Parallelizing the Expectation-Maximization Algorithm for Gaussian Mixture Models in OpenMP and MPI

Summary

In this project, we parallelize the Expectation-Maximization Algorithm for Gaussian Mixture Models in C using OpenMP and MPI. We use similar implementations and then compare the timing of the two with simulated datasets with varying array lengths, shapes, and levels of noise.

Background

The Expectation-Maximization Algorithm

If we have unlabeled data, can we:

1. *Identify which data belong to which sub-population?*
2. *Find the parameters of the sub-distribution?*

The Expectation-Maximization algorithm can answer both of these questions. It is used to simultaneously find the parameters of a mixture model and calculate the labels of each point in the dataset. Because of this, the EM algorithm is commonly used when there is missing or unobserved information. [5]

E-step: Calculates likelihood values of unobserved variables

M-step: Maximizes the log-likelihood of the parameters using the values from the E-step

The maximization of the parameters occurs by updating the parameters at each iteration. Once source describes it as, "the EM algorithm is actually maximizing a lower bound on the log likelihood [4]." It does this using a mixing coefficient α, or weight, for each component which is updated at each iteration with the parameters. To get the labels, we use the highest likelihood among the cluster estimates at each point.

Gaussian Mixture Models

The Gaussian mixture model (GMM) is used when we can assume that observed data has been generated from multi-modal Gaussian distributions. For the EM algorithm, this means that the probability density function of the mth component, or label, is:

$$p(\mathbf{x}|\theta_m) = \frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma_m|^{\frac{1}{2}}} e^{-\frac{1}{2}}(\mathbf{x}-\mu_m)^T\Sigma_m^{-1}(\mathbf{x}-\mu_m) \quad (1)$$

where $\mathbf{x}$ is the input data and $\theta_m = (\mu_m, \Sigma_m)$. We will estimate the mean and variance of GMM's using EM.

Major applications of GMM for EM include clustering, density estimation, and image and signal processing. It is used often in fields such as machine learning, computational biology, and natural language processing. [6]

Approach

The general approach for parallelization is as follows:

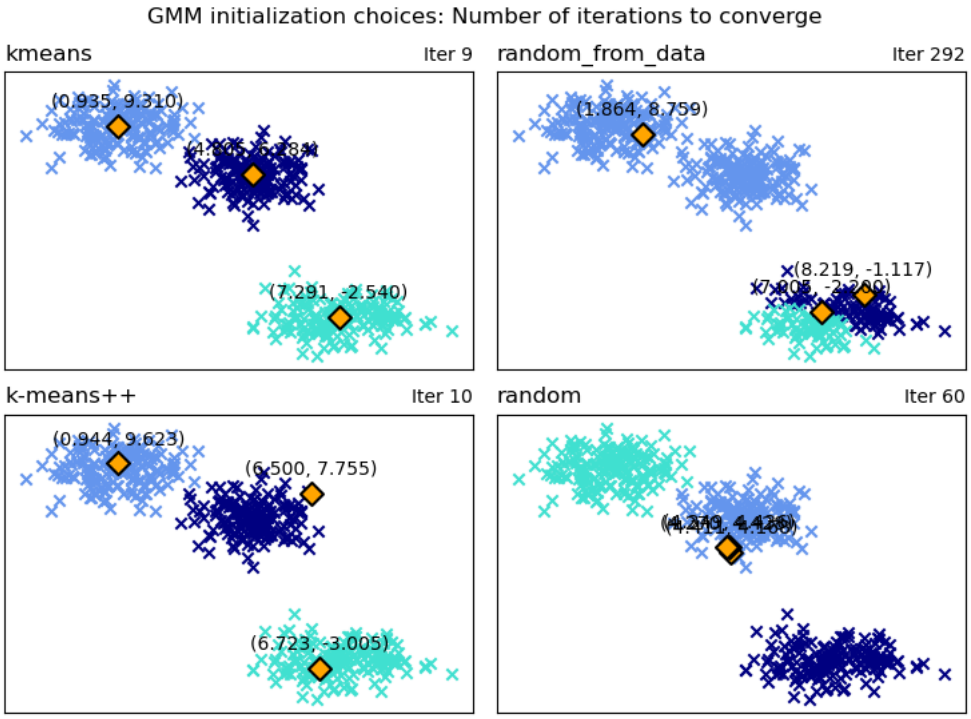
Algorithm Parallelizing EM for GMM

```
Initialize parameters
while Iteration < Maximum do
    E-Step: Split threads over array and calculate the likelihood
    Synchronize
    M-step: Update α and μ
    Synchronize
    M-step: Update Σ
    Synchronize
    if Criterion < Tolerance then
        break
    end if
end while
Output labels and timing
```

where the likelihood is $h_m^{(j)}(\mathbf{x}^i) = \frac{\alpha_m^{(j)} p_m(\mathbf{x}^i|\theta_m^{(j-1)})}{\sum_{k=1}^K \alpha_k^{(j)} p_k(\mathbf{x}^i|\theta_k^{(j-1)})}$, the weighting coefficient is $\alpha_m^{(j)} = \frac{1}{N}\sum_{i=1}^N h_m^{(j)}(\mathbf{x}^i)$, and the parameters are calculated as $\mu_m^{(j)} = \frac{\sum_{j=1}^N h_m^{(j)}(\mathbf{x}^i) * \mathbf{x}^i}{\sum_{j=1}^N h_m^{(j)}(\mathbf{x}^i)}$, and $\Sigma_m^{(j)} = \frac{\sum_{i=1}^N h_m^{(j)}(\mathbf{x}^i)(\mathbf{x}^i - \mu_m^{(j)})(\mathbf{x}^i - \mu_m^{(j)})^T}{\sum_{i=1}^N h_m^{(j)}}$ where j is the iteration number, N is the array size, and K is a user-defined number of components. [6]

Initialization

Figure 1: Initialization Testing



For each component, the mean μ of size Kxd, the covariance matrix Σ of size Kxdxd, and the mixing coefficient α of size K need to be initialized, where d is the dimension of each point in the array. The mixing coefficient is initialized uniformly, and the sum of all of the mixing coefficients should equal 1: $\alpha_1 = \alpha_2 = \ldots = \alpha_K = \frac{1}{K}$. The covariance matrices are all initialized to be equal to the global covariance matrix: $\Sigma_1 = \Sigma_2 = \ldots = \Sigma_K = \Sigma_{global}$.

Because the EM algorithm for GMM is a local optimization method, the results are sensitive to initial estimates of the component means. There are 4 common methods used for initializing the means, shown in figure 1:

1. K-means clustering (kmeans)
2. The initialization method of K-means clustering, where the first mean is chosen randomly, and subsequent means are chosen by weighting to favor points further away from existing means (k-means++)
3. Selecting K random points from the data (random_from_data)
4. Adding small perturbations to the mean of the entire dataset (random)

Using scikit-learn in Python, we compared the performance of the 4 methods and found that selecting random points from the data (random_from_data) was unable to correctly

identify clusters, particularly in the case where the initial means were close together. We decided to implement adding small perturbations to the mean of the entire dataset (random) as our focus was on the EM algorithm and another group is doing the K-means algorithm. Later, we found that k_means++ converged faster and more consistently so we implemented that instead.

E-Step and M-Step

Figure 2: M-Step Weights and Dependecies

$$\alpha_m^{(j)} = \frac{1}{N} \sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)$$

$$\mu_m^{(j)} = \frac{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i) * \mathbf{x}^i}{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)}$$

$$\Sigma_m^{(j)} = \frac{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)(\mathbf{x}^i - \mu_m^{(j)})(\mathbf{x}^i - \mu_m^{(j)})^T}{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)}$$

Parallelization is done following the Kwedlo paper [6]. The E-Step is relatively simple to parallelize because the operations across the data array are independent. Because of this, we simply divide up the array and its likelihood calculations using for loops. This step takes in a matrix of N data vectors (i) * d dimensions and outputs a matrix of likelihoods that is N data vectors (i) * K components (m).

The M-Step requires two main parallelization steps. First, we calculate the weights $w_j = \sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)$ and $v_j = \sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i) * \mathbf{x}^i$ in order to then calculate α_m^j and μ_m^j. Second, since Σ_m^j is dependent on the updated μ_m^j, we synchronize the threads after the first step and then calculate Σ_m^j. For the OpenMP implementation, we use `# pragma omp parallel for` to both steps. For the MPI implementation, we similarly split up the indices among the threads and have them loop through a local first and last index. The main thread collects the values of H after the E-Step since each thread works on a different section of H. In the M-step, we use local variables for w, v, and the numerator of Σ and reduce them using `MPI_Reduce`. At the end of this step, the main thread sends the updated parameters to the other threads.

Convergence

The convergence criteria is taken from the Kwedlo paper as $\frac{\log(h_m^{(j)}) - \log(h_m^{(j-1)})}{\log(h_m^{(j)})} < \epsilon$ where $\epsilon \ll 1$ [6]. We adjust it to be $|\frac{\log(h_m^{(j)}) - \log(h_m^{(j-1)})}{\log(h_m^{(j)})}| < \epsilon$ so that the algorithm iterates for long enough. If the absolute value is not added, whenever the log-likelihood at the current step is worse then the previous step the program will end. Since the means were initialized randomly

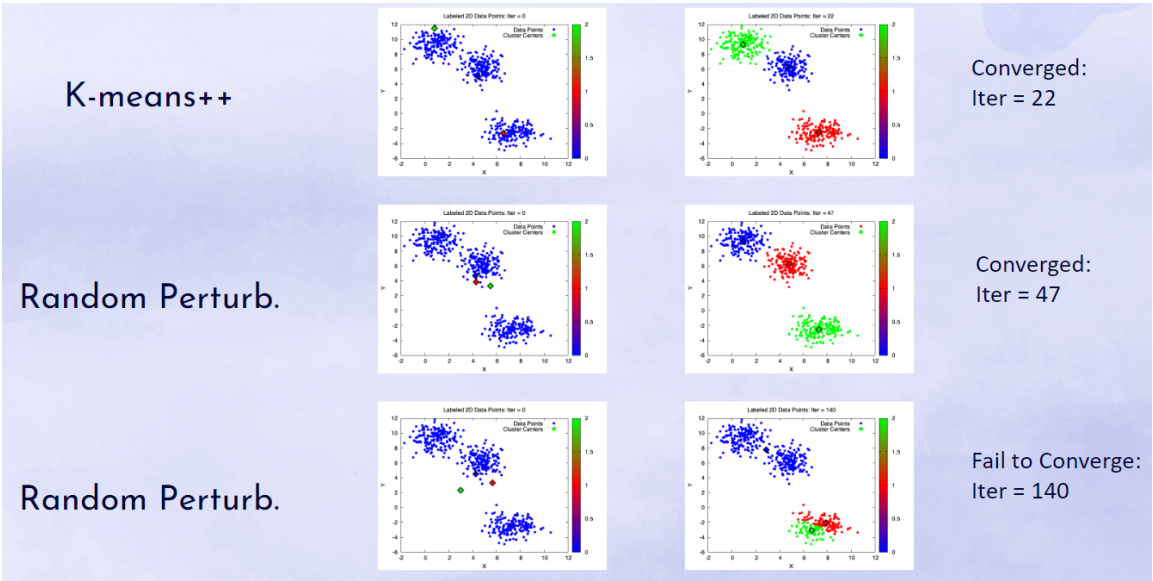
about the global mean, the algorithm has a tendency to converge to the global mean quickly with a better likelihood and stop before the actual means are found. With K-Means++ we leave this adjustment so that it has more time to converge accurately.

Results

We tested the algorithm on results of size 10^p where p $\in$ 2, 3, 4, 5, 6. The datasets are 2D comma-delimited files titled with data_p.csv (eg. `data_2.csv` is for 10^2). We asked for 5 clusters from each dataset. The files are generated randomly from a 5-modal normal distribution, and thus we have the true means. We also tested the data on varying shapes as well as noise and variance levels. Scatter plots were generated locally using the `gnuplot` package.

Convergence

Figure 3: Initialization Performance



We found that convergence was heavily affected by mean initialization. As shown in figure 3, KMeans++ was more accurate and less likely to fail then implementing random perturbations around the global mean. The algorithm generally takes <30 iterations to converge.

Clustering

Figure 4: Clustering results on normally-distributed datasets. Columns from left to right are before clustering, after clustering with em_gmm, and original clustering in scikitlearn.

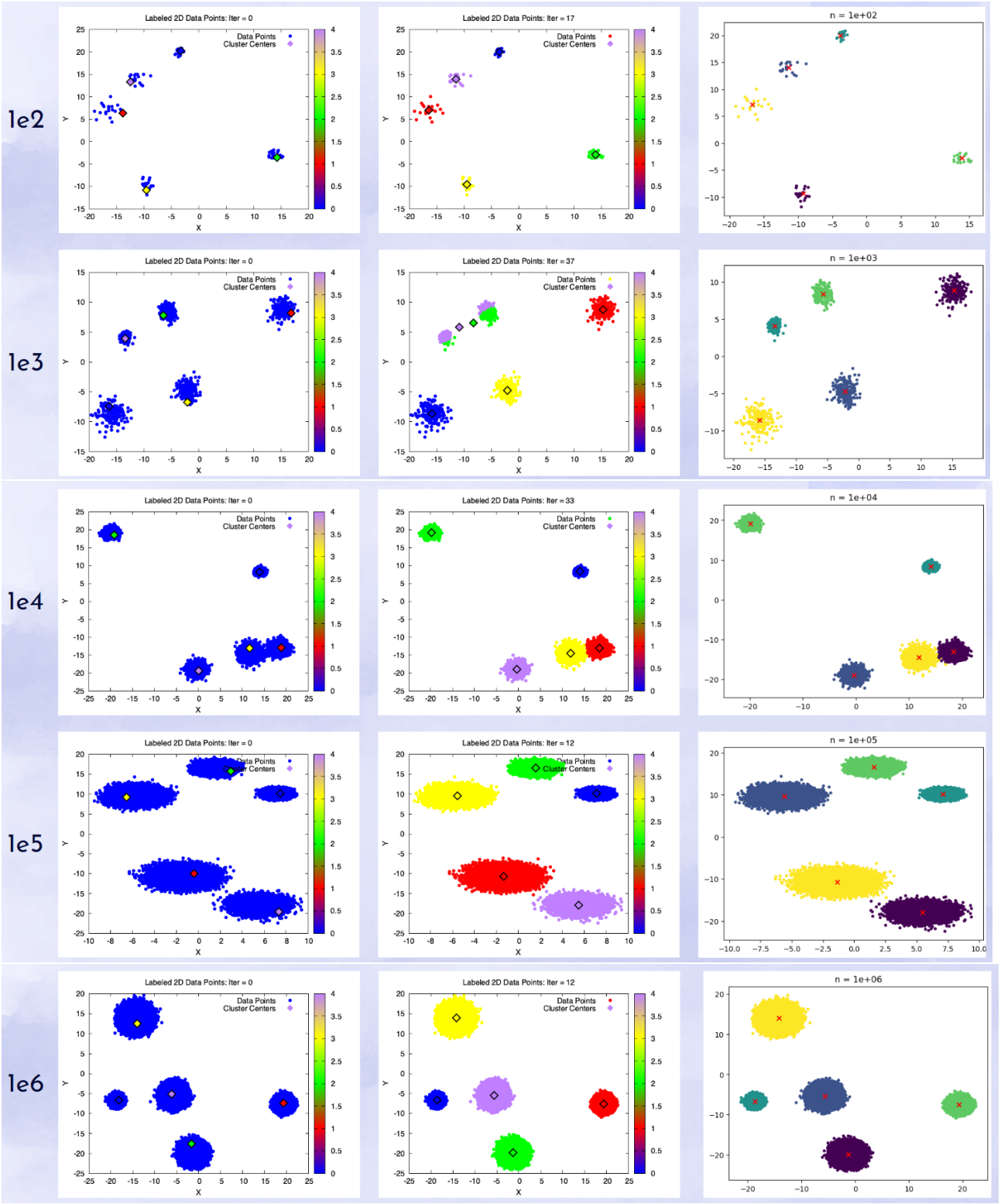
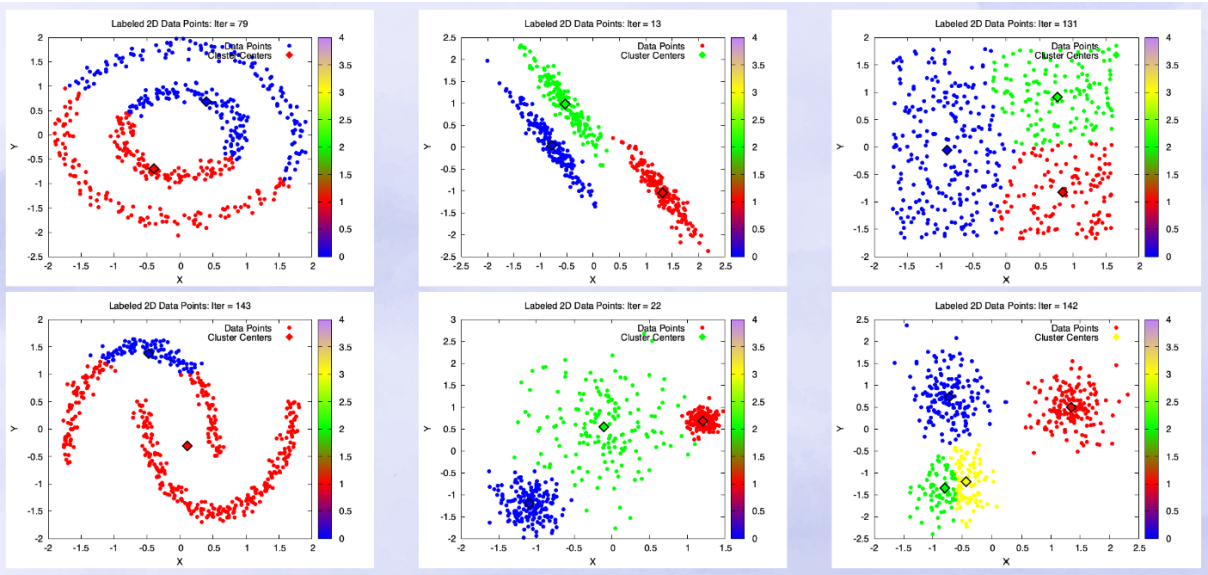


Figure 5: Clustering results for irregular datasets.



The clustering performance visually is spot on with normally-generated data, as shown in figure 4. The only dataset it failed on was of size 10^3, where two close clusters had mixed labels. In figure 5, we show that it performed well on oval datasets as well as ones with noise and higher variance. It performed poorly on the circles and curves, but this is to be expected since they are not normally distributed.

Mean Accuracy

The accuracy of the method is generally pretty good. The means are always within one value of the true mean the dataset was generated with. The method will never be perfect because the method is using local optima, and thus can get caught at the wrong extrema. As well, it is estimating parameters and features entirely from the data. Visually, the placement of the mean parameter at the final timestep, as shown in the previous clustering section, verifies that the means end up in locations that are reasonable.

Timing

Figure 6: Table of number of processors (#p), timing (t), speedup (S), and efficiency (E) for all datasets.

data_2.csv

#p	t OMP	t MPI	S OMP	S MPI	E OMP	E MPI
1	0.019867	0.017685	NA	NA	NA	NA
2	0.012031	0.011218	1.65	1.57	.82	.78
4	0.007751	0.007638	2.56	2.31	.64	.57
8	0.006462	0.006192	3.07	2.85	.38	.35
16	0.015974	0.039434	1.24	.44	.07	.02
32	0.040286	0.058568	.49	.30	.01	0

data_3.csv

#p	t OMP	t MPI	S OMP	S MPI	E OMP	E MPI
1	0.142511	0.135323	NA	NA	NA	NA
2	0.075747	0.074896	1.88	1.80	.94	.90
4	0.043098	0.042657	3.30	3.17	.82	.79
8	0.025500	0.027183	5.58	4.97	.69	.62
16	0.017187	0.035414	8.29	3.82	.51	.23
32	0.048593	0.095992	2.93	1.40	.09	.04

data_4.csv

#p	t OMP	t MPI	S OMP	S MPI	E OMP	E MPI
1	2.525904	2.394892	NA	NA	NA	NA
2	1.347715	1.287682	1.87	1.85	.93	.92
4	0.691616	0.676991	3.65	3.53	.91	.88
8	0.384936	0.407189	6.56	5.88	.82	.73
16	0.220709	0.289407	11.44	8.27	.71	.51
32	0.345432	0.914516	7.31	2.61	.22	.08

data_5.csv

#p	t OMP	t MPI	S OMP	S MPI	E OMP	E MPI
1	11.097214	10.620924	NA	NA	NA	NA
2	5.629918	5.810275	1.97	1.82	.98	.91
4	3.059698	3.106804	3.62	3.41	.90	.85
8	1.711860	1.915580	6.48	5.54	.81	.69
16	0.951879	1.311321	11.65	8.09	.72	.50
32	1.108001	2.358589	10.01	4.50	.31	.14

data_6.csv

#p	t OMP	t MPI	S OMP	S MPI	E OMP	E MPI
1	111.193706	106.226970	NA	NA	NA	NA
2	58.988837	58.026323	1.88	1.83	.94	.91
4	30.565575	31.126690	3.63	3.41	.90	.85
8	16.993853	19.151262	6.54	5.54	.81	.69
16	9.588141	13.081638	11.59	8.12	.72	.50
32	10.496749	22.673569	10.59	4.68	.33	.14

Figure 7: Bar chart comparing timing of OpenMP and MPI implementations.

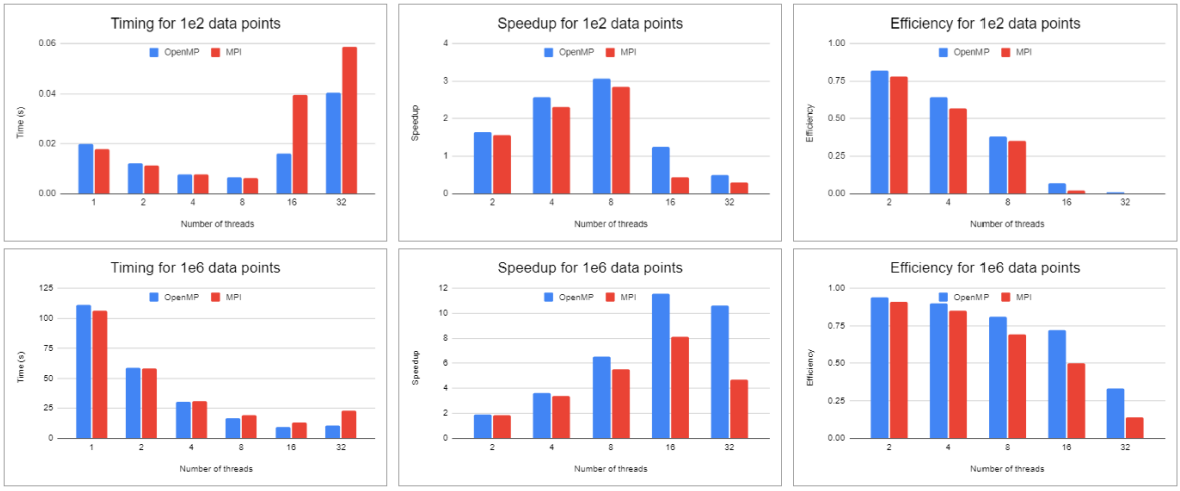
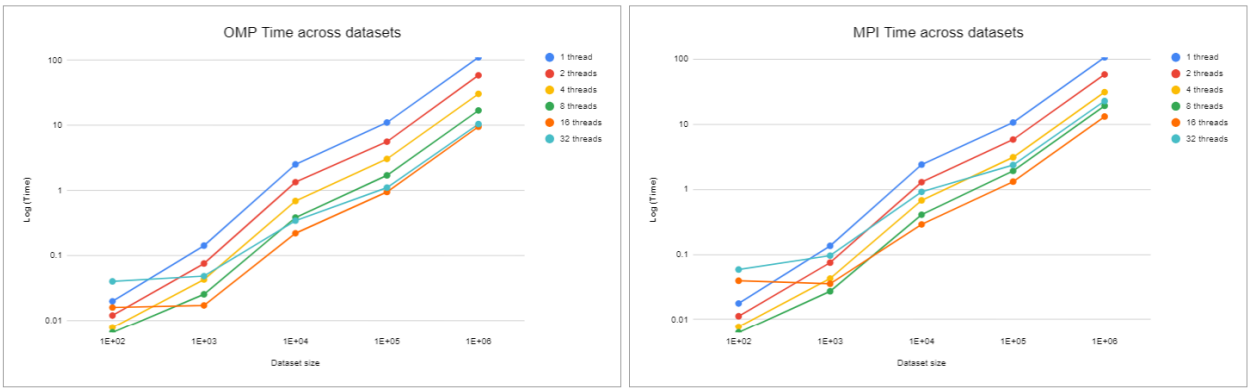


Figure 8: Plot of log of Walltime versus Array Size.



Timing was done with the functions `omp_get_wtime` for the OpenMP implementation and `MPI_Wtime` for the MPI one. For both implementations, the speedup is significant and increases with the thread number up to 16 threads, and then drops at 32 threads, likely because Tuckoo throws a warning that their are not enough slots to run 32 threads and the `--oversubscribe` option is required when running MPI, shown in figure 8. The efficiency is very high with two threads, and then decreases with number of processors. OpenMP always has better speedup and efficiency by varying amounts, shown in figure 7. Figure 6 shows us that the highest speedup (11.65) is OpenMP at 16 processors and an array of size 10^5 and the highest efficiency (.98) is OpenMP at 2 processors and an array of size 10^5. The lowest speedup (.3) is MPI at 32 processors and an array of size 10^2 and the lowest efficiency (0) is also MPI at 2 processors and an array of size 10^2. MPI might have higher overhead due to the large amount of sending and receiving required in the implementation. They both perform quite well at high array sizes.

Acknowledgements

We would like to thank Professor Miguel Dumett for guiding us. Thank you to Sergio Colis Chavez for helping to debug the `MPI_Send` and `MPI_Recv` logic.

Code

The code can be found in the holzer/EMAlgorithm directory on tuckoo, as well as on GitHub at https://github.com/zholzer/GMM_EM_Algorithm. All instructions on how to compile and run the code and view output files are found in the README.md file.

Division of Work

We feel the division of work was fair and played to our backgrounds. In no specific order, we list our main contributions. We contributed equally to the results, report, and slides.

Christine Cho: Dataset generation in `scikitlearn`, initialization functions (`findMeanVector`, `initializeMeans`, `computeDistanceSquared`, `initializeMeansKMeansPlusPlus`, `initializeCovarianc` `initializeCoefficients`), OpenMP `EStep` (`find_determinant`, `gaussJordan`, `pdf`), OpenMP parallelization of the E-Step, clustering results (`plotPoints`)

Zoe Holzer: main file reading and writing (`getLabels`), `MStep`, (`MStepOMP`, `MStepMPI`), `checkConvergence`, timing, MPI parallelization of the E-Step, MPI parallelization, organizing the functions into header files and documentation on how to compile and run, PBS batch file

References

[1] Gibbons, T.J., Pierce, G., Worden, K. et al.. "A Gaussian mixture model for automated corrosion detection in remanufacturing." In: Advances in Manufacturing Technology XXXII. 16th International Conference on Manufacturing Research ICMR 2018, 11-13 Sep 2018, University of Skövde, Sweden. Advances in Transdisciplinary Engineering, 8 . IOS Press . ISBN 978-1-61499-901-0 https://doi.org/10.3233/978-1-61499-902-7-63

[2] Mike, Obrizan, Vladimir. "Parallelization: pthreads or OpenMP?" answers. Web. 1 May. 2024. https://stackoverflow.com/questions/935467/parallelization-pthreads-or-openmp

[3] N. S. L. P. Kumar, S. Satoor and I. Buck. "Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA," 2009 11th IEEE International Conference on High Performance Computing and Communications, Seoul, Korea (South), 2009, pp. 103-109, doi: 10.1109/HPCC.2009.45.

[4] Sridharan, Ramesh. "Gaussian mixture models and the EM algorithm." Web. 1 May. 2024. https://people.csail.mit.edu/rameshvs/content/gmm-em.pdf

[5] Wikipedia contributors. "EM algorithm and GMM model." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 21 Jul. 2022. Web. 1 May. 2024.

[6] W. Kwedlo. "A Parallel EM Algorithm for Gaussian Mixture Models Implemented on a NUMA System Using OpenMP," 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turin, Italy, 2014, pp. 292-298, doi: 10.1109/PDP.2014.77.