15618-Final Project Report

Parallelism Machine Learning Algorithms Design and Tradeoff Analysis

Tianqi (Andy) Wu, Alex Zhang

1. Summary

Machine learning algorithms have been a hot topic, albeit their time-consuming quality. Our team decides to achieve parallelism to reduce time for supervised and unsupervised algorithms with APIs such as OpenMP and MPI while ensuring their correctness. We parallelized the two APIs based on straightforward examples such as KNN and K-means clustering with at most ~88x for KNN and at best ~14x for K-means. We combined both OpenMP and MPI together and yielded successful results.

2. Background

Goal. Our team is able to parallelize KNN and K-means with both OpenMP and MPI. Both reached significantly better results compared to their sequential implementation.

Our code repository consists of the following:

Data folder. The data folder consists of original Abalone data from the UCI repository, and another dataset generated with a python script (generator.py) called mass_abalones. The scripts and the large dataset are created to evaluate performance based on various datasets and thread numbers.

Key data structure. The key data structure is the Abalones class, which includes all columns from the abalones data. Given the complexity of loading data in C++ (high performance and completeness, but not ease of development), it's hard to load with arbitrary columns and extend to all data structures, and we decide to stick with abalones data of various sizes.

Algorithm and Explanation. KNN is based on the idea that data points that are closest to a data point of interest will be most similar with itself. In a sequential version, it loops through testing data points, each of which finds the closest points and makes classification/regression decisions based on majority vote/(weighted) average. K-means clustering, on the other hand, is based on finding a dataset of similar characteristics by clustering. It initializes with several “cluster centers”, and then labels each data point by assigning its label to the closest cluster center; after that, each data point with the same label (weighted) averages to generate a new cluster. Such will repeat until (1) it reaches the maxIteration or (2) it converges.

Algorithms input and output. The KNN regression algorithm inputs the entire abalones data set and outputs the regression result for each abalone; the K-means algorithm inputs the entire abalones data set and outputs the final cluster assignment. The output files are stored in output.txt when running the algorithms. We have validated their correctness by comparing values with Scikit-learn with a Jupyter Notebook in the validator folder. Specifically, the K-means algorithm has a DEBUG flag for seeing the final clusters.

What parts would benefit from parallelization. For KNN: For every testing data point, KNN computes its distance with every training data point. This is very expensive for a enormous dataset: if we have 30000 testing data points and 10000 training data points, we have to compute 30000 * 10000 euclidean distances. We also see that this calculation is highly parallelizable: there is no dependency between the euclidean distances’ calculations so no mandatory communication between threads are needed. We have noticed that since each data point has to find out the K nearest neighbors, the computational cost for each data point on finding the K nearest neighbors is about the same, providing reasons for static assignment.

For K-means: two phases are valid for parallelization: 1. finding where cluster labels the data points belong to, and 2. averaging all data points with the same labels for new clusters. Since both processes are computationally heavy (we cannot avoid going over the whole dataset for clustering accurately) are based on the closest points and each single datapoint/cluster will not interfere with other data point/cluster's clustering process, it makes parallelization rewarding and easy to implement. Given that the initialization process is sequential by nature (furthest point heuristic, for example), the authors decide not to parallelize it, but included in the timing area to discuss whether initialization method may significantly slow the algorithm down.

Break down the workload

For KNN: We also should have great locality: for each testing data point, we will access the training data points consecutively, implying a high spatial locality. We thought SIMD could speed up the computation of euclidean distances, but the result indicates otherwise (we will explain this later).

For K-means: The dependencies are as follows: The main algorithm consists of a series of loops. Each iteration of the loop is dependent on the previous loop. Each loop consists of two important steps: finding the label based on the cluster, and then finding the new cluster based on these labels.

Approach

We parallelized the algorithms with the following design:

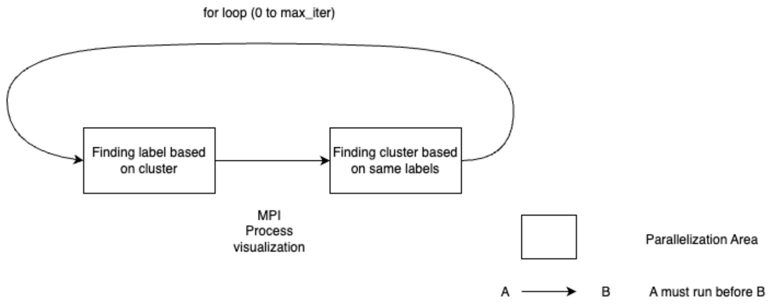
Language and API. We used C++, because it is known for high performance and we can make sure the measurements will go well. For API choices, we uses OpenMP and MPI.

Change of Serial Algorithm

For KNN's OpenMP implementation, we simply let each thread compute the classification/regression of a portion of the testing dataset (We will explain why this simple approach works best later.)

For KNN's MPI implementation, each thread calculates the nearest neighbors by receiving and holding a portion of the testing dataset based on static assignment instead of dynamic assignment given that (1). Since each datapoint calculates the same amount of K nearest neighbors, the work that each thread does to deal with each single datapoint is almost equal (2). Sending a chunk of continuous data is better than sending a series of single data.

For Kmean’s OpenMP implementation, given the dependency graph shown below, we decide to make parallelism happen when (1) finding the labels based on the closest cluster (2) Finding clusters based on the average of elements with the same label. This will make sure that parallelism will be exerted on both finding labels based on clusters and finding clusters based on the same labels.


for loop (0 to max_iter)
Finding label based on cluster
Finding cluster based on same labels
MPI Process visualization
Parallelization Area
A
B
A must run before B

For MPI, all threads will be broadcasted with all data of abalones at the very beginning of the program. Then, for each iteration, they will first be sent a series of clusters and be statically assigned a subset of abalones to work on and label them according to the clusters, and send the sub-labels back to the master thread(pid 0). After all threads have finished the first step, each thread will be broadcasted with the labeling result and assigned to generate a number of new clusters by gathering and averaging data points with the corresponding label. We still used static assignment, given that each datapoint has to go through the full list of clusters to decide which cluster is the closest, and each cluster will be decided only via the average of all the labels corresponding to it.

Finally, we implemented KNN and Kmeans,using OpenMP and MPI together.

Approaches That Didn’t Work Well

For KNN, we experimented with dividing the training data set, instead of dividing over the testing data set. The implementation of this is in knn.cpp, with the function name knn_parallel_pq. Here is a table comparing the speedup for approach we decided to use and knn_parallel_pq, running on 50000 abalones:

	Approach we decided to use	knn_parallel_pq
2 Processor	1.94	1.80
4 Processor	3.80	2.89
8 Processor	6.73	3.89

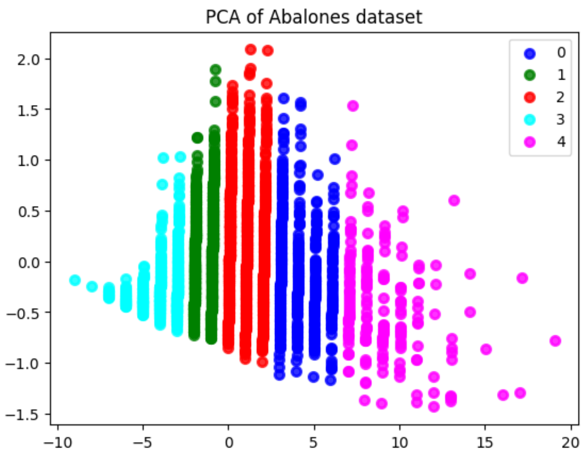
There are 2 reasons for why dividing over the training data set produces a higher speedup. We realized that if we divide over the training data set, then for every testing data point, we have to synchronize once. But if we divide over the testing data set, we only have to synchronize once for the entire algorithm. Moreover, if we divide over the training data set, the number of times we will create and destroy threads equals the number of testing data points, whereas dividing over the testing data set will only create and destroy threads once.

RESULTS

Measurement and Performance: We are able to achieve about 88x speedup for KNN and 10x speedup for K-means when using default settings.

For correctness, we have a Jupyter notebook file in the validators folder called FinalProjectResultValidator.ipynb that helps confirm our results meet the correctness criteria.

For the K-means algorithm, here is an example graph after visualization for K-means clustering with pseudo-random initialization. You can find that the result is quite convincing, since similar data after PCA are grouped together after PCA is run.



Dataset and How the requests are generated: For the dataset, the authors used original abalones data from UCI. The authors of this report are able to generate arbitrarily large data on their own based on a Python script generator.py and the original abalones data from UCI abalones.data for testing and exploring parallelism results. Such data called mass_abalones.data will provide a relatively good estimate of how our parallel implementation will perform on larger datasets. Both dataset are in the dataset folder. The author also encourages readers to generate their own datasets.

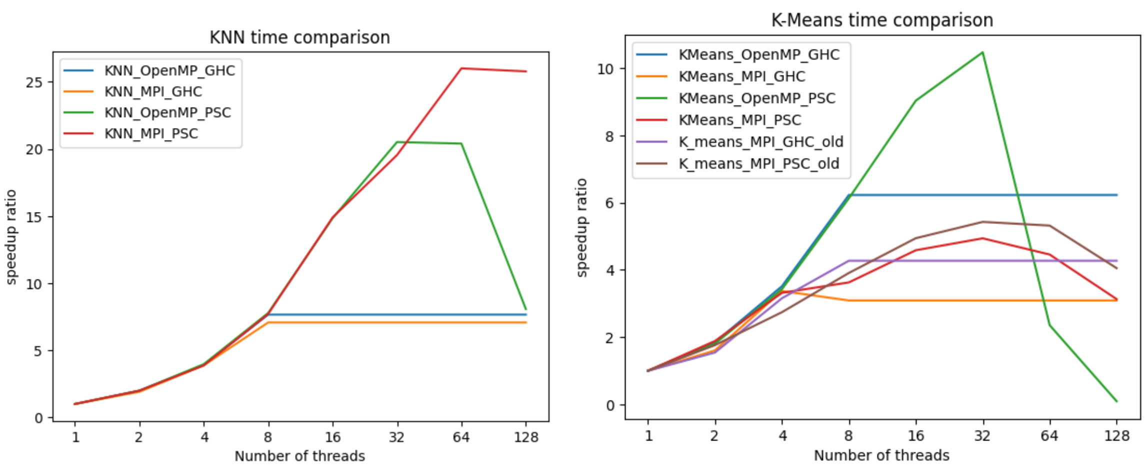
Experimental setup, size of the inputs, and results. We initially used experimental setup with the original abalones data (we only removed commas between columns for better C++ parsing). The size of inputs is ~5000 data points with ~10 rows; the data points are split between training/testing in an around 7/3 portion.

Graphs of speedup or execute time

Here are graphs based on various settings. Please note that our baseline is time run on a single thread:

1. Using either OpenMP or MPI for both KNN and K-means(we used K = 10 for KNN, and K=5 & maxIter = 100 for K-means, based on the original abalones data). Please note that we have included

the initialization process for both KNN and K-means in the timer for consistency for consistency reasons, and will continue that for the following measurements:



Graph: KNN and K-means Time Comparison, based on the original Abalones data (5000 data points)

You can see that KNN for OpenMP and MPI version has almost the same speedup on GHC; KNN for OpenMP on PSC performs worse than KNN_MPI version. For K-means time comparison, OpenMP performs better than MPI on PSC when the number of threads is from 8 - 32. The old version for MPI only parallelizes over labeling over the whole dataset, while the new version parallelism over both labeling and cluster finding. However, such parallelism only works better when the number of threads is small.

2. Using a combined version of OpenMP and MPI (we only discuss KNN here, as we found combining OpenMP and MPI on Kmeans doesn't increase speedup): We tested on different combinations of numbers of MPI processes, and how many OpenMP threads per MPI process. We only look at the performance when the dataset size is 59000, and running on PSC(128 cores):

# of MPI process-> # of OMP threads per MPI process↓	2	4	8	16
2	3.956	7.884	15.56	31.13

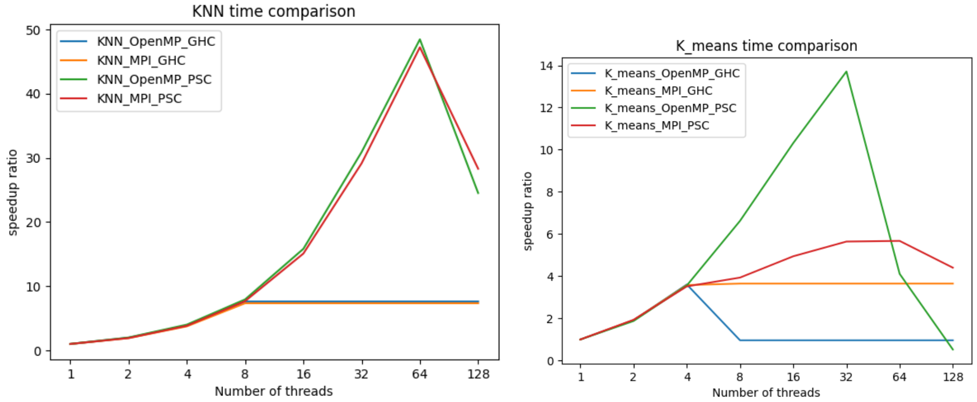
4	7.899	15.56	30.14	59.42
8	15.56	30.60	56.42	88.49

We found out that by utilizing all 128 cores on a psc node, we are able to achieve 88.49x speedup. This results is nice, as we showed the speedup is almost linear with the number of cores we use.

Different problem size & Corresponding Behavior:

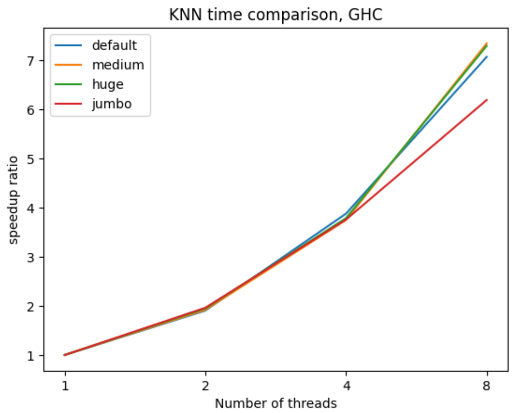
We decide to experiment with both how mass dataset would influence performance, and how performance will change when dataset size changes.

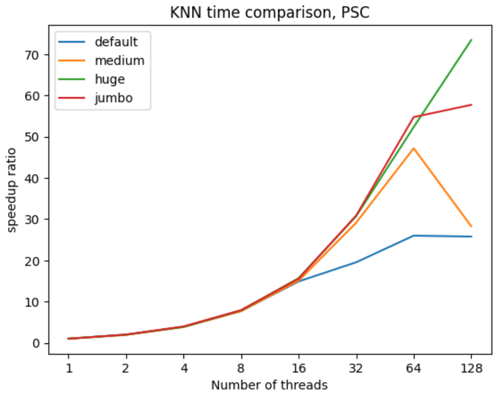
Speedup with both KNN and K-means, based on the mass_abalones data:



Graph: KNN and K-means Time Comparison, based on ~15000 data points

We can find out the result is relatively similar from smaller data when comparing within the same dataset. Then, we try to compare the speedup of the smaller ones from the larger ones. Here are the results from comparing the larger dataset and smaller one based on various datasets, for the same algorithm (KNN_MPI):

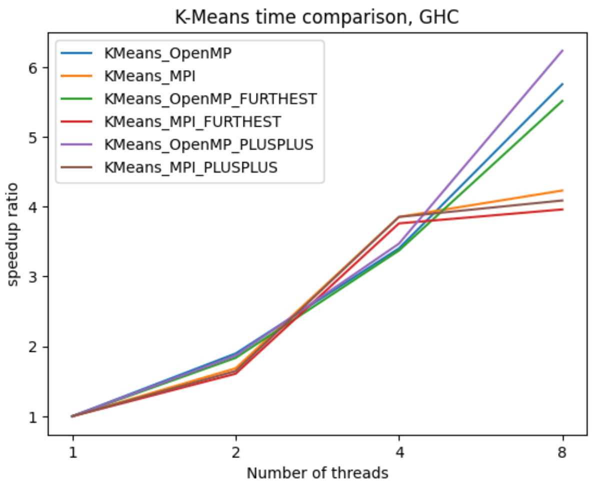


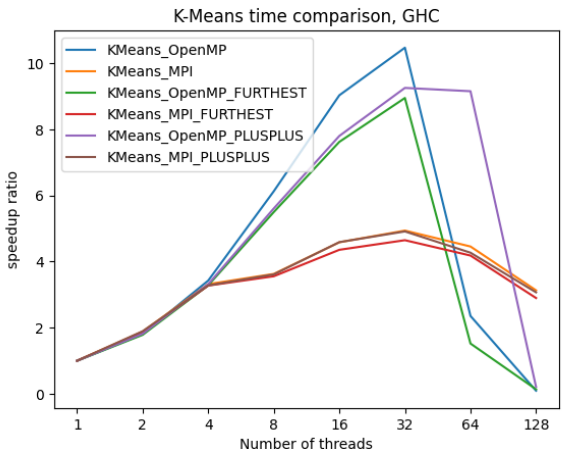


Generally speaking, the speedup is almost the same when the number of threads is small. For GHC, when the number of threads gets huge, the speedup decreases when the size of the dataset increases; however, for PSC, when the number of threads gets huge, the larger the dataset, generally the more the speedup. We are curious why the MPI for KNN in GHC gets a counterintuitive result. Therefore, we used performance tools to analyze the cache miss rate when K = 10, np = 8, and find cache miss to be a potential reason why the performance decreases (we are not able to do the same perf evaluation on PSC because of software limits):

	default(~5000)	medium(~15000)	Huge ~(25000)	Jumbo (~55000)
	10.81	7.366	13.531	27.062

We also figured out that for K-means, there is some evidence that various initializations will influence the speedup. However, the influence is not usually significant, and may be ignored in some circumstances.





What limited your speedup. We will focus on the openMP version of KNN, as it has the poorest speedup among the 4 algorithms we implemented (6.5x on a 8-core machine). The analysis of K-means will be included in the deeper analysis portion.

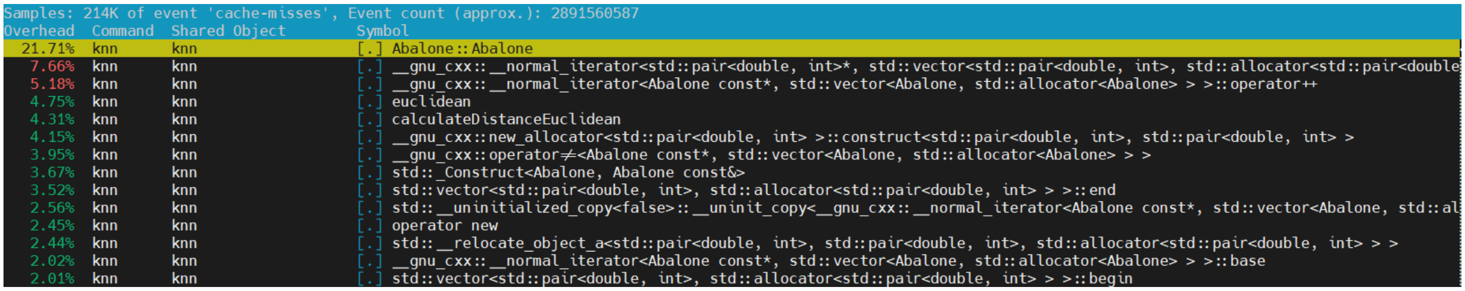
We believe the limiting factor is the cache miss rate, due to the dataset being overly large. We produce a table that measures the cache miss rate, by running command perf stat -B -e cache-references, cache-misses ./knn on various data set size and various processor count:

Processor Count-> Data Size ↓	1 Processor	2 Processor	4 Processor	8 Processor
~19000	0.047	0.082	0.873	12.920
~29000	0.268	0.876	10.067	33.472
~54000	6.62	13.255	35.975	50.256

From this table, we notice the cache miss rate shows a positive correlation with the number of processors and the data size. Both of the correlations naturally makes sense; as the data size increases, the working set each processor needs to handle increases, thus the capacity misses increase. When the number of processors increases, competitive access to the shared memory cache increases (an example of competitive cache misses: processor A brings in a needed cache line

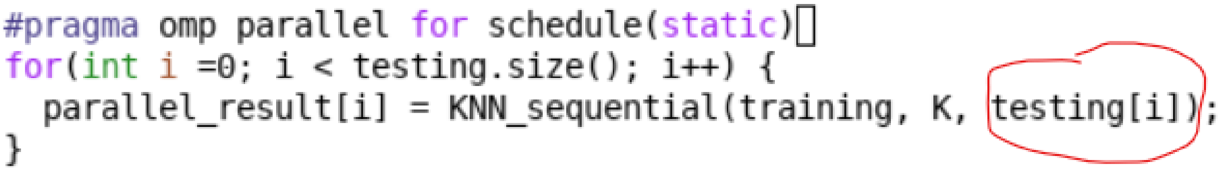
C1 from main memory into the shared memory cache. Then after a while, processor B brings in the needed cache line C2 into the shared memory cache, and it evicts C1 from shared memory. After a while, processor A needs some data that is not in its local cache, but is contained in C1. However, because processor B evicted it already, processor A needs to bring that cache line from main memory into the shared memory cache again.) (https://arxiv.org/abs/1701.01630)

As we reasoned above, the increase in cache miss rate is somewhat natural. Nonetheless, we still investigate where does the cache misses happen. We record the number of cache misses with abalone data size = 54000 and 8 processors. Here is a snippet of the report:

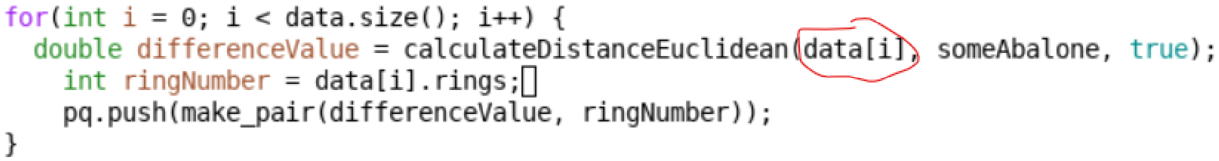


```
Samples: 214K of event 'cache-misses', Event count (approx.): 2891560587
Overhead  Command  Shared Object        Symbol
  21.71%  knn      knn                  [.] Abalone::Abalone
   7.66%  knn      knn                  [.] __gnu_cxx::__normal_iterator<std::pair<double, int>*, std::vector<std::pair<double, int>, std::allocator<std::pair<double
   5.18%  knn      knn                  [.] __gnu_cxx::__normal_iterator<Abalone const*, std::vector<Abalone, std::allocator<Abalone> > >::operator++
   4.75%  knn      knn                  [.] euclidean
   4.31%  knn      knn                  [.] calculateDistanceEuclidean
   4.15%  knn      knn                  [.] __gnu_cxx::new_allocator<std::pair<double, int> >::construct<std::pair<double, int>, std::pair<double, int> >
   3.95%  knn      knn                  [.] __gnu_cxx::operator≠<Abalone const*, std::vector<Abalone, std::allocator<Abalone> > >
   3.67%  knn      knn                  [.] std::_Construct<Abalone, Abalone const&>
   3.52%  knn      knn                  [.] std::vector<std::pair<double, int>, std::allocator<std::pair<double, int> > >::end
   2.56%  knn      knn                  [.] std::__uninitialized_copy<false>::__uninit_copy<__gnu_cxx::__normal_iterator<Abalone const*, std::vector<Abalone, std::al
   2.45%  knn      knn                  [.] operator new
   2.44%  knn      knn                  [.] std::__relocate_object_a<std::pair<double, int>, std::pair<double, int>, std::allocator<std::pair<double, int> > >
   2.02%  knn      knn                  [.] __gnu_cxx::__normal_iterator<Abalone const*, std::vector<Abalone, std::allocator<Abalone> > >::base
   2.01%  knn      knn                  [.] std::vector<std::pair<double, int>, std::allocator<std::pair<double, int> > >::begin
```

From this report, we see that the Abalone’s copy constructor incurs the largest percentage of cache misses. So we inspect our code and see where did we call the copy constructor:



```
#pragma omp parallel for schedule(static)
for(int i =0; i < testing.size(); i++) {
  parallel_result[i] = KNN_sequential(training, K, testing[i]);
}
```



```
for(int i = 0; i < data.size(); i++) {
  double differenceValue = calculateDistanceEuclidean(data[i], someAbalone, true);
    int ringNumber = data[i].rings;
    pq.push(make_pair(differenceValue, ringNumber));
}
```

However, after several failed attempts to reduce cache miss rate(such as padding the Abalone’s size to a multiple of cache line, copying a “testing” vector for every thread, sharing the training and testing array across threads), we realize that the competitive cache misses(caused by too many processor and a large data set) might be present no matter what do we do.

As mentioned before, we initially thought SIMD will speed up the computation of euclidean distances. But in reality, running KNN (with data size ~= 59000 and 8 processors) with SIMD on computation of euclidean distances is 7% slower than running KNN without SIMD on computation of euclidean distances. After doing some research, we found out that computation of euclidean distance is memory-bound due to our data being floats, so using SIMD doesn't give speedup.

For the K-means implementation, the authors believe that the algorithm cannot reach perfect speedup because of the following reasons:

1. Initialization phase is done on the master thread (PID0). So initialization phase has to be sequential in nature, and according to Amdahl's Law, what constrains the speedup of the program is inherently the part that cannot be done in parallel. Previous visualization on initialization has also shown that initialization plays a role in affecting performance.
2. Implicit barriers between each phase. As previously mentioned, each iteration of K-means has to make sure that a phase can only continue after the previous phase/iteration has finished. Because of that, threads that run inherently faster will still have to wait for all other ones until all threads reach the finish line. Such will restrict speedup, making it not able to reach the perfect line.
3. Communication cost. It is known that MPI communication will take a significant amount of time. The following communications must happen: 1. Communication at the very beginning, for transferring the static assignment parameters(offset, chunksize, etc), and one-time transferring of the whole abalones array, 2.Communication in each loop for [phase 1] sending updated clusters to each thread, and collecting abalones labeling [phase 2] sending each thread the complete labeling array, and collecting the updated cluster from them. These steps may create potential memory bottlenecks.

Deeper analysis.

1. **Time breakdown of each section of algorithm, on MPI KNN & KMenas, OpenMP KNN:**

For OpenMP Kmeans(on GHC): It consists of two portions: (1). Assigning each data point a cluster assignment (2) Averaging the data points in a cluster to form a new data point. A table that measures the percentage of time each step takes:

Phase-> Thread Count ↓	Phase 1	Phase 2
1	86.5%	13.5%
2	82.4%	17.6%
4	78.7%	21.3%
8	75.7%	24.3%

This results makes perfect sense: because there are much more data points than clusters, we hypothesize that phase 1 should have more parallelism to exploit than phase 2. As the number of threads increase, we indeed observe the proportion of time spent on phase 1 decreases, implying that phase 1 exhibits more parallelism than phase 2.

For MPI KNN: It consists of two portions: (1). One-time communication hyperparameters (chunksize, etc) and part of the array; (2). Computation and (3) sending/gathering the result.

Here is a percentage measurement of these steps with a form of a table:

Thread count	Phase 1(comms)	Phase 2 (compute)	Phase 3(comms)
16	1.54% (0.000737s)	98.35% (0.047890s)	0.046% (0.000022s)
32	2.49% (0.000916s)	97.33% (0.035784s)	0.08% (0.000032s)
64	6.18% (0.001743s)	93.13% (0.026248s)	0.56% (0.000157s)
128	12.8% (0.003454s)	84.6% (0.022911s)	2.12% (0.000485s)

We can deduce that communication will be costly as the number of threads increases from this table.

MPI K-means: There are several sections, divided coarsely since we have already identified the number of threads a problem:

1. One-time communication: sending hyperparameters and data;

2. generate label from clusters;

3. generate clusters from averaging the labels.

Aware that more iterations will make the first one-time communication less costly , the authors used maxIter = 100 to isolate other causes.

Thread count	Phase 1(one-time comms)	Phase 2 (labeling)	Phase 3(clustering)
4	1.37% (0.000489s)	78.48% (0.028094s)	19.99% (0.007155s)
8	1.46% (0.000486s)	46.7% (0.015499s)	51.63% (0.017135s)
16	2.28% (0.000607s)	32.87% (0.008765s)	64.63% (0.017237s)
32	2.49% (0.000603s)	24.24% (0.005860s)	73.02% (0.017650s)
64	6.33% (0.001692s)	21.91% (0.005649s)	72.33% (0.019344s)
128	8.81% (0.003539s)	27.90% (0.011201s)	63.09% (0.025329s)

We can find out that generally, when the number of threads increases, phase 2(labeling) takes fewer portions of time, while phase 3(clustering) takes longer. Thus, if we are working on a system with fewer thread counts, it would be wise to optimize phase 2 more; if we are working with more threads, we should optimize phase 3. It would also be advisable to reduce communications by switching to less threads/allow only sequential execution on the master thread.

2. **Where is there room to improve and how**:

Based on the measurements, we can find out that, even though more threads would generally mean better speedup, it also comes at an increased communication cost. Therefore, here are some ways to improve:

1. Reduce communication: for example, for K-means MPI, sorting out data that have certain labels and then passing them to each thread. This will reduce the communication space, at a possible cost of being unable to broadcast data to all given each thread will receive unique data.

2. Improve the sequential version. If possible, write in a more cache-coherent method; if communication starts dominating the time, consider switching to sequential execution with a single powerful thread.

Analysis on choice of machine target

We think the GPU would have been a better choice. The main advantage of using GPU is that there are more cores than CPU. We realized our algorithm for KNN and Kmeans mostly do arithmetic computations, which is perfect for GPU, which has much more cores than CPU that are specialized in dealing with simple arithmetic computations. Also, GPU has a larger memory bandwidth than CPU. As we discussed above, we couldn't parallelize the computation of euclidean distances, because we reached a memory bottleneck. With a higher bandwidth, we might be able to overcome the memory bottleneck and exploit the parallelism in computation of euclidean distances.

References

The authors also used official websites for MPI, Python/Bash scripting and visualization.

The authors also referred to websites such as Stackoverflow. Here is a list of citations that the authors visited for evaluation purposes:

https://arxiv.org/ftp/arxiv/papers/1701/1701.01630.pdf

https://stackoverflow.com/questions/65548244/no-speedup-for-omp-simd-reduction

https://stackoverflow.com/questions/6565040/sse2-double-multiplication-slower-than-with-standard-multiplication

The authors start from scratch without referring to existing C++ parallel programming implementations. Note that some of these links are appended with the helper code files (.py, .sh files) themselves.

List of Work done by each Student

We believe that both authors of this report constitute an equal and fair portion of the full project.

Work	Responsibilities
Building C++ sequential infrastructure & algorithm	Tianqi (Andy) Wu
OpenMP naive implementation (correctness checked) & Advanced optimization (PQ methods, etc)	Alex Zhang
MPI naive implementation & Advanced optimization (MPI over both labeling and clustering)	Tianqi Wu
Automated Scripting (running OpenMP, MPI, generating dataset)	Both
Data visualization for analysis	Both
Combining OpenMP and MPI	Alex Zhang
Performance Analysis (cache miss analysis, parameter tuning, evaluation, reasoning)	Both
Construction of the report	Both