^3) which is worse than Lee's Visibility Graph Algorithm O(N^2logN), choosing the naïve algorithm was better for this scenario. The objective for implementing the computation geometry problems with MPI Java was to analyze how well these problems parallelize as more computing nodes are used. In addition, after this MPI implementation, the plan was to use the visibility graph algorithm in the MASS implementation. Constructing the visibility graph with a rotational sweep was more complex, which might have added difficulties with MASS. Figure 2. shows the correct implementation of the visibility graph.

A graph of a triangular figure

Description automatically generated

*Figure 2. Visibility Graph with three obstacles*

Besides this change, the implementation is similar to what I described in the previous term report. I partitioned the obstacle points, the starting point, and the destination point to the computing nodes and these nodes created the visibility graphs with their assigned points. This information was saved as a HashMap where the key was a vertex as a point and the value as a list of points of the vertices which can create an edge with this specific point in the visibility graph. Next, all the partial visibility graphs were sent back to the master rank to be combined and used to execute Dijkstra’s algorithm [10] to find the shortest path. The benchmark results can be found in section 4.

* 1. **Parallelization of GIS Queries**

For this quarter, I decided to make changes to the workflow of the original project plan. This is because I realized it did not make sense to focus on some of the milestones before starting to work on the other milestones. In addition, I realized in the previous quarter that this project plan does not suit my workstyle and it would be more beneficial for me to shift the milestones based on how effective I am. Especially, the “Integration of Raster Data” milestone did not make sense to me to start working on before I had the Computational Geography problems implemented with MASS optimized and bug-free. Additionally, I prefer to focus on one problem at a time. Thus, I decided to tackle parallelizing each GIS query one at a time, evaluate the initial performance, and apply performance enhancements and debug before moving to another Computational Geography problem. A big proportion of the efforts in this quarter were dedicated to parallelizing these GIS queries.

* + 1. ***Range Search MASS***

Similarly to the Range Search MPI implementation. The MASS implementation is used to identify all points within a specified rectangular boundary from a set of points. The previous MASS implementation of Range Search used a K-d Tree data structure to organize the points and search the tree for the points that are in the range. The MPI benchmark results from the previous term report showed that the K-d Tree construction time parallelizes as more computing nodes are used. This signified that it might also be plausible with the Range Search MASS implementation.

This implementation starts by reading the data points from a shapefile (.shp). Shapefiles offer a nontopological format to store the geometric location of geographic features. In addition to points, shapefiles can store other complex geometries such as lines and polygons. Another major factor for choosing shapefiles is that shapefiles can be easily visualized on maps, which can make it possible to show the points in range on a map. The next step was partitioning the data into subsets corresponding to the number of MASS Places. This ensures that each subset is of a manageable size, thereby parallelizing the workload. Now MASS gets initialized, and the Places and Agents are created. Each subset gets assigned to the appropriate Place and these Places construct a K-d Tree. For this MASS implementation, the same K-d Tree construction algorithm was used as in the MPI implementation. Once the K-d Trees are constructed, the Agents are responsible for performing the range search on the K-d Trees. Each agent is associated with a Place and collects the points in range from that specific K-d Tree. Lastly, the agents return the points in range to the master node to be written to the GIS database and to be generated on a global map as coordinates. There is a possibility that an Agent does not find a single point which falls under the specified range. In this scenario, the Agent gets terminated which leads to less agents returning the points in range and making the execution more efficient.

GIS features such as reading a shapefile and creating a map, need an external library. In this implementation I have used The Open-Source Java GIS Toolkit: GeoTools [5]. I decided to use this because this library was used in previous GIS queries, and it was already installed. Currently, I am currently facing some issues with generating the map with this library as some of the classes from the library cannot be found. Most likely the path for these classes has been updated in the library and further investigation is needed.

* + 1. ***Convex Hull MASS***

For this implementation, I first tried to approach this implementation as I did with the Convex Hull MPI problem. I tried to distribute input points equally to Places and create a partial hull for each subset. Just like with the MPI implementation I tried to merge the partial hulls with a divide and conquer strategy. However, trying to merge correct partial hulls and not calling the Places which already have shared their partial hulls became an issue. This approach with MASS was not efficient. However, Professor Fukuda another approach to calculate the Convex Hull with MASS.

This was the elastic band algorithm [7]. I used Potturi (2023) Convex Hull MASS implementation as a reference for my implementation.

In this approach, the data points are read from a shapefile as in the Range Search MASS implementation. At the same time, the minimum and maximum values for both latitude and longitude are calculated to define bounding box to partition the data across the MASS Places. MASS initializes a two-dimensional grid of Places where each Place represents a portion of the overall space and contains a subset of the data points based on their spatial location. A modification I made which was specifically important for my implementation was how each Place receives the points. In the previous implementation, each Place read the input file and collected the points which belonged to the specific Place’s space. Considering spatial scalability, using a large number of points may slow the performance drastically as each Place tries to read from the same file which can lead to overhead. I read the points only once and pass the points to the Places when these are initialized. Additionally, I am planning to check whether it is possible to distribute the points to their correct subsets before initializing the Places eliminating the need each Place to iterate all the points.

Next, The Agents are initialized and assigned starting positions at the boundary of the grid. These agents traverse through the grid, and the strategy is to move inwards to identify potential points which might be part of the outer hull. If a Place contains potential outer hull points, these points are marked as visited and collected, and the current agent is terminated. This process in a simplified manner is shown in Figure 3 below. From the reduced set of data points the final Convex Hull is calculated with either Andrew's monotone chain algorithm [6] or Graham's scan algorithm [9]. These algorithms are the same algorithms I used in the Convex Hull MPI implementation. I decided to use both algorithms to investigate whether either of these algorithms behave differently between MASS and MPI.

Just like in the Range Search MASS implementation, the final Convex Hull points are written to the GIS database. In addition, I am planning to generate a map showing the Convex Hull points and connecting the points to visualize the actual hull which contains the rest of the points.

One caveat with this implementation is that as more data points are it is better to use more Places to reduce more of the points which are not potential outer hull points. This means more agents should also be used, otherwise the agents might miss a Place which contains potential outer hull points. This results in an incorrect Convex Hull due to missing outer hull points. However, using more agents means more time is spent traversing through the grid.

A diagram of a crime scene

Description automatically generated

*Figure 3. Agent moving inwards to find outer hull points [11]*

* + 1. ***Euclidean Shortest Path MASS***

The last MASS implementation I worked on this quarter was the Euclidean Shortest Path problem. The application aims to find the shortest path between two points in a 2D grid while avoiding obstacles. For this Computational Geometry problem, Potturi (2023) has already implemented an efficient MASS version which I am using as a reference for my implementation.

This implementation reads obstacle data from a file, where each line represents an obstacle defined by a set of coordinates of the obstacle corners. I have added the start and destination points for the pathfinding are specified as input arguments. The previous implementation only accepted obstacle vertices as starting and destination points. This scenario might not necessarily occur in a real-world scenario. The obstacles are stored as “Obstacle” objects which determine the blocked cells which reside inside the obstacles. This information is useful later when the agents move through the 2D grid to find the shortest path between the start and destination point. MASS initializes a two-dimensional grid of Places where each Place represents a cell of the overall space. Each cell can be a blocked cell if it is inside of an obstacle, a vertex cell if it is a corner of an obstacle, or a free cell.

This implementation starts with just one Agent in the starting cell. The agents use a form of depth-first search (DFS) to explore the grid until the destination cell is found. They consider moving to their neighbor cells and propagate through the grid by spawning new agents to these new cells. Each agent checks what type of cell is the current cell and acts upon that information. To avoid redundancy, each cell maintains a visited status and a current shortest path from the start cell. Agents will be terminated if they have not found a shorter path than the current shortest path. This pathfinding process also considers obstacles and terminates an agent if it is moving into a blocked cell. Once the destination cell is reached the path and the shortest path is reported back to master node. Figure 4 shows the agent propagation through Places in a 2D grid. In this figure the Places are colored as blue, the arrows indicate the next spawn locations for new agents, the red polygons describe obstacles and the points marked with a letter are the vertices.

A graph of a diagram

Description automatically generated with medium confidence

*Figure 4. Agent propagation through Places in a two-dimensional grid*

While testing the implementation, I found a couple bugs which impact finding the correct shortest path. In some scenarios, all the agents get terminated before the destination cell is found and, in some cases, the shortest path is not calculated correctly. I was able to fix the agents terminating early. However, I am still investigating the incorrect shortest path. As the implementation is still not working correctly, I have not moved the implementation to the GIS environment yet. Once I move the implementation to the GIS environment, I can read the obstacles from a shapefile and just like with the previous MASS implementations create a map which shows the obstacles and the shortest path taken. I have not benchmarked this implementation yet as I have been focusing on fixing the bugs. Once this implementation works, I will do benchmarks and compare the results with the Euclidean Shortest Path MPI implementation.

1. **Results**

The Range Search benchmark was conducted with my own created dataset of 1 million randomized (latitude, longitude) coordinates. Compared to the previous term report’s benchmarks I have excluded the read and write times from the total times and the range query time because these times were not relevant for benchmarking the CPU scalability of the range search algorithm and the range query time on average was under 10ms which is relatively small compared to the total time and the K-d tree construction time.

Figure 5. shows 4 different execution times with different numbers of computing nodes. The times are as follows from left to right: average of all K-d Tree construction time with MPI, average total algorithm time with MPI, average of all K-d Tree construction time with MASS and total algorithm time with MASS. The figure shows that the Range Search problem is CPU scalable, and the use of MASS parallelizes the Range Search algorithm. For the MASS implementation the execution times start stabilizing after 10 computing nodes. The MPI implementation generally outperforms the MASS implementation in terms of total time taken for both Kd-Tree construction and the total algorithm. This could be because of the lower overhead in MPI for the types of MPI communication functions used in this benchmark. In the MASS implementation, the agents need to return the points in range back to the master node. This is a slow operation in MASS and the difference between the MASS times is shown in the figure by getting the difference between the MASS times. In addition, compared to the current MASS-GIS system’s benchmarks for Range Search, the K-d Tree constructions take much less time even though using more data points. This shows that the current Range Search MASS-GIS implementation has been improved in both CPU and Spatial scalability aspects.

*Figure 5. Benchmark results for Range Search MASS & MPI*

The Convex Hull problem was benchmarked with 1 million randomized points .txt file. Similarly in the previous benchmarks I have removed read and write times from the total times like in the previous Range Search benchmarks. Figure 6. shows 4 different execution times with different numbers of computing nodes. The total time MPI (Andrew’s/Graham) describes total algorithm time for MPI with Andrew’s Monotone Chain Algorithm and Graham Scan algorithm. The total time MASS (Andrew’s/Graham) corresponds similarly to the MASS implementation with both Convex Hull construction algorithms. Figure 6. shows that with the MPI implementations parallelize slightly with the increase in node count for both algorithms, with Andrew Monotone Chain algorithm consistently outperforming Graham scan. This behavior we already saw in the previous term report. The worst times for the MPI benchmarks were (Andrew’s) 1954 ms with 1 node and (Graham) 3317 ms with 1 node. The best times the MPI implementations received were (Andrew’s) 1218 ms with 10 nodes and (Graham) 1257 ms with 13 nodes.

Unlike with MPI implementations, the MASS implementations don’t show CPU scalability as the execution time increases with more nodes. I think one of the reasons is that in MASS, both agents and places come with overhead communication. The implementation requires frequent communication between agents and places and the communication overhead could impact performance, especially as the number of nodes increases. As for the high execution times, I think this comes from each Place iterating through the data points to collect the points which fall under the Place’s spatial portion. By solving this issue, the execution times might get faster. Potturi (2023) benchmarked his Convex Hull implementation with 100k points. Even though his benchmarks show CPU scalability with 4-8 computing nodes, The execution times are high compared to my execution times with 1 million points. This Convex Hull MASS implementation needs further investigating. As for future benchmarks, I decided to use Andrew’s Monotone Chain Algorithm. Both are similar in execution times and in complexity O(NlogN) but Andrew’s Monotone Chain algorithm has fewer hidden costs such as fewer comparisons between points because two separate loops for upper and lower hull.

*Figure* 6*. Benchmark results for Convex Hull MASS & MPI*

The Euclidean Shortest Path was benchmarked with 300 obstacles where each polygon had three to six vertices. I implemented a method to create these obstacles in a way that the obstacles do not intersect each other. In this benchmark I did not exclude the read and write times because there are at most 1800 data points and the time reading these points compared to the whole execution time is small. The write time has the same reason and only a handful of points are written to a file. Figure 7. displays that the problem is highly scalable as more computing nodes are used. The performance improvement slows down after 11 nodes meaning the communication overhead in MPI starts being a major factor in the execution. For most of the execution time is spent creating the visibility graph because the algorithm is bound to O(N^3). The Dijkstra’s algorithm is bound to O(V + E log V) where V is the number of vertices and E is the number of edges. This results in Dijkstra’s algorithm being more efficient than the visibility graph algorithm and taking a small proportion of the execution time.

*Figure 7. Benchmark results for Euclidean Shortest Path MPI*

1. **Next quarters plan**

Before the next quarter starts, the highest priority is to fix the bug in the Euclidean Shortest Path MASS implementation as soon as possible. This makes it possible to generate a map with the correct shortest path. After that I will be able to run benchmarks for this implementation which I will share with the committee members separately, so they know if the Euclidean Shortest Path MASS implementation is CPU scalable or not. The next priority item is the Largest Empty Circle MASS implementation. In this quarter I ran out of time to start the Largest Empty Circle MASS implementation. One of the reasons was to figure out how to move my other MASS implementations to the GIS environment because I was not able to build the environment and execute either the previous GIS queries or my implemented GIS queries. It took quite a bit of time to figure out how to run the GIS queries.

For the next quarter, an important item is to have the map generation working on the implementations to visualize the outputs as this is a crucial part of GIS systems. Currently, most of the benchmarking has been conducted with datasets I have created. My aim for next quarter is also to find GIS datasets relevant to each GIS query to benchmark with actual real-world data. The last significant item in my agenda is to start experimenting with reading raster data after the MASS implementations are implemented. In addition to these tasks, a big part of next quarter is start drafting the final paper. I need to distribute my time efficiently between the next quarter’s tasks for me to complete all these tasks.

1. **Summary**

For this quarter, the focus was on implementing the Computational Geometry problems with MASS and comparing the performance against their MPI implementations. One of the quarter’s achievements was the successful implementation and benchmarking of the Euclidean Shortest Path using MPI, which demonstrated promising CPU scalability and set a baseline for Euclidean Shortest Path MASS.

The Range Search MASS implementation showed CPU scalability and outperformed the current existing Range Search implementation, which indicated an improvement in both CPU and spatial scalability. On the other hand, the Convex Hull MASS implementation showed challenges in achieving CPU scalability with MASS, and this suggests further optimization is needed.

While implementing the Euclidean Shortest Path problem with MASS, bugs were encountered which impacted the accuracy of the shortest path calculation. In the upcoming weeks, one of the priority items is to resolve these issues focusing on ensuring the shortest path is calculated correctly and integrating the implementation into the GIS environment for benchmarking against the Euclidean Shortest Path MPI implementation.

Plans for the upcoming quarter include finalizing the Euclidean Shortest Path MASS implementation, implementing the Largest Empty Circle MASS, and integrating raster data to the GIS queries. Other important tasks for the upcoming quarter include searching for relevant real-world GIS datasets for benchmarking and drafting the final paper.

While the current implementation is a little behind the aspirational plan, I am ready to act and try to finish most of the remaining tasks before the upcoming quarter starts. I am confident that I can catch up to the project plan and meet the goals.

1. **References**

[1] MASS: A parallelizing library for multi-agent spatial simulation. Retrieved July 16, 2023, from <http://depts.washington.edu/dslab/MASS/>

[2] Chuang, T; Fukuda, M. (December 2013). A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems. In Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering. Retrieved July 19, from <https://ieeexplore.ieee.org/abstract/document/6755210>

[3] Coleman, D. (2012). Lee’s O(n2 log n) Visibility Graph Algorithm: Implementation and Analysis. Retrieved December 13, from https://dav.ee/papers/Visibility\_Graph\_Algorithm.pdf.

[4] Nguyet, T. T. N.; Van Hoai, T.; Thi, N. A. (October 2011). Some advanced techniques in reducing time for path planning based on visibility graph. In 2011 Third International Conference on Knowledge and Systems Engineering (pp. 190-194). IEEE. Retrieved March 13, from https://www.researchgate.net/publication/221460582\_Some\_Advanced\_Techniques\_in\_Reducing\_Time\_for\_Path\_Planning\_Based\_on\_Visibility\_Graph

[5] GeoTools. GeoTools, the open source Java GIS toolkit. Retrieved from https://geotools.org/

[6] Alzubaidi, A. M. N. (2021). Minimum Bounding Containers of 2D Convex Polygon. Retrieved December 13, from

https://www.researchgate.net/publication/354380006\_Minimum\_Bounding\_Containers\_of\_2D\_Convex\_Polygon.

[7] Saadati, S., & Razzazi, M. (December 2022). Natural way of solving a convex hull problem. Retrieved March 13, from https://arxiv.org/ftp/arxiv/papers/2212/2212.11999.pdf

[8] Potturi, A. (June 2023) Programmability and Performance Analysis of Distributed and

Agent-Based Frameworks. Retrieved March 13, from https://depts.washington.edu/dslab/MASS/reports/AnirudhPotturi\_whitepaper.pdf

[9] Santoso, H. (2019). Mapping and Grouping of Farm Land with Graham Scan Algorithm on Convex Hull Method. Retrieved December 13, from <https://www.researchgate.net/publication/337508872_Mapping_and_Grouping_of_Farm_Land_with_Graham_Scan_Algorithm_on_Convex_Hull_Method>.

[10] Dijkstra, E. W., Beauguitte, L., Maisonobe, M. (1959). A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, p. 269271 Version bilingue et commentée. 2021. Retrieved December 13, from https://dl.acm.org/doi/abs/10.1145/3544585.3544600

[11] Fukuda, M.; Potturi, A.; Helzerman, J. (October 2023) Agent-Based versus Data-Streamed Big-Data Parallelization in Computational Geometry. Research Square. Retrieved December 13, from https://www.researchsquare.com/article/rs-3414sss742/v1

**Appendix**

**URL for code**

* <https://bitbucket.org/mass_application_developers/mass_java_appl/src/master/>
* Branch: shahruz/gis\_improvements

**How to run the codes (MASS)**

* Clone the MASS core library from mass\_java\_core bitbucket repository
* Navigate to the downloaded MASS core library folder and install the library: “mvn -DskipTests clean package install”.
* Clone mass\_java\_appl repository and git checkout to shahruz/gis\_improvements branch
* Build the MASS application with the appropriate path to the specific GIS query Applications/gis\_database/pom.xml
* The paths are commented out in the pom.xml, the specific GIS query path just needs to be uncommented.
* Run: mvn package
* Copy jar file to the MASS application folder from target folder: cp ./target/gis\_database-1.0-SNAPSHOT.jar .
* Edit the nodes.xml with appropriate masshome, username and port
* Execute the jar file:
  + Range Search:
    - java -jar gis\_database-1.0-SNAPSHOT.jar <number of Places> <minX><maxX><minY><maxY>
  + Convex Hull:
    - java -jar gis\_database-1.0-SNAPSHOT.jar <grid size>

**How to run the codes (MPI)**

* Have java installed.
* If MPI is not already set up, follow instructions from Computational Geometry MPI/range\_search\_mpi/mpi\_setup.txt
* Launch mpd CSSmpdboot -n <number of machines you want to connect to> -v
* To compile the codes:
  + cd <computational geometry problem you want to compile>/src
    - Locations:
      * Range Search: Applications/RangeSearch/MPI
      * Convex Hull: ComputationalGeometry/ConvexHull/MPI
      * Largest Empty Circle: Applications/LargestEmptyCircle/MPI
      * Euclidean Shortest Path: Applications/EuclideanShortestPath/MPI
  + javac <all the .java files in this folder>
  + e.g. javac KdTree.java Point.java rangeSearch.java
* To run the codes:
  + mpirun -np <number of machines you want to use> java <main class>
  + Range Search: rangeSearch <minX><maxX><minY><maxY>
    - e.g. rangeSearch "23.8647" "49.472737" "-127.663167" "-59.202464"
  + Convex Hull: convexHull
  + Largest Empty Circle: largestCircle
  + Euclidean Shortest Path: euclideanShortest
* Stop all mpds: mpdallexit