**Documentation:**

**Running**

1. I have included a JAR file if this is handy.

2. But it would be preferred using the assignment.zip, and importing this. Alternatively, I have just zipped the entire project.

3. Using MPJ, java -jar $MPJ\_HOME$\lib\starter.jar assignment.Main -np 4

Where $MPJ\_HOME$ is the home directory of MPJ.

**Introduction**

Since memory in the application has higher priority than computing power (a very large data set that could expand to millions of rows), I have chosen to write it in MPI. This is simply because MPI can expand to allow inter-node (or inter-server) computation. This means rather than struggling for memory on a single node (or server), several nodes can have their independent memory. MPI will make the application more scalable for larger data sets.

If for whatever reason, more computing power is expected of the CPU, this is when OpenMP can be used. OpenMP works with shared memory and thus combining MPI and OpenMP to create a hybrid application would then be best. Each node will perform tasks using OpenMP, while MPI handles the inter-node communication.

**Assumptions**

It is assumed that the speed at which the data is loaded into the application is disregarded. This would be logical to my introduction.

**Objectives**

* Minimize the communication between threads as this can slow down the MPI application.
* Maximize the speed at which the application sorts through the data.
* Minimize the data being transferred between threads.

**Algorithm**

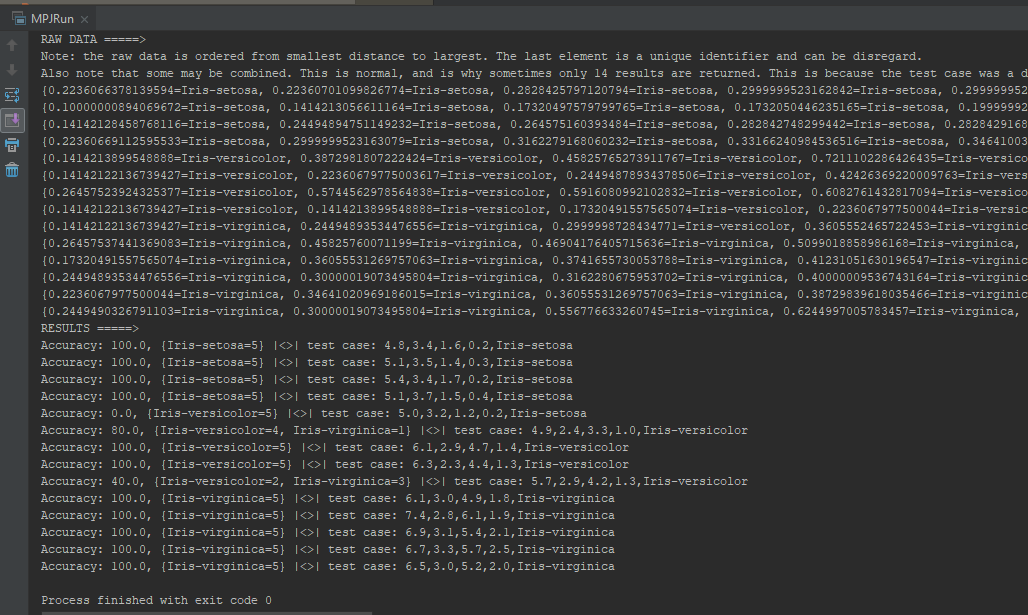
1. Load the data set from the file, *iris.data*. Let the training set be the entire data set, minus a single test vector.
2. Scatter the training set evenly across all threads.
3. Each thread should calculate the distance for each vector.
4. Each thread should sort the results, and should not return any more than *k* results.
5. The main thread will gather all the results, and find the top k results. No more than *k\*n* results should be returned to the main thread, where *n* is the number of threads.
6. The main thread will find the best matching plant type based on the final results.

**Review**

After reviewing the algorithm during implementation, it was required to broadcast the test cases to each thread.

**Testing**

Here is a screenshot of an example output,



Each thread returns their set of results for each test case. But they do not return any more than K, and they are also sorted so they are the K most optimal results.

**Conclusion**

Yes, for this application OpenMP would be most ideal since it works with shared memory which means there will be no bottlenecks around transmitting data between threads. However, I still stand with the decision on using MPI for the reason if this data set was to expand, or the host was not capable of computing the results: MPI would be faster. Also, with multiple nodes or servers, memory isn’t a problem which means larger data sets aren’t a problem either. It’s essential to minimize communication between threads for optimal performance.

Deadlock can still occur with MPI even though it has distributed memory, however comparing with OpenMP there are risk factors of race conditions.

**NOTE:** Regarding selecting five test cases from each plant type, this could be CPU intensive: continuously finding random integers until five for each is fulfilled. Considering this is a single core task, I have left it out.