Parallelization of the Expectation Maximization Algorithm

First Degree Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Manesh Narayan K
ID No. 2013B1A70542P

Under the supervision of:

Prof. Sundar S Balasubramaniam

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
January 2018

Declaration of Authorship

I, Manesh Narayan K, declare that this First Degree Thesis titled, 'Parallelization of the Expectation Maximization Algorithm' and the work presented in it are my own. I confirm that:

- This work was done mainly as a part of the requirements for the course BITS F421T Thesis.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: _____

Date: _____

Certificate

This is to certify that the thesis entitled, "*Parallelization of the Expectation Maximization Algorithm*" and submitted by Manesh Narayan K ID No. 2013B1A70542P in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor
Prof. Sundar S Balasubramaniam
Professor,
BITS-Pilani Pilani Campus
Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Parallelization of the Expectation Maximization Algorithm

The Expectation Maximization algorithm is an iterative algorithm for estimating the maximum likelihood for model parameters when the data is incomplete, or latent variables are present. As a clustering algorithm, it is widely used for machine learning, computer vision, image reconstruction etc, and hence attempts have been made to parallelize it prior to this, largely limited to the shared memory multiprocessing paradigm and for specific purposes. This work focuses on the implementation of a fast and robust general purpose Hybrid system, relying on both distributed and shared memory multiprocessing environments, formulated in C++ language using the Open MPI and OpenMP libraries to access the two parallel processing paradigms. Cluster based computing is achieved using minor reformulation of the algorithm to use a MapReduce like framework provided by the Open MPI library. A linear speedup is observed in the execution times with increase in number of nodes in the cluster and a sligtly sub linear speedup is observed with the increase in number of threads per process.

Acknowledgements

For a piece of work as extensive as a thesis, input from a lot of people in a myriad of ways is required, without which the completion of the thesis in the accorded time would be a far more challenging task. It would be remiss not to take a moment to thank them for their help and guidance.

I would like to thank my thesis supervisor, Prof. Sundar S Balasubramaniam of Department of Computer Science and Information Systems at BITS Pilani. This opportunity to work on a topic which I found very interesting was provided to me by him. I am also thankful to him for providing me with his guidance, insights, time and resources as and when required by me, while providing me with the freedom to work at my own pace.

Mr. Saiyedul Islam of the department of CS and IS has provided me with the guidance and insights stemming from his experience, which helped me correct a few inaccuracies in my work and cut down on development time. For this I am thankful to him.

I would also like to thank Mr. Sailesh, Mr. Ajay and the department of CS and IS for providing me access to the computing facilities for the testing of my work.

I would like to express my gratitude to BITS Pilani for providing students with good facilities and the freedom to use them for their academic pursuits.

Lastly I would like to thank my parents for their constant support and encouragement during my time here at BITS Pilani.

Contents

List of Figures

Chapter 1

Introduction

The Expectation-Maximization algorithm is an algorithm used to find the maximum likelihood estimates for the parameters of the model as predicted from the data, when the dataset is incomplete, has missing data points[1] or has some latent variables. The EM algorithm functions as a clustering algorithm. The algorithm uses an iterative approach to estimate the maximum likelihood function. Maximum likelihood estimation can find the best fit model for a set of data; however it fails for incomplete data sets. The EM algorithm is much better suited in this case. It starts by guessing some random values for the missing data, uses it to calculate a second set of values. This set of values is then used to make a better guess at the first set of values and so on till convergence.

Many applications for the EM algorithm have been established including pagerank computations[2], anomaly detection[3] and probabilistic PCA[4]. In this work we focus on the utility of the EM algorithm as a simple clustering algorithm.

The EM algorithm is a very slow algorithm. The number of iterations taken to converge can be very high and varies wildly with the number of data points, dimensionality and initial guesses. Each iteration by itself is very time consuming due to the high number of computations. Furthermore, extremely large datasets may take up a large amount of space due to the numerous storage structures required to store intermediate values.

As a result, we aim to test if this algorithm is fairly well suited to leverage the benefits offered by parallelization.

In today's day and age, the availability of data and the complexity of the problem to be analyzed is ever increasing. As a result, the memory and runtime requirements are skyrocketing. Given sufficient time the algorithm will converge; however the sheer amount of time taken is prohibitive in nature. Due to physical and operational constraints, a single core can only be pushed upto a certain frequency of operation[5]. Any speedup beyond this point requires some degree of

parallel computation. With the increasing stress on Big Data, parallel processing has become more important than ever before, to churn out results in time frames wherein the results would still remain relevant.

Broadly speaking there are two types of parallel architectures, shared memory model and distributed memory model[6]. A shared memory model would typically have multiple cores with the address space being shared by all the cores. A shared memory model may be further subclassed based on whether all cores have equal access times to all parts of the address space or not. A distributed memory model consists of multiple processors with each of them possessing a separate address space and being loosely coupled. These systems are also called as clusters.

In this work we aim to provide an implementation of the Expectation Maximization algorithm which is capable of leveraging the hybrid model of shared and distributed memory processing. This will allow for faster convergence owing to the parallel processing employed.

Chapter 2

The EM Algorithm

2.1 Introduction to the EM algorithm

The Expectation Maximization algorithm is a statistics based alogrithm which is used to estimate variance components for the data present in a dataset which can be assumed to belong to a mixed model.

The simplistic description for a mixed model is that it is an experiment where the different observations belong to different distributions or groups. The observations are influenced by both fixed effects and random effects. Fixed effects can be described as the influence of the group to which an observation belongs on the observation itself[7]. Belonging to a group implies that the observation must fall within acceptable limits based on the parameters of the group i.e. mean and covariance or atleast more so than from those of any other group . Random effects are the variations occuring in the observations due to randomness[7].

If the group or cluster parameters are known, we can identify which observations belong to which cluster based on where the probability of a given observation belonging to a cluster is highest, using Bayes rule. If instead it is known which cluster an observation belongs to, the cluster mean and covariance can be computed using the formulas for mean and covariance. However in the event where the cluster parameters and the cluster identities are unknown, there is no easy way to solve the complicated set of equations. As a result the need for a clustering algorithm like EM is felt.

The EM process can be roughly described as two sets of equations, where each set of equations needs the values computed from the other set to be computed. This interlocking dependency results in possibly unsolvable equations upon substituting one set into the other. Instead the EM algorithm arbitrarily picks values for one of the two sets and use it to estimate the second set, following which the newly computed values from the second set of equations are then used

to estimate the first set[8]. This is done iteratively till the point where the change in cluster parameters between two consecutive iterations is lesser than a threshold value epsilon, at which point they are said to have converged.

All clusters are assumed to be characterized by multivariate normal distributions with cluster means, covariances and prior probabilties of the clusters being the model parameters. The probability density function for all observations is assumed to be represented by a Gaussian Mixture Model over all the cluster normals.

The EM algorithm is a soft assignment clustering where each point has a probability of belonging to each cluster and all these probabilities are computed and accounted for[1], as opposed to a hard assignment clustering algorithm like K-means, where each point can belong to only one cluster. In fact, the K-means algorithm can be thought of as a special case of EM algorithm where the probabilities are thresholded to 0 or 1[9].

The EM algorithm if often used for data clustering for applications in the field of machine learning and computer vision. With the ability to observe latent variables, EM is becoming a useful tool for risk assessment and management of a portfolio. New applications are also being found in the field of medical image reconstruction as well as psychometrics. The potential for the utilization of this algorithm are far and wide ranging.

An often faced problem is that larger data sets take a lot of iterations to compute, resulting in long turnover time. Furthermore, extremely large datasets may take up a large amount of space due to the numerous storage structures required to store intermediate values. Parallelization of the EM algorithm is a step to decrease the time taken to estimate model parameters and also leverage the additional storage resulting from multiple machines being present on the cluster.

2.2 Equations

The equations for the EM algorithm are frequently represented in slightly different ways in different publications. However the end result in all cases is always the same. In this work, the following set of slightly tweaked equations (can also be found in some literature) is used for optimization purposes.

$$f(x_i|c) = \frac{1}{\sqrt{(2\pi)^D * |\sigma_c|}} * e^{(-1/2)*((x_i-\mu_c)^T*(\sigma_c^{-1})*(x_i-\mu_c))} \tag{2.1}$$

Where f is the probability density function at the point x_i attributable to the cluster c. Since covariance and means are not known in the first iteration they are arbitrarily assumed in the first iteration. Often this function is represented as $P(x_i|c)$, however strictly speaking, it is just

a density function and not a probability value. However because it is a density function, we can assume an infinitesimally small interval epsilon centered around x_i. The multiplication of the density function and epsilon gives us the probability of the point x_i given cluster c or $P(x_i|c)$. However this operation is never required as the values of epsilon are cancelled out in subsequent equations. The C++ code written to implement this refers to these density values as $P(xi|c)$.

$$P(C|x_i) = \frac{P(x_i|C)P(C)}{\sum_c P(x_i|C)P(C)} \tag{2.2}$$

$P(C|x_i)$ is the cluster posterior probability, given point x_i. P(C) is the prior probability for the cluster C, which is assumed to be equal for all clusters in the first iteration. $P(x_i|C)$ is used directly from the previous equation. Since both numerator and denominator contain only $P(x_i|C)$ terms and each of those terms contains the epsilon term as the common factor, it can be cancelled out, giving us :

$$P(C|x_i) = \frac{f(x_i|C)P(C)}{\sum_i f(x_i|C)P(C)} \tag{2.3}$$

The goal of maximum likelihood estimation is to choose model parameters such that the likelihood of the datapoints given the parameters is maximized. Since this is hard, the EM approach is used where a guess is taken for the parameters and then cluster posterior probabilities are computed using Bayes theorem.

$$w_{ic} = \frac{P(C|x_i)}{\sum_i P(C|x_i)} \tag{2.4}$$

This notation can be thought of as the weight or importance of a point x_i to the cluster c. Some sources use w_{ic} to denote $P(C|x_i)$ but in this work it is always representing the weight as shown above. Doing this computation at this phase reduces the number of computations to be done downstream.

$$\mu_{c_j} = \sum_i w_{c_i} * x_{i_j} \tag{2.5}$$

Here, the value of the j^{th} dimension of the cluster mean is computed for cluster c.

$$\sigma_{c_{jk}} = \sum_i w_{ic} * (x_{i_j} - \mu_{c_j}) * (x_{i_k} - \mu_{c_k}) \tag{2.6}$$

This equation computes the covariance between two attributes or dimensions j and k for a given cluster c.

$$P(C) = \frac{\sum_i P(C|x_i)}{I} \tag{2.7}$$

This equation computes the prior probabilities for the clusters for the next iteration. There is a mathematical guarantee that the priors of all clusters add up to 1, which allows us to allot equal probability of the value 1/C. where C is the number of clusters, to all clusters for the first iteration.

$$\sum_c ||\mu_i^t - \mu_i^{t-1}|| \leq \epsilon \tag{2.8}$$

This is used to check for convergence. For each cluster, the euclidean distance between the cluster centroids between the current and previous iterations is computed and if the net change across all clusters is less than the predefined ϵ, the model is said to have converged.

The equations 1 and 2 represent the E or expectation phase of the algorithm while the subsequent equations represent the M or Maximization phase of the algorithm.

2.3 Things to remember

One thing to keep in mind is that the EM algorithm is susceptible to local optima due to its design and as such may require multiple runs with new initializations to identify the global optimum. The runtime of the algorithm in terms of number of iterations is also heavily dependent on the initial model parameters assumed, as well as the number of clusters assumed[10].

There are several ways to estimate the number of clusters. However the focus of this work lies on the parallelization of the existing EM algorithm and hence estimation of number of clusters is beyond the scope of this work.

2.4 Pseudo-code

The pseudo code for the EM algorithm is as follows:

EM algorithm

Dataset D, number of points I, double ϵ, GMM $C = \{C_1,..,C_k\}$

//initialization

For all k cluster centroids, initialize model C with

μ_c = rand(D), σ_c = identity matrix, P(C) = 1/k;

//Computations

while *not converged* **do**

  //E-step : estimate likelihood

  foreach *cluster* $c \in C$ **do**

    foreach *point* $i \in I$ **do**

      Compute P(x_i|c) and P(c|x_i)

    end

  end

  //M-step : update model

  foreach *cluster* $c \in C$ **do**

    foreach *point* $i \in I$ **do**

      Compute partial means

    end

  end

  Compute priors

  foreach *cluster* $c \in C$ **do**

    foreach *point* $i \in I$ **do**

      Compute partial covariances

    end

  end

  Check for convergence

end

Algorithm 1: EM algorithm

2.5 Alternative Implementation - Immediate EM

The convergence of the EM algorithm can be accelerated if instead of a centralized maximization step, the model parameters are updated immediately each time a point changes its cluster association. The E and M steps are combined at all points of time. Since this work did not adapt this variation, we would not be going into much depth about the algorithm except to discuss the basics and its unsuitability for adaptation.

The k-means algorithm was first implemented in an immediate model update fashion[11][12]. The immediate-EM model was described later based upon this earlier work[13]. The Immediate-EM converges as soon as all n points of the data sets have been processed. The set of equations from the classical EM algorithm can be rewritten to enable fast model updates for immediate-EM.

However immediate-EM comes with a lot of associated communication and synchronization overheads since model updation is done after each point and this needs to be communicated to all nodes containing the model. So for larger data sets, the communication overheads will occupy the majority of runtime. Furthermore, this model of operation necessitates that only one point be processed at a time, which restricts the scope of parallelism, allowing only parallelization within the computations specific to one point.

Hence this variation was found unsuitable for this work.

Chapter 3

Design and Implementation of a Parallel EM Algorithm

3.1 Equation Analysis

Partitioning The first step is to identify the tasks that can be executed concurrently. Since each of the 5 equations are dependent on the results from the preceding equations, we cannot execute the equations in parallel, ruling out function decomposition. However since the computation of a given equation for one point is independent of the computation of the same equation for another point, we can do a given computation for all points in parallel. Hence data decomposition is the primary level of parallelism, implemented using the Open MPI library. The points are distributed almost equally to all the nodes in the cluster in the beginning and then the computations proceed independently till communication is needed for model updates.

At the secondary level of parallelism, OpenMP library is used to run multithreaded operations. The parallelism is more fine grained in this case, used to run looping operations in parallel.

Communication

In the initialization and data transfer phase, MPI communication is required thrice, once for passing the observation count, cluster count and dimensionality, once for scattering the points across the cluster and once for passing the initial arbitrary model parameters. During the computation phase, 3 MPI map reductions are needed per iteration, for computing w_{ic}, means for clusters and covariances. Each of these depend on the preceding computations, necessitating communication after each of them to allow the succeeding computations to proceed. All MPI communications are funneled through the master omp thread.
In case of OpenMP, no communication or reduction mechanisms are required. However, for values to be read correctly from shared variables after writes or to prevent concurrent writes,

synchronization is required. Synchronization using barrier or critical constructs are expensive operations. However, a lot of the need for synchronization can be eliminated by restructuring the code so that changes requiring barriers or flush to be synchronized precede a construct which provides an implicit barrier or flush, like an *omp for loop*. Furthermore in some cases, need for critical sections to access shared regions can be eliminated by reordering the loops or changing the loop which runs in parallel if one of them iterates over the clusters and the other one over all points.

3.2 Parallel Pseudocode

Parallel EM algorithm

Dataset D, number of points I, double ϵ, GMM $C = \{C_1,..,C_k\}$

//initialization

For all k cluster centroids, initialize model C with

μ_c = rand(D), σ_c = identity matrix, P(C) = 1/k;

Distribute I points to n machines with I' points each

//Computations

while *not converged* **do**

  //E-step : estimate likelihood

  foreach *cluster* $c \in C$ **do**

    foreach *point* $i \in I'$ **do**

      Compute P(x_i|c) and P(c|x_i)

    end

  end

  Communicate - $\sum_i P(c|x_i)$ (Reduction)

  //M-step : update model

  foreach *cluster* $c \in C$ **do**

    foreach *point* $i \in I'$ **do**

      Compute partial means from I' points

    end

  end

  Communicate - compute mean (Reduction)

  Compute priors

  foreach *cluster* $c \in C$ **do**

    foreach *point* $i \in I'$ **do**

      Compute partial covariances from I' points

    end

  end

  Communicate - compute covariance (Reduction)

  Check for convergence

end

Algorithm 2: Parallel EM algorithm

3.3 Implementation Details

Distributed memory For distributed memory computing, at the beginning the points are distributed uniformly to all the MPI processes by the master process. After this point, 3 communications are required per iteration to do reductions over posterior probability, means

and covariances. Since each of these computations depends on the upstream values previously computed, the reductions need to be done separately. This is a fairly coarse grained level of parallelism because each process functions independently for its set of points apart from the time of communication.

Shared memory For multi core computing, the parallelization is focused in the for loops. Each of the *foreach* loops operating over the points i are parallelized using omp threads. Parallelization was also attempted over the outer *foreach* loops operating over the clusters, but the number of clusters is usually very low, thereby acting as the bottleneck in the parallelization; hence it was abandoned. Furthermore, parallelizing along the number of points allows us to avoid some critical accesses. The outer loop runs across all clusters and the inner loop runs across all points, so as to utilize the locality of data and prevent frequent misses as the data for all points within the same cluster would be stored together. All threads are initialized at the beginning of execution and retained for the runtime and all the for loops are parallelized using these threads. Since the majority of computations are spread over all datapoints, this gives us a fair level of parallelism.

Hybrid system The hybrid system at a high level, is a simple amalgamation of both of the above mentioned systems. They both work at a different level of parallelization. However putting the 2 systems together requires some additional synchronization so as to funnel the MPI communications through a single omp process and can cause some additional delays.

3.4 Program structure

The code for the parallelized EM algorithm is sectioned into 5 different files, for more convenient development. The purposes of the files are as follows:

main.cpp
The initialization of the MPI environment and thread support check are carried out in this file, following which it invokes the EM algorithm. Post the algorithm execution, it also finalizes the execution and facilitates exit.

mat.h and *mat.cpp*
This user defined file provides access to the class *Matrix* which contains functions for computing common matrix operations as well as functions for enabling the allocation of multiple matrices in a contiguous memory location so as to allow for fast reduce operations. The decision to implement a class instead of using a pre existing library was made so as to explore options for parallelism within matrix computations. Part of the code for this was inspired from a report titled *A parallel implementation of Expectation Maximization algorithm* found on the website

of Brigham Young University. However the report is dated 1993 and the code appended was found to be dysfunctional due to library, compiler and technology changes. Furthermore the basic model was also a two level mixed effects model, preventing an effective porting of the code to test and compare results. So though the code borrowed inspiration from this existing work, most of the code had to be rewritten to meet the requirements of this work and environment.

EM.h and *EM.cpp*
This user defined header contains the class *Expectation Maximization*. This class is responsible for doing all the computations for the E and M phase of the algorithm and checking for convergence. The parallel operations start after the distribution of data points from master node. This class provides both the process level (Open MPI) as well as thread level (OpenMP) parallelism. This class handles the parallelism as well as communication operations. The different member functions of this class are of 3 types; 1 for reading input and distributing data, 1 for printing the final output and the others for doing the 5 sets of computations.

3.5 Code Compilation and Execution

The code is well commented to permit easy understand-ability and maintenance. For details regarding input format and code compilation and execution, refer the Readme file.

Chapter 4

Experiments

4.1 Experimental Setup

4.1.1 Cluster setup

A cluster of 16 nodes was setup over LAN to run all the experiments. Each of the 16 nodes comprised of a desktop computer running on an Intel i5-4570 processor. These are quad-core processors with hyperthreading disabled in them by the OEM, resulting in 4 physical and 4 logical cores. Each machine has 16 GB of RAM and runs Ubuntu 16.04 LTS operating system with g++ version 5.4.1. The PureOMP implementation for running only on shared memory system was also tested on a 48 core machine containing Xeon processors and 196 GB of RAM.

4.1.2 Dataset

Due to a lack of well curated and sufficiently large dataset to run experiments on, a synthetic dataset was used to test the code. The dataset is 2-dimensional with points sourced from 31 normal distributions. This gives rise to 31 clusters. The dataset comprises of 2 million points. To test scaling with number of datapoints, similar datasets of sizes upto 20 million were used.

4.2 Procedure

A variety of experiments were run to identify the scaling potential of the hybrid implementation of EM algorithm which are listed as under:

- Scaling of the algorithm with an increase in number of nodes (Open MPI) was tested first without multithreading

- Scaling of the algorithm with an increase in number of threads (OpenMP) was tested without multiprocessing
- Scaling of the algorithm with an increase in both threads and processes was tested
- Scaling of the algorithm with an increase in number of datapoints was tested
- Efficiency of OpenMP vs Open MPI was tested by running multiple OMP threads and then multiple MPI processes on a single machine

Since there is a limitation on the number of machines available, 16 machines were set up in a cluster. However upto 64 MPI processes were used in testing by running upto 4 MPI processes per machine, as each machine has 4 logical cores. Furthermore due to the number of cores being limited to 4, the number of threads were kept at a maximum of 4. The PureOMP implementation was tested with upto 64 threads.

As stated previously, the number of iterations till convergence for the EM algorithm varies widely based on the initial cluster parameters assigned[10]. Hence for the individual runs to be comparable, the initial assignments for cluster means were kept constant. The cluster covariance matrices were initialized as identity matrices and all clusters were assumed to be equally probable to assign priors.

The aim of checking scaling with the number of datapoints is to see if the code can keep up with the increased number of computations per iteration. However since EM's iterations to converge varies with data points, adding additional points will change the number of iterations as well, making the results incomparable. Hence the original dataset of 2 million points was duplicated to produce 4 million and so on upto 20 million points. The equations behind the algorithm are all weighted and since all the inputs are being duplicated equal number of times, all computations will produce the same values as with the original dataset and hence the iterations till convergence will remain the same, with just an increase in the number of computations due to an increase in points. This has been verified mathematically and experimentally.

The EM algorithm is quite slow to converge for large datasets and for the dataset with 2 million points, it took approximately 17 hours with 1381 iterations. This makes it difficult to conduct all experiments as there are issues with system availability in the lab, lab working hours and so on. As a workaround, the time taken to execute a sufficiently large number of iterations was instead measured to get results. The reasoning is that the number of computations per iteration remains the same, causing the execution time of one iteration to be roughly the same through multiple runs in the same conditions. The slight differences in execution time of one iteration can be put down to scheduling and runtime variations and other such non code related overheads. These

can be averaged out given a sufficient number of iterations. This was also tested experimentally and found to be sound.

Chapter 5

Results and Discussion

5.1 Interpretation of results

The efficiency of parallelization of an algorithm is generally measured in terms of speedup. Speedup can be defined as the ratio of turnaround time of two variants of the algorithm, one usually without parallelization and the other with some degree of parallelization. In a distributed system, there is no concept of global time. Instead, the time taken from the start of execution of the algorithm till the end of the algorithm as seen at the master node is used to measure execution time.

$$Speedup = Execution\ time\ in\ sequential\ run / Execution\ time\ in\ parallel\ run \tag{5.1}$$

The execution time measured is the time taken for execution of the algorithm and does not include the initial input of points and parameters of the dataset.

5.2 Performance of EM on distributed memory system

This section details the results observed for speedup with increase in number of MPI processes used for execution.

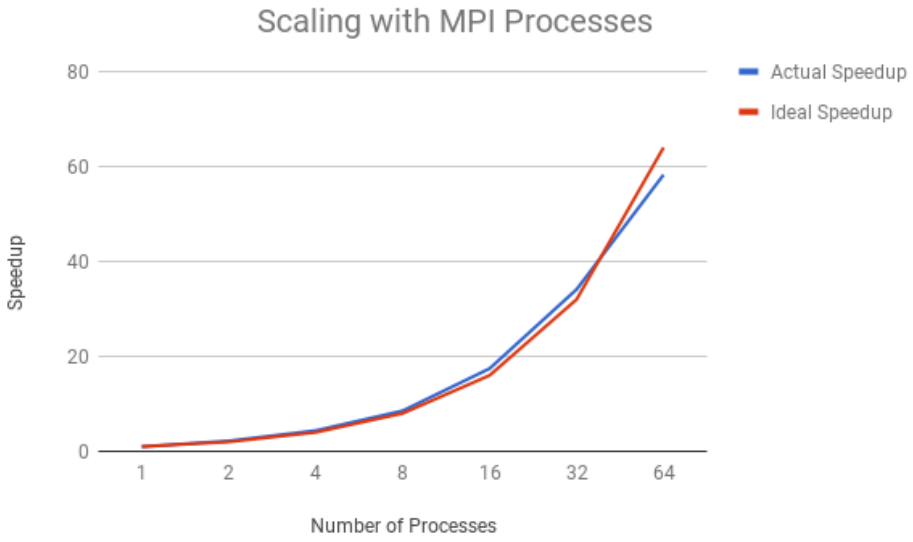

FIGURE 5.1: Performance on a distributed memory system

The cluster contains 16 quad-core nodes. After 16 MPI processes, the number of processes per node starts exceeding 1. Since the systems are quad-core, upto 4 processes per node are used, letting us test with upto 64 processes.

A near linear speedup is observed upto 8 nodes, which tells us that the algorithm is well parallelized. The minor differences are due to communication overheads.

In the cases with 16 and 32 processes, a slightly super linear speedup is observed. This can be attributed to the cache effect. With more nodes in play, the accumulated cache size also increases, allowing bigger working sets to fit in cache memory and dramatically reducing memory access times[14].

In the case of 64 processes however, since atleast one core would be running system and background processes, we can't expect perfect speedup. As expected, a slightly below ideal speedup is observed at that point.

5.3 Performance of EM on shared memory system

This section details the results observed for speedup with increase in number of OpenMP threads used for execution.

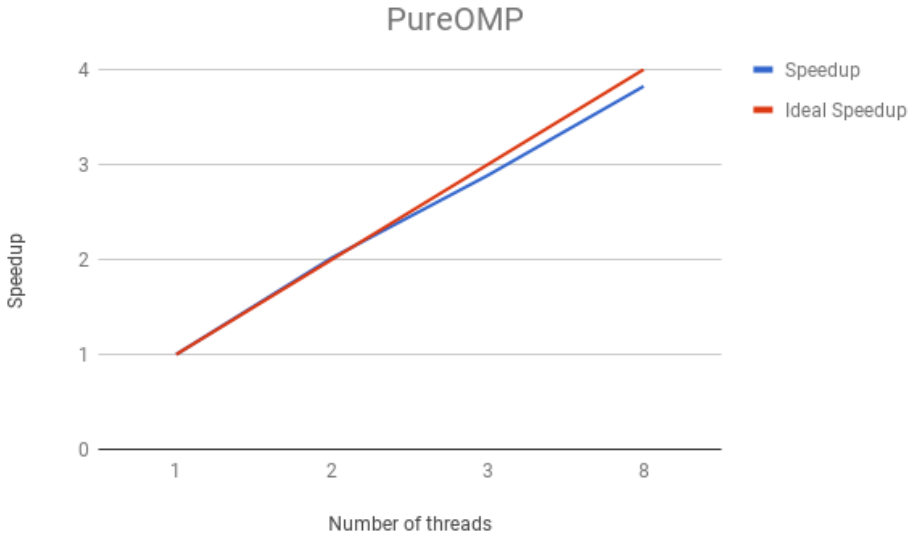

FIGURE 5.2: Performance of EM implementation on a shared memory system

To test the scaling with increase in OpenMP threads, a single node is used to run the algorithm. Since the machines have 4 logical cores, we can test only upto 4 threads. In the ideal scenario, the speedup should be linear with increase in the number of threads. However, since one core will be running system and background processes, the utilization of 4 threads is expected to give sub par performance, as can be observed.

In a shared memory setting, delays resulting from synchronization, implicit and explicit barriers, atomic updates and access to critical sections add up. Also, the barriers put up to allow MPI calls funneled through the master are still in place, as otherwise the code won't be standardised. This may result in further delays. Though code restructuring can eliminate some synchronization requirements, a lot of them can't be removed using *nowait* clause, as the next set of computations would need updated values.

As can be seen from the results, all of this results in a sub linear performance.

To eliminate these delays caused in the program by the hybrid paradigm, the code was changed to remove all the MPI functions and a pure OMP implementation was created and it was tested on a 48 core machine.

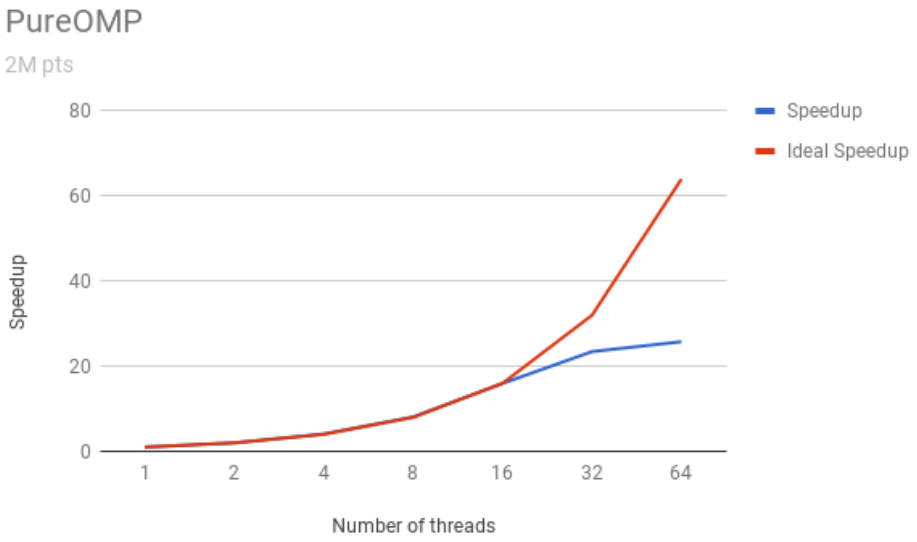

FIGURE 5.3: Performance of a pure OMP implementation with 2 million points

This implementation is showing good scaling upto 32 threads. However beyond that, the performance is showing degradation. When the problem size is kept fixed and the number of threads is increased, the amount of work done per thread decreases. As a result, the thread overheads become a bigger fraction of the total amount of CPU time used by each thread, and the benefits of parallelization are reduced[15].

To test this explanation, the program was run with 20 million points. The results are as follows.

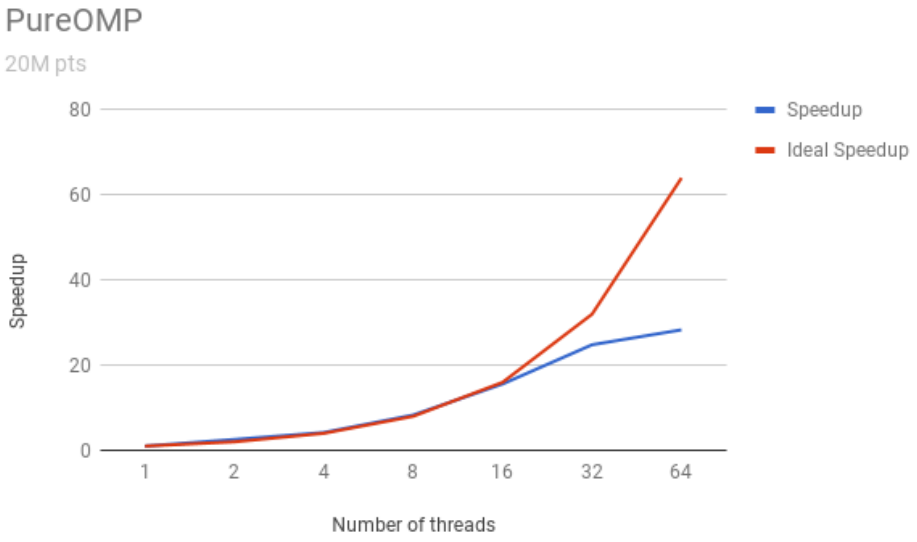

FIGURE 5.4: Performance of a pure OMP implementation with 20 million points

The performance shows a marked improvement in speedup for both 32 and 64 threads with 20 million points. Given a sufficiently large dataset, the speedup can be a lot better than that

observed with 2 million points. However due to time constraints (20 million points need around 2.5 hours for a single run consisting of 20 iterations), larger datasets of the scale of 200 million points couldn't be tested.

However the observation is that even a pure OMP implementation shows good scaling dependent on the dataset size.

5.4 Comparison of performance on distributed vs shared memory systems

This section is aimed at comparing the speedups achieved by running multiple MPI processes on one machine in comparison to multiple OMP threads on one machine.

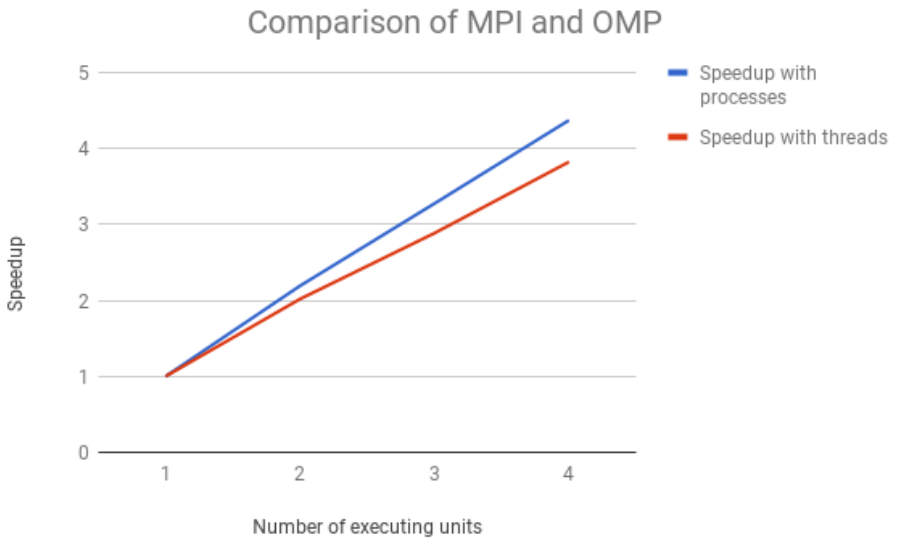

FIGURE 5.5: Comparison of performance on distributed vs shared memory systems

MPI provides a much better scaling. This can be attributed to the fact that each MPI process runs in its separate address space, eliminating the need for shared variables and critical accesses. The only time synchronization occurs is to facilitate the reduce operations required for computation. In comparison, in OMP, for instance each time a parallel loop executes or each time a shared variable is updated, an implicit or explicit barrier is required to ensure coherence in the values. Furthermore, due to the structure of the EM algorithm, in some computations, regardless of whether the loop iterating over clusters is parallelized or the loop iterating over the points is parallelized, some variable would always require access through the critical block.

As can be expected the performance of MPI processes is better.

5.5 Performance of EM on a hybrid system

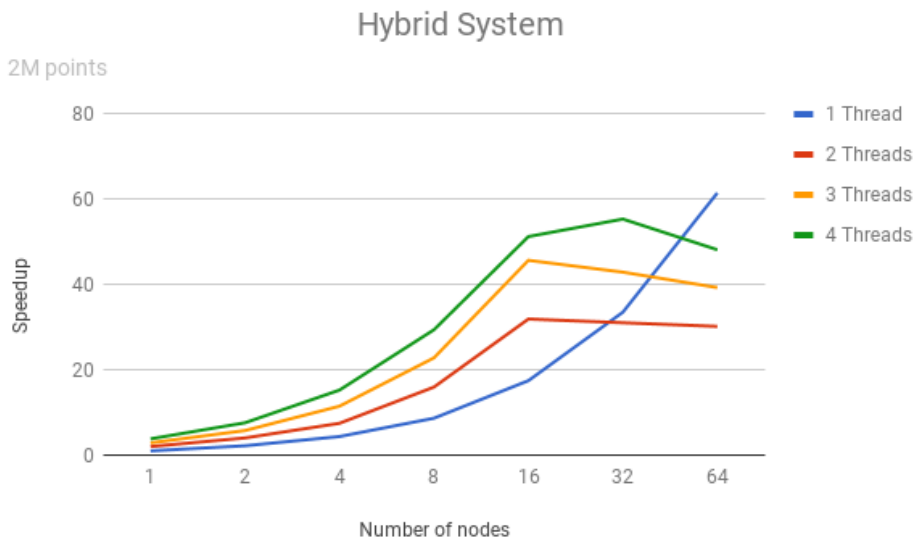

FIGURE 5.6: Performance of Hybrid system for 2 million points

This section deals with the testing of a Hybrid of MPI and OpenMP paradigms. Each of the plotlines show the performance of a certain number of threads with different number of nodes, as seen in 5.6. However inferences about efficiency can't be drawn directly based on the positions of the plotlines as the number of threads is different at each point on the x axis. However given a certain number of MPI processes, the run with a higher thread count would have a smaller run time and conversely would be a more effective implementation. The idea is to check if for given a number of executing units does the speedup compare. Since the entire cluster has 16 machines with 4 cores each, allowing parallelism upto 64 executing units, so we can't expect any speedup beyond that. Though the system scales well, the system isn't as effective at higher number of threads.

As stated in the section 5.3, the overheads are higher in higher levels of parallelism, so to see effective scaling, the problem size should be larger[15]. Hence we check the effectiveness of the system with 20 million points.

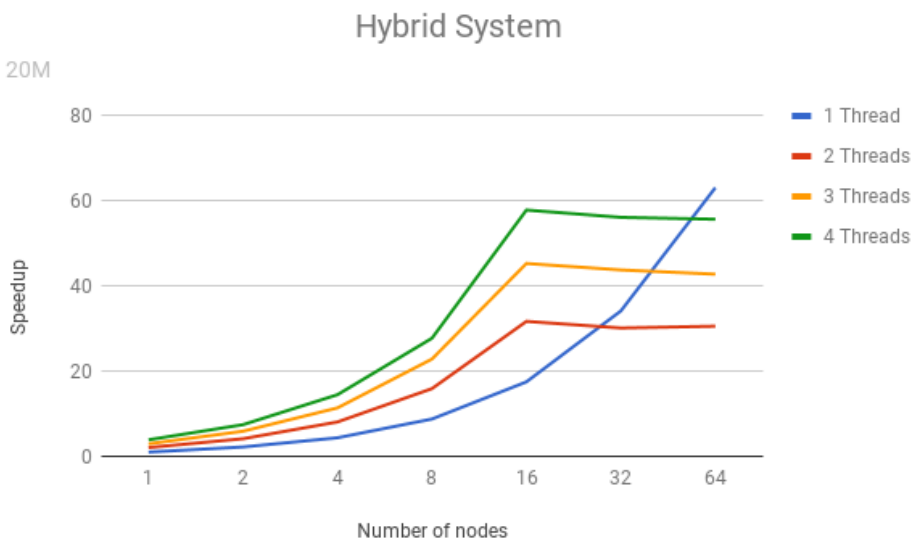

FIGURE 5.7: Performance of Hybrid system for 20 million points

As we can see in 5.7, the performance displayed is much better than that seen earlier. The speedup seen with 16 processes and 4 threads per process is close to the value seen with 64 processes and 1 thread per process, and both are acceptably close to 64.

Hence this Hybrid system shows good scaling.

5.6 Scaling with increase in number of datapoints

As stated earlier in section 4.2, the datasets were designed to keep the number of iterations constant while increasing the number of computations. In an ideal scenario, the execution time taken for these different datasets should increase linearly with increase in number of data points, as the number of computations would also increase linearly with increase in the number of points.

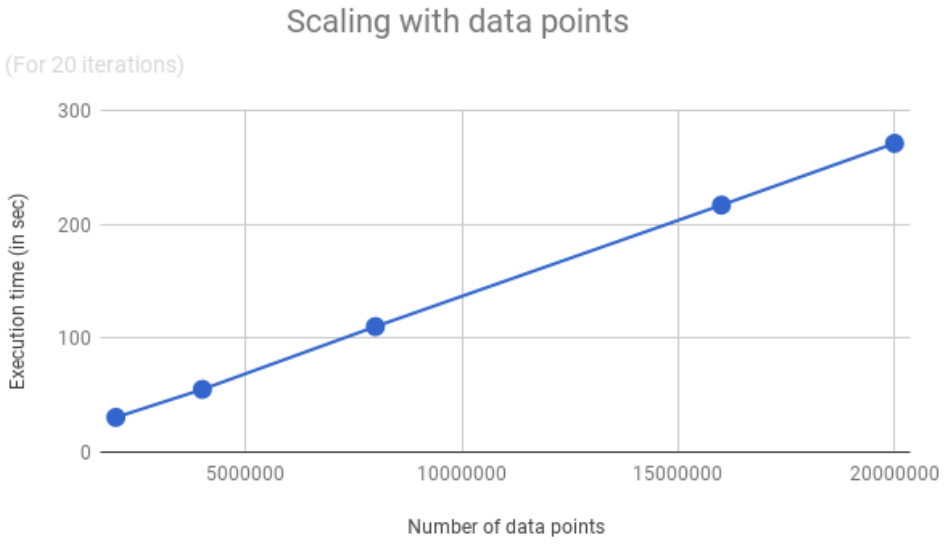

FIGURE 5.8: Scaling with increase in number of datapoints

All the measurements here are done on 16 nodes running 4 processes each.

This graph shows an almost linear increase in execution time, implying that the algorithm is scaling well.

Chapter 6

Conclusion

This project required a thorough study of the working of the Expectation Maximization clustering algorithm, its equations and the issues associated with the algorithm. This implementation required devising and testing multiple strategies for parallelizing segments in slightly different ways to identify the most effective one, which was finally used to run these experiments. Restrictions presented by the testing equipment have handicapped some of the experiments; nevertheless a fair representation of the capabilities of this implementation have been demonstrated.

The MPI part of the implementation shows a much better scaling capability than the OpenMP part in all aspects. A further look can be taken into restructuring the code or using different functions to increase the performance of the OpenMP, although most such possible optimizations are already in place. The restrictions imposed by the OpenMP paradigm will probably result in the pure OpenMP or hybrid implementation never catching upto the performance demonstrated by a pure MPI implementation (with the number of executing units remaining the same). However, as discussed before, with an increase in problem size, the pure OpenMP and hybrid versions show improved scaling ability with number of processing units. Hence these can be effective for sufficiently large problems. Hence while the MPI implementation shows linear scaling, the hybrid implementation shows almost linear scaling with sufficiently large problem sizes and even the PureOMP version shows acceptable scaling for larger datasets. The MPI and hybrid versions are scalable for parallelization purposes.

Care also needs to be exercised while developing a hybrid implementation. Both these paradigms come with their own set of issues and in a hybrid setup, the errors occurring will be harder to identify and debug, such as the ones caused by omp threads running inside an MPI runtime environment.

Bibliography

[1] Arthur P Dempster, Nan M Laird, and Donald B Rubin. "Maximum likelihood from incomplete data via the EM algorithm". In: *Journal of the royal statistical society. Series B (methodological)* (1977), pp. 1–38.

[2] Polyxeni Zacharouli, Michalis Titsias, and Michalis Vazirgiannis. "Web page rank prediction with pca and em clustering". In: *International Workshop on Algorithms and Models for the Web-Graph*. Springer. 2009, pp. 104–115.

[3] Jorge Silva and Rebecca Willett. "Hypergraph-based anomaly detection of high-dimensional co-occurrences". In: *IEEE transactions on pattern analysis and machine intelligence* 31.3 (2009), pp. 563–569.

[4] Sam T Roweis. "EM algorithms for PCA and SPCA". In: *Advances in neural information processing systems*. 1998, pp. 626–632.

[5] Krste Asanovic et al. *The landscape of parallel computing research: A view from berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[6] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*. Vol. 42. John Wiley & Sons, 2005.

[7] Joseph C Gardiner, Zhehui Luo, and Lee Anne Roman. "Fixed effects, random effects and GEE: what are the differences?" In: *Statistics in medicine* 28.2 (2009), pp. 221–239.

[8] Todd K Moon. "The expectation-maximization algorithm". In: *IEEE Signal processing magazine* 13.6 (1996), pp. 47–60.

[9] Mohammed J. Zaki and Jr. Wagner Meira. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014. ISBN: 9780521766333.

[10] CF Jeff Wu. "On the convergence properties of the EM algorithm". In: *The Annals of statistics* (1983), pp. 95–103.

[11] John A Hartigan and Manchek A Wong. "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[12] JA Hartigan. "Clustering algorithms. 1975". In: *John Willey & Sons* (1975).

[13] Claudia Plant and Christian Bohm. "Parallel EM-clustering: Fast convergence by asynchronous model updates". In: *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE. 2010, pp. 178–185.

[14] John L Gustafson. "Fixed time, tiered memory, and superlinear speedup". In: *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*. 1990, pp. 1255–1260.

[15] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.