Josh Helzerman

CSS 497 Term Report Winter 2023

INTRODUCTION

This quarter, Winter 2023, I completed the final five credits of my capstone research project. I worked with Professor Munehiro Fukuda and a team of graduate students on benchmarking and implementing computational geometry applications. Our goal as a team was to compare agent-based implementations of computational geometry algorithms using the Multi-Agent Spatial Simulation (MASS) Java library with divide-and-conquer implementations of those same algorithms using stream-based distributed system libraries: Apache Spark and Hadoop MapReduce. We then benchmark these algorithms and measure their programmability and execution performance.

I was given two tasks this quarter. The first was to implement the Convex Hull algorithm on Apache Spark and Hadoop MapReduce. The second was to benchmark a Point Location algorithm implemented on MASS by a previous student. However, the original implementation turned out to be faulty, so this task also involved improving the implementation and adding a feature to MASS to improve performance.

For information on where to find the implementations and how to run them, go to appendix A.

CONVEX HULL

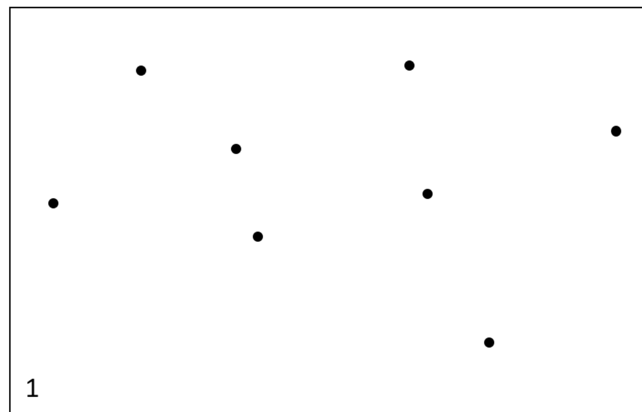
During the Fall quarter, I used a divide-and-conquer convex hull algorithm to construct convex hulls within my Voronoi diagram implementation. When we learned that the MASS implementation of Voronoi would not be completed in time and a MASS implementation of Convex Hull appeared more promising, we decided to focus our efforts on Convex Hull. I extracted my sequential Convex Hull algorithm from the Voronoi diagram program and used it to write two distributed versions of the algorithm using MapReduce and Spark.

The Algorithm

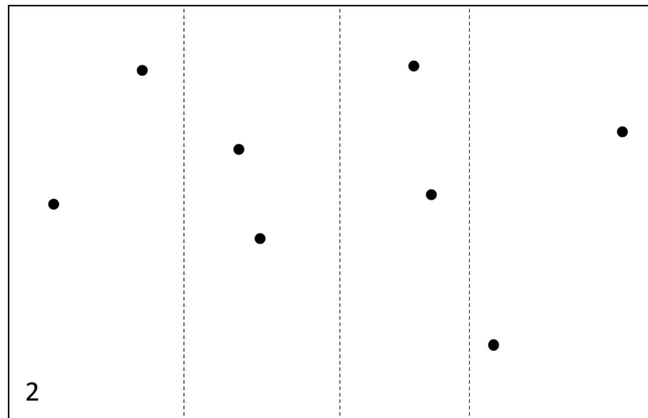
The Convex Hull sequential algorithm uses a divide-and-conquer strategy. Divide-and-conquer lends itself to parallel programs because we can split the problem up among computing nodes (divide), use the full algorithm on each node to solve a portion of the problem (conquer), and merge each node's solution quickly using our merge step of the algorithm. The merge step is linear in worst-case complexity. The full algorithm has a worst-case complexity of O(NlogN) where N is the number of points. Thus, the worst-case complexity of each node's share of the algorithm is O((N/K)log(N/K)) where K is the number of computing nodes. Since the nodes run in parallel and the merging between nodes is done in linear time, the distributed algorithm's complexity is equal to each individual node's complexity.

The algorithm's process is explained in the following steps:

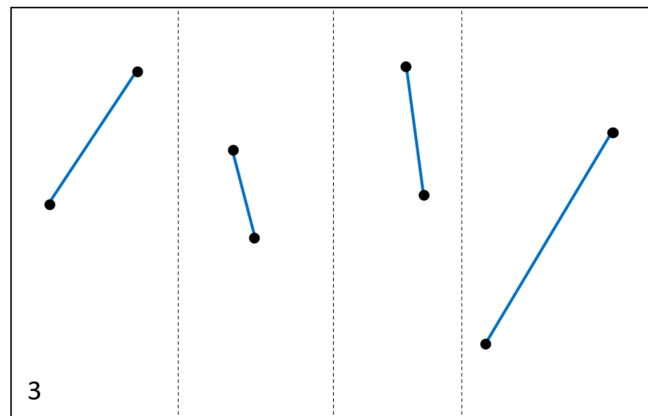
1. We receive a set of points. The points have already been sorted by their X-axis.



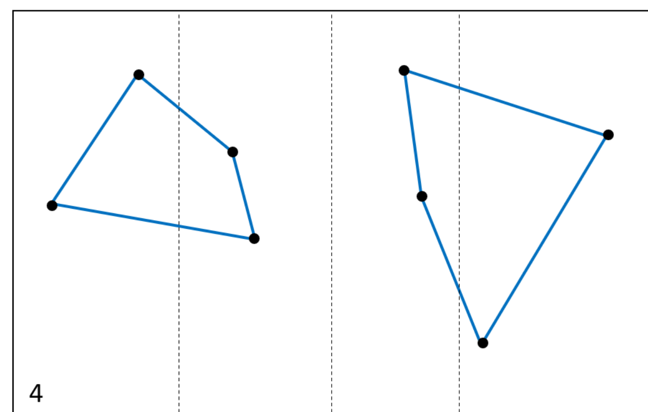
2. Divide the points into groups of one or two.

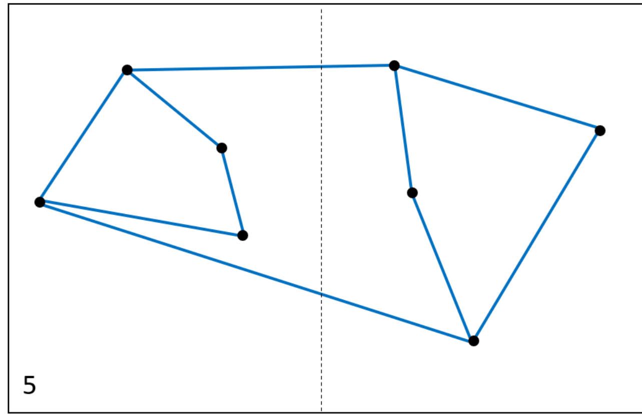


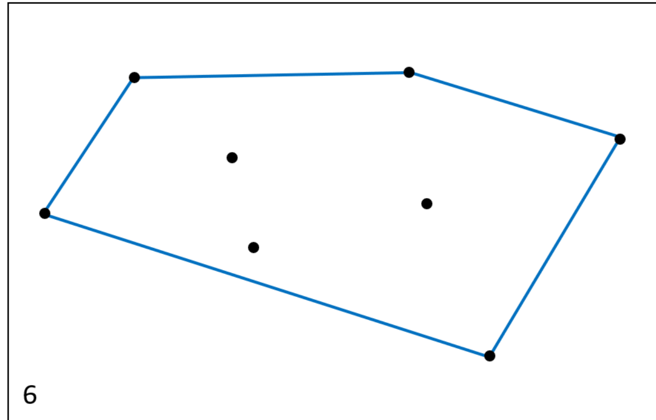
3. Create convex hulls for each group.



4. Merge each pair of neighboring convex hulls until one remains. Each merge is O(N) and there will be a log(N) number of merges.



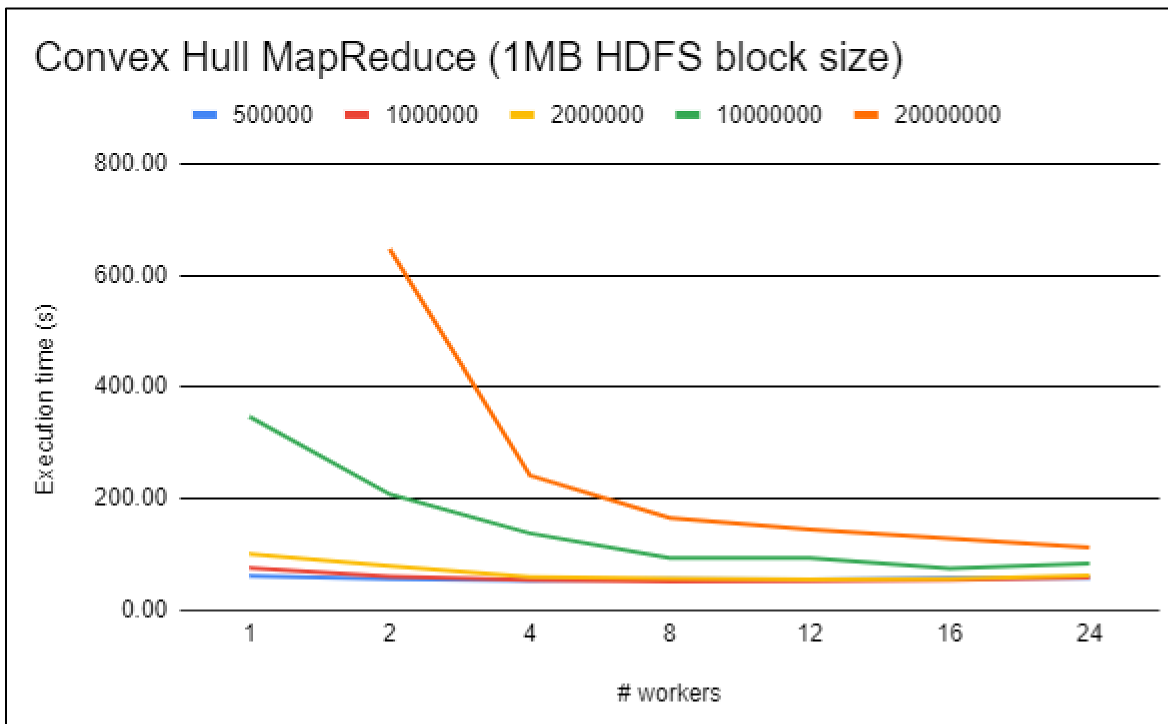




The merge step is a bit complicated. Detailed explanations of it can be found in the Computational Geometry lecture notes by Professor David M. Mount [1] and on the algorithm tutor website [2]. To explain it briefly, we find the topmost and bottommost tangents of each convex hull, and the points that each of these tangents intersect. Then, we delete all points on each convex hull that are between these two tangent points points.

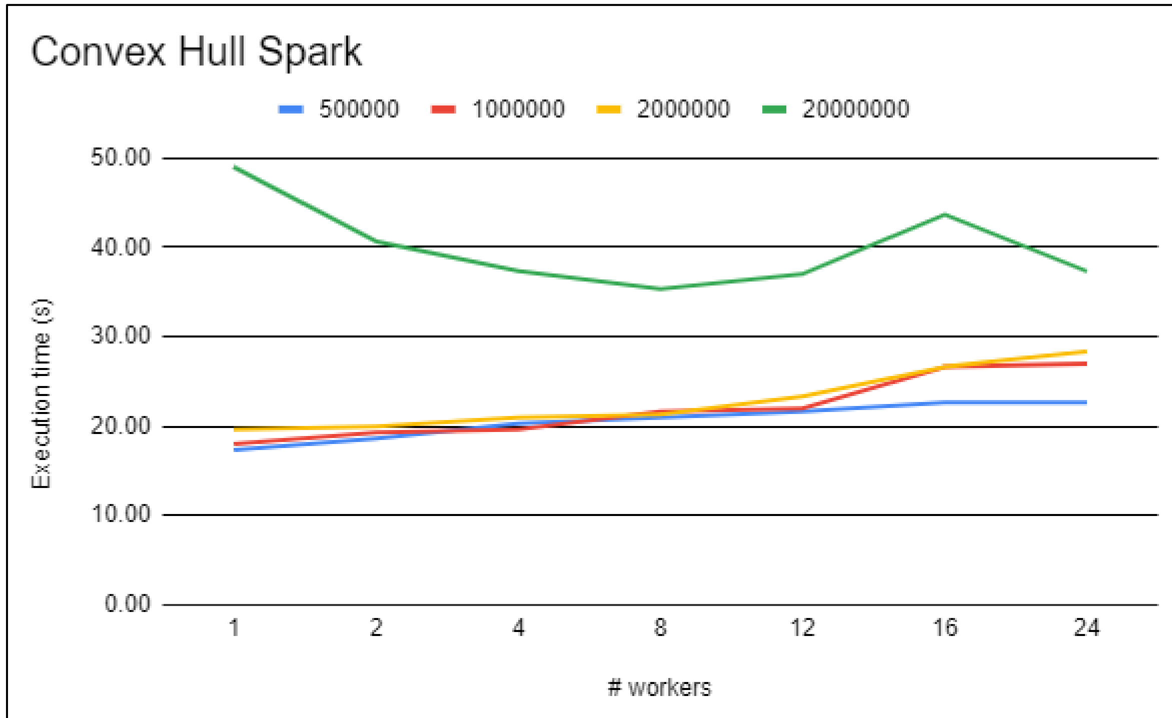
MapReduce

MapReduce uses files to distribute input and manage output (intermediate and final). These files are distributed amongst computing nodes using the Hadoop Distributed File System (HDFS). The default use-case for MapReduce is for analyzing or manipulating very, very large log files (terabytes in size). Our use case is very different from this. Our files are very small (6-29 MB), but for each line in the file we require much more computation than simple text search or manipulation. For this reason, to get performance that improves as we add nodes to the cluster, I had to configure HDFS for this use case. To do so, I changed the block size of HDFS from the default of 128 MB to 1 MB. HDFS distributes files over the cluster using blocks, so since our input files were smaller than the default block size, they were not getting distributed over the cluster. Each file would fit into one block and the block was replicated three times across the nodes. This led to no increase in performance as we increased the number of nodes. After changing the block size, we were able to see diminishing returns for each node added. This is the expected behavior of a parallel program.



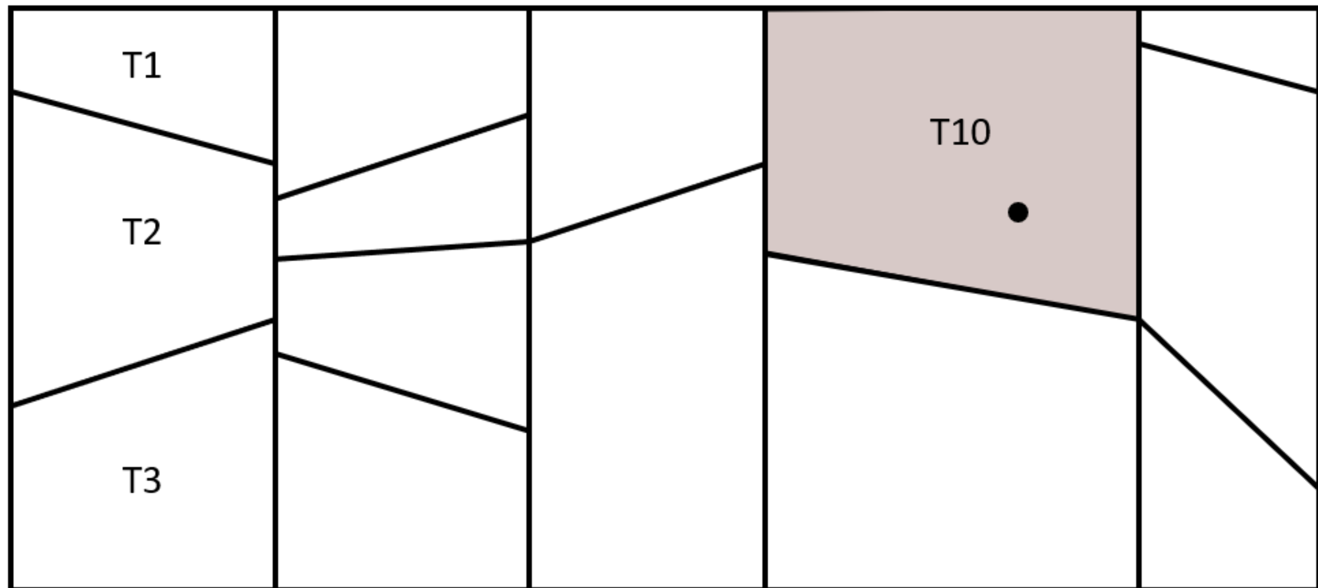
Spark

Implementing the algorithm on Spark was very simple and required no extra configuration tuning. It is identical to the MapReduce algorithm. Spark distributes the input and intermediate data to each node's memory as opposed to disk. I believe this is why it is so much faster than MapReduce for this case. Swapping could become an issue with larger inputs, but I never saw anything even at 20 million points.



POINT LOCATION

During the Fall quarter, I attempted to benchmark Point Location, but found that the algorithm and MASS had runtime issues that made the program incredibly slow. In Winter, I returned to Point Location to see if I could speed up its execution. I ran across several issues, not all of which were able to be solved during the quarter.

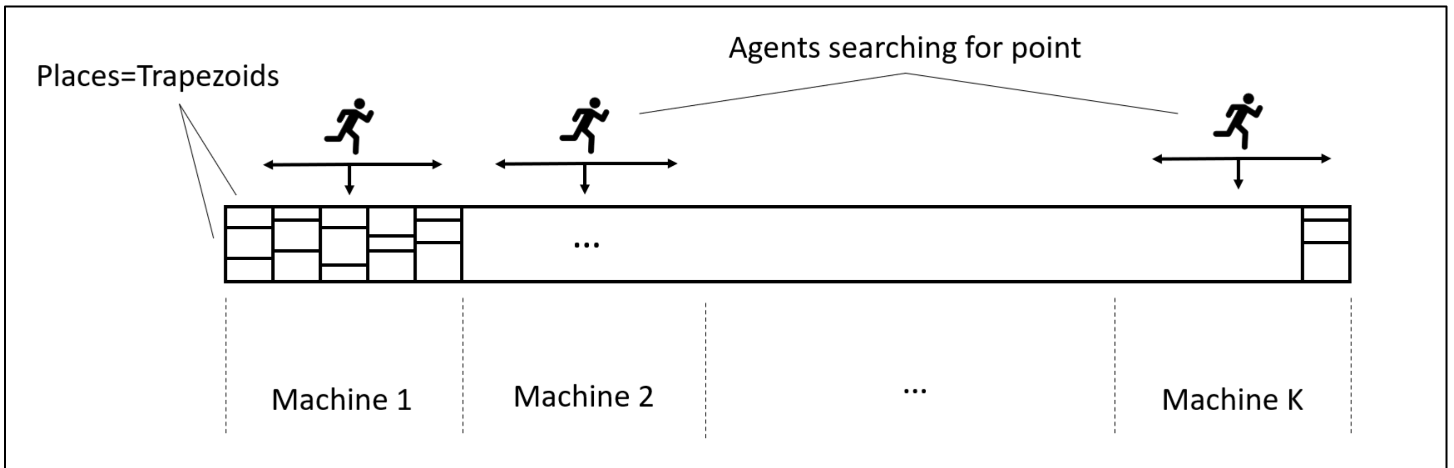


Getting the Program to End

When I originally ran the program, it never ended. The program ends when an agent finds the point. The agent broadcasts a message to all other agents saying the point is found. Upon receiving this message, all agents kill themselves. Unfortunately, due to how event annotations are implemented in MASS, child agents generated when this message is sent do not receive the message, and they continue to propagate afterwards. The solution to this is simple. I send the message twice. When an agent finds the point it broadcasts the message, then it migrates, then it broadcasts the message again upon arrival at the next place. This workaround successfully ends the program.

Agent Distribution

Originally, the application only spawned one agent in node 1. This agent then propagated out until the point was found. This solution did not take advantage of all of our computing nodes simultaneously, so I changed the program to spawn one agent on each computing node. More information about this can be found in my Fall report.



Disabling MASS Global Clock

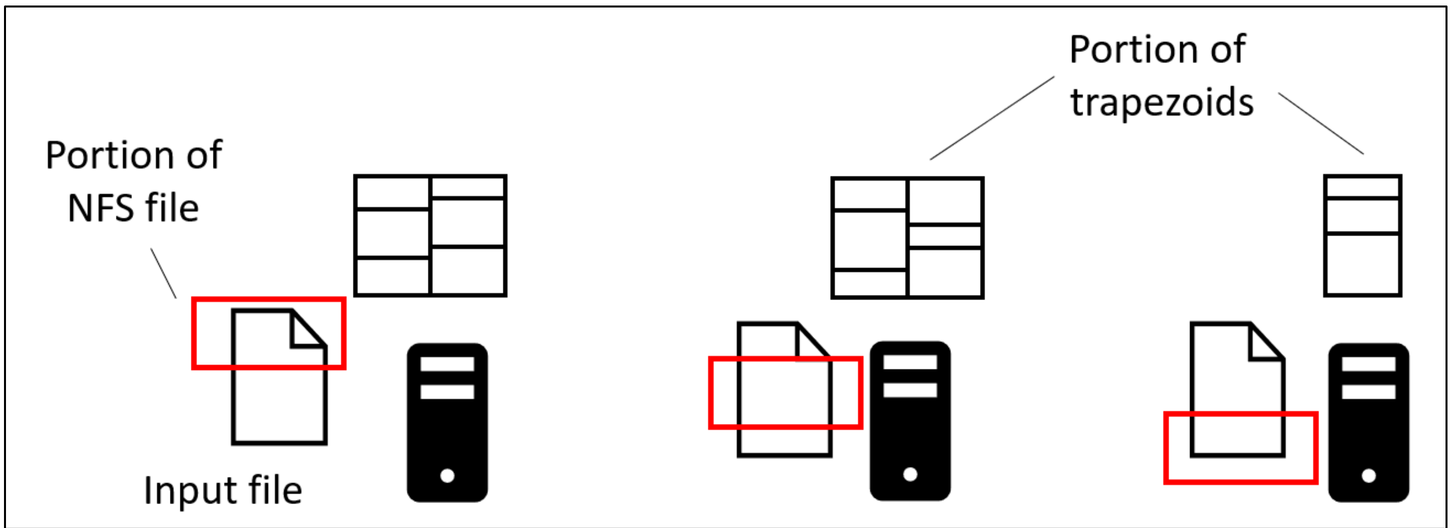
MASS, by default, uses a global clock that synchronizes all events. This is useful for simulations but was not necessary for this algorithm. Unfortunately, the clock slows down the execution time severely as we increase the number of places and agents. I disabled the clock by commenting out a single line in MASS. Anyone planning to test Point Location will have to compile their own MASS library with this disabled if they want to get optimal results. More information on how to do this can be found in the Point Location README in the Application repository.

New MASS Parallel IO Feature

After completing the previous improvements to Point Location and using improvements to the MASS library made by Matthew Sell, I still found the execution time of the program to be too slow. It was taking hours. I found that much of the time was spent during the generation of places.

In the original implementation, the master node reads the input file into memory. The input file contains data for constructing trapezoid objects which is Point Location's abstraction for MASS places. If it is given an input file for 2 million trapezoids, all 2 million trapezoids are created in-memory on the master node even if it will not manage all these trapezoids. This is a waste of memory at best, but it gets worse. The master node then sends all these trapezoids as serialized java objects to each computing node. That means that, following our hypothetical situation, if we are using twelve nodes, we end up sending 24 million trapezoids across the network! This is incredibly redundant and slow.

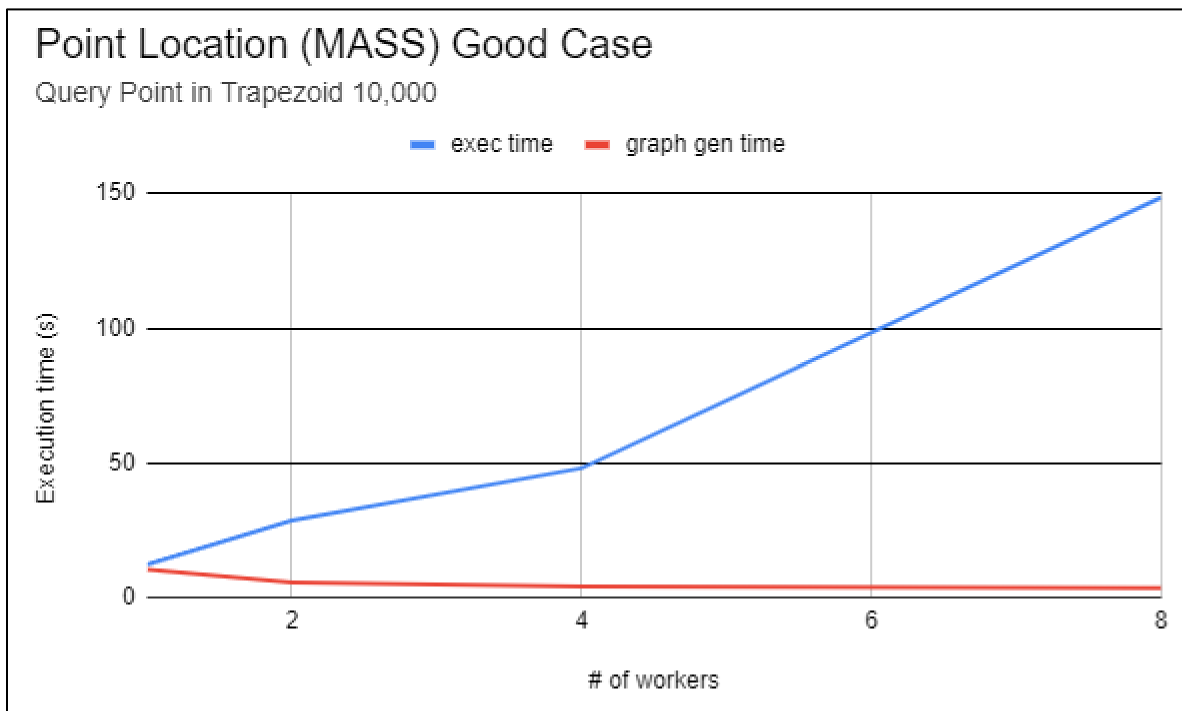
After expressing my concern, I learned that MASS offers a small feature called Parallel IO. It works by taking advantage of the network file system (NFS) that already exists on all CSSMPI and Hermes machines. When the input file is originally uploaded to one of these machines, NSF distributes it across all machines. This is why we are able to log into any machine on the cluster and access files we uploaded on other machines. In other words, NFS has already distributed our input data for us. Sending it over the network during runtime is redundant. Instead, we can use parallel IO to tell each node to read from disk only the portion of the file it needs for computation. Then it can instantiate only those trapezoids into memory.



To get this to work, I had to add a new feature to Parallel IO. Originally, it offered only two modes: READ and WRITE. However, these modes split up the NFS input file by byte. Thus, each node would get an equal share of bytes from the file. The Point Location input files are not separable at the byte level. Each trapezoid required a line to represent itself and each line varied in size. My first solution was to create a sed script that padded my input files. Now each line was the same length (therefore the same size in bytes). This worked for splitting input when we had even numbers of nodes and lines but failed for odd numbers of nodes, as lines would get split down the middle by each node.

My final solution to the problem was to add a new mode to Parallel IO: READ_BY_LINE. This mode follows the same logic as READ but branches off when splitting the file. Since the file lines are padded, it finds the size in bytes of the first line. It then Finds how many lines are in the file by dividing the size of the file by the size of the line. It then determines how many bytes each node should read, rounding each input split to the nearest line. This still allows for fast parallel input, as we jump to the byte in the file that we want to start reading from. Now, users can use parallel IO without having to worry about bytes. As long as they put all the data for each place on a single line and pad each line to be equal in length, they can use parallel IO READ_BY_LINE to improve their MASS performance.

In the case of Point Location, this improved the execution time from hours to minutes. Graph generation is incredibly fast and gets faster with each node we add. Unfortunately, there is still an unknown issue within MASS that slows agent migration as we add nodes.



Conclusion

I hope that my HDFS configuration and testing scripts can be used by future students to improve our results for the rest of the Computational Geometry MapReduce implementations. My Convex Hull folder contains a bunch of documentation, scripts, and configuration files that someone could use to configure their own MapReduce cluster to replicate my results. I hope that future students will find value in my Parallel IO feature in MASS. Finally, I hope that the issues slowing down agent migration in MASS will be identified. I believe that is the last issue preventing us from seeing desirable Point Location results.

Appendix A: Running The Programs

The implementations described in this report can be found in two Bitbucket repositories. The table below provides URL links to each folder in the repository that is relevant.

Convex Hull	Mass Application Repo -> develop branch -> ComputationalGeometry/ConvexHull/
Point Location	Mass Application Repo -> develop branch -> Applications/PointLocation/
Parallel IO	MASS Java Core Repo -> develop branch -> src/main/java/.../Parallel_IO*
Voronoi Diagram**	https://github.com/helzee/voronoi-sequential-2 -> Master branch

**Look at the commit history to see all the files I changed. I also changed Place.java to allow for the new IO type. Unfortunately, my auto-formatter changes are also tracked so you'll have to peruse through a bunch of whitespace changes to find it.*

***I don't mention Voronoi diagram in the report because it was not used for benchmarking and is not finished. See my Fall term report for details on what I believe needs to be done to get it working. For anyone planning to use it, I recommend changing the old Convex Hull algorithm in Voronoi to the new one used in MapReduce/Spark. I fixed a bug in the MapReduce/Spark version that occurred due to integer overflow with large inputs. The other branches of the Voronoi repo are Convex hull attempts that are not up to date with the versions used for MapReduce/Spark.*

Convex Hull MapReduce

To run Convex Hull, first ensure that you have cloned the Mass Java Applications repository and are within the *develop* branch. Navigate to the *Computational/ConvexHull/MapReduce* directory. Read the *README.Md* file.

I created bash scripts for running the tests. These scripts also setup your Hadoop cluster (though I have not tested them outside of my own environment, so use them with care). If you want to use the scripts you need to edit my HDFS config files in the *resources* folder or else they will break your HDFS configuration. Below is a list of the files to edit in the *resources* folder. Generally, if you see my username (helzee) or a port number, you'll have to change it.

In *hdfs-site.xml, mapred-site.xml, and core-site.xml*:

a. Change all occurrences of *helzee* (my username) to your own username.
b. Change all port numbers to your own corresponding port number.

The other files in the folder are there for your reference.

You also must add *JAVA_HOME* to your *.bash_profile* before running the script. You can see what my *.bash_profile* looks like in the *references* directory. Here is the line that you must add to your bash profile:

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk
```

This line may need to change if the directory structure containing the java 1.8 JDK has changed. Also, the *.bash_profile* reference uses the INCORRECT version of Hadoop. The correct version of Hadoop is inserted into your *.bash_profile* in the *setupHadoop.sh* script. The correct version of Hadoop is 0.20.2.

After you have edited the above-mentioned files in *resources,* edited your *.bash_profile* to include *JAVA_HOME*, and restarted your terminal (logout then login), you should run *./setupHadoop.sh* to setup your HDFS cluster. After this script has run successfully, logout and login to the ssh terminal again to allow your terminal to get the new environment variables.

Now you can run the *runAllTests.sh* script to run all test combinations of the program. To change test combinations, just change the numbers in each for loop of the script.

1. Change the number of machines by editing the for loop at line 39
2. Change the number of input points by editing the for loop at line 63
3. Change number of trials by editing the for loop at line 66

While the tests run, you can track execution times by looking at *execution_times.csv.* Results are only ever appended to this file, so delete it if you want to start fresh. To see if the Convex Hulls are correct, look at the *output/output.png.* It is

updated after every run of the program. If the image shows a square-like shape with one section of it missing, then it is correct. If the image appears spiky or breaks the rules of a convex hull, something is wrong.

For more information on setting up MapReduce and HDFS, there is a Microsoft Word document that walks through the setup process called *Hadoop-0.20.0-Installation.docx*. I do not know where it is hosted or where I can host it for visibility after I graduate, so you will have to ask for it from someone on the DS Lab team.

Convex Hull Spark

Running Spark is much easier than MapReduce.

1. Download and unpack Spark 3.3.1
2. Add the path to the unpacked Spark folder as *SPARK_HOME* in your *.bash_profile*.
3. Restart your shell (logout then login)
4. Go to $SPARK_HOME/conf and copy *spark-env.sh.template* to *spark-env.sh*. Command is below. Replace the ports with your corresponding ports:
   ```
   cp spark-env.sh.template spark-env.sh
   ```
5. Open *spark-env.sh* and add the following:
   ```
   SPARK_MASTER_HOST=cssmpi1h.uwb.edu
   SPARK_MASTER_PORT=28600
   SPARK_MASTER_UI_PORT=28601
   SPARK_WORKER_PORT=28602
   SPARK_WORKER_WEBUI_PORT=28603
   ```
6. Save and close *spark-env.sh.* Go back to the ConvexHull/Spark directory in the Mass Applications repo.
7. Run the *run.sh* script. You can edit the for loops within this script to change the test combinations in the same fashion as MapReduce's scripts.

Point Location MASS

As with all of these projects, I have updated the README of Point Location. Check the README.md for my most recent updates as of March 2023.

Before running this program, you should pull the latest MASS Java version from the mass core develop branch. This will contain my Parallel IO code.

You will need to change the MASS library slightly to improve execution performance.

1. Go into the Mass core library (in the develop branch).
2. Go to *AgentsBase.java*
3. Comment out line 1497: Here is what it should look like after you comment it out.
   ```
   // can't fast-forward - just increment Global Logical Clock value
   //clock.increment();
   ```
4. Now go to the topmost directory in Mass Java Core and compile the Mass Java Core library using Maven.
   ```
   mvn -DskiptTests clean package install
   ```
5. Now go to PointLocation directory in the Mass Java Applications repository in the develop branch. When you are in the PointLocation directory, compile it using Maven.
   ```
   mvn clean package install
   ```
6. Now, the program should just work. I wrote scripts to setup your nodes.xml, the input files are already there (they are padded using the *padFile.sed* script). All you have to do is run *runMany.sh*. You can change the point we are looking for in *runMany.sh*, just change the inputs to *run.sh*. Inputs are x then y. You can change the number of agents spawned per node in the *runMany.sh* for loop. You can change the rest of the test configuration (nodes, trapezoids, trials) in the *run.sh* script for loops.

For more information on the MASS ParallelIO feature, I have updated the MASS java core wiki in Bitbucket.

BIBLIOGRAPHY

[1] Mount, D. M. (2002, Fall). Mount 754Lects. Retrieved from University of MaryLand: https://www.cs.umd.edu/~mount/754/Lects/754lects.pdf
[2] https://algorithmtutor.com/Computational-Geometry/An-efficient-way-of-merging-two-convex-hull/