Parallel Machine Learning Implementations in Scala

An Expectation Maximization Algorithm for

Gaussian Mixture Models

Pierre Grydbeck

Bachelor Semester Project under the supervisions of :
HEATHER MILLER, PROF. MARTIN ODERSKY

ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE (EPFL)
1015 LAUSANNE, SWITZERLAND

Abstract

An increasing number of applications in the industry as well as in research require the analysis of large datasets using machine learning (ML) algorithms. However because of the increasing size of available datasets, the execution time of the applications can become prohibitively long.

Parallel to that, there is a clear trend in the hardware industry of increasing the amount of cores inside CPUs and CPUs inside machines.

The goal of this work is to evaluate whether or not it is possible to take advantage of this trend to speed-up ML applications using Scala's parallel collections and the Menthor framework[3] as it is expected. A widely used ML algorithm, namely the Expectation Maximization algorithm for Gaussian Mixture models was implemented in two different ways: one using the parallel collections and the other with the Menthor framework. The implementations were then carefully benchmarked to assess their performance.

Both implementations perform, as expected, much faster when they run on multi-core machines. For example when running on consumer-grade hardware (Intel Core i7 with four cores that supports hyper-threading) the speedup factor for the parallel collection could reach 3.5 when all cores were used. The Menthor implementation showed good results. Although the running time was usually around 50% longer than with the implementation using the parallel collections it showed very good speedups when multiple cores were available (4.2 times faster with all cores compared to the use of only one). The benchmarks run on a server with four AMD Dual-Core processors also showed good speedups but the results indicated that due to possible hardware architecture limitations the use of all eight cores (four processors) is not always significantly better than the runs using only two of the available cores.

In conclusion, developers should take advantage of this new opportunity to increase the running speed of their applications by taking full advantage of the available hardware while writing very little additional code.

Contents

Code

The code related to this project is available at:
https://github.com/noobii/Expectation-maximization-with-Scala

1 Introduction

1.1 Context

An increasing number of applications require the analysis of large data sets. Social networks analyze user data to carry out targeted advertisements. Online selling platforms use customer's shopping history to refine their offerings. In research topics such as genetics or neurology, the analysis of big data sets have become critical to allow new discoveries. However a problem is persistent. The multiplication of analysis and the increase in size of the available data sets require increasingly long running times. This limitation prevents new applications of being developed or research from being carried out at a fast pace.

Another clear trend in the industry is the multiplication of cores inside CPUs and CPUs inside machines. This trend has become a real opportunity and challenge for developers to take advantage of. The multiplication of cores makes it increasingly difficult to write software that takes advantage of all available hardware while still being maintainable and scalable.

1.2 Scala

To address these challenges new models and tools were introduced early in the Scala libraries. Since their introduction, they have been greatly developed and refined. To this day Scala solves the parallelization challenge with high-level, elegant and easy-to-use libraries such as parallel collections (`scala.collection.parallel`) and the Scala actors (`scala.actors`). Other libraries such as Akka actors and other frameworks that take advantage of these new opportunities have been developed by other parties.

1.2.1 Parallel collections

The Scala parallel collections were introduced in Scala 2.9.0. They are composed of a parallel counterpart to "classical" sequential collections (`scala.collection`). Most corresponding collection classes are available in a parallel version. `List` becomes `ParList`, `Array` becomes `ParArray`, `Map` has `ParMap` etc.

The parallel collections take advantage of Scala's well-designed type hierarchy (figure 1) to have all typical methods available such as `map`, `fold`, `find` etc. The only difference is, of course, that when they are invoked on a parallel object they transparently run in parallel, trying to take advantage of all available hardware.

To make this possible the *parcols* use an underlying divide and conquer approach. [6] Each class uses a splitter method that is recursively applied to divide the collection into smaller parts that can be processed separately. After that each piece has been processed, they are recombined using a combiner function that also must be implemented. The underlying implementation uses Java's Fork/Join framework for efficient load balancing. The developer that wants to write parallel applications

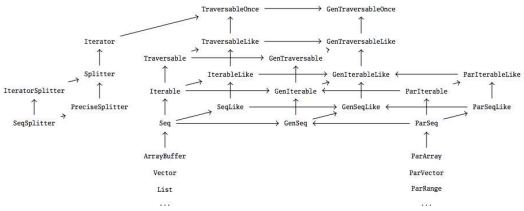

Figure 1: Scala's collections hierarchy (http://docs.scala-lang.org/overviews/parallel-collections/architecture.html)

does not have to explicitly do anything but to use the parallel collection or convert the sequential objects using `.par`.

1.2.2 Actors

Scala's main concurrency construct for writing concurrent code is actors[4]. Unlike traditional concurrency models, they do not use shared data or locks. Instead they communicate by passing asynchronous messages. In practice, each `Actor` can represent a represents a thread for example. Programming constructs in Scala use loops to process available messages in the mailbox by using pattern matching. An efficient implementation makes it possible to have little overhead by avoiding expensive context switching in some cases. Thus, it is possible for example to have thousands of active actors using only one active thread, for example.

1.2.3 Menthor

To make it easier to build machine learning applications that take advantage of multi-core environments with Scala, the Menthor framework was developed by DR. PHILIP HALLER and HEATHER MILLER [3]. The Menthor framework operates on data graphs. Each vertex holds a value of type `Data` that is iteratively updated. Menthor uses, like other graph processing frameworks, an iterative approach of super-steps that are composed of sub-steps.

At each step, computations are done locally at each vertex. Then each vertex can send out messages containing its results to its neighbors. During the next sub-step, each vertex can access a list of received messages. All the steps are synchronized. It is also possible to apply reduce operations on all the data vertices using a `crunch` function. The reduced result is then available at the next sub-step as a message.

The underlying implementation uses Akka actors that are similar to Scala's actors. All vertices are partitioned into groups that are each managed by a Worker that takes care of transferring messages and crunch results. The Workers are themselves managed by Foremen that perform similar tasks. Both the Foremen and the Workers are actors. The clever use of actors makes it possible for all operations to

be run in parallel leveraging all available hardware.

1.3 Goal

The purpose of this bachelor semester project is to investigate the use of Scala's parallel collections and Menthor framework to take advantage of multi-core architectures when writing machine learning applications working with large data sets. The aim is to provide quantitative results by comparing the running time of the same machine learning algorithm implemented sequentially, and with parallel collections as well as with the Menthor framework.

1.3.1 Expected outcome

It is expected that both the parallel collections and Menthor framework implementations will show significant speed improvements when an increasing amount of cores are available. The ideal case would be to have linear speedup compared to the amount of available cores.

It is also expected that the implementation using the parallel collections will be straightforward given a working sequential implementation. The Menthor implementation might require a longer learning curve since it uses a specific architecture.

2 Method and implementations

As mentioned previously a machine learning algorithm was implemented with the following implementations:

1. Scala sequential collections
2. Scala parallel collections
3. Menthor framework

It was chosen to implement a Gaussian Mixture (GM) Expectation Maximization (EM) algorithm. This kind of algorithm is extensively used for data clustering applications in industry as well as in research topics such as neurology or biology. The different implementations were then carefully benchmarked using generated data of various sizes and difficulty to assess their relative performance in different environments.

2.1 Expectation maximization algorithm

The expectation maximization algorithm (EM) is a class of algorithm used for finding maximum a posterior (MAP) estimates or the maximum likelihood of parameters in statistical models. The algorithm is composed of two alternating steps that are applied iteratively. First, the expectation (E) step is performed. It uses current estimates of the parameters to compute the expectation of log-likelihood.

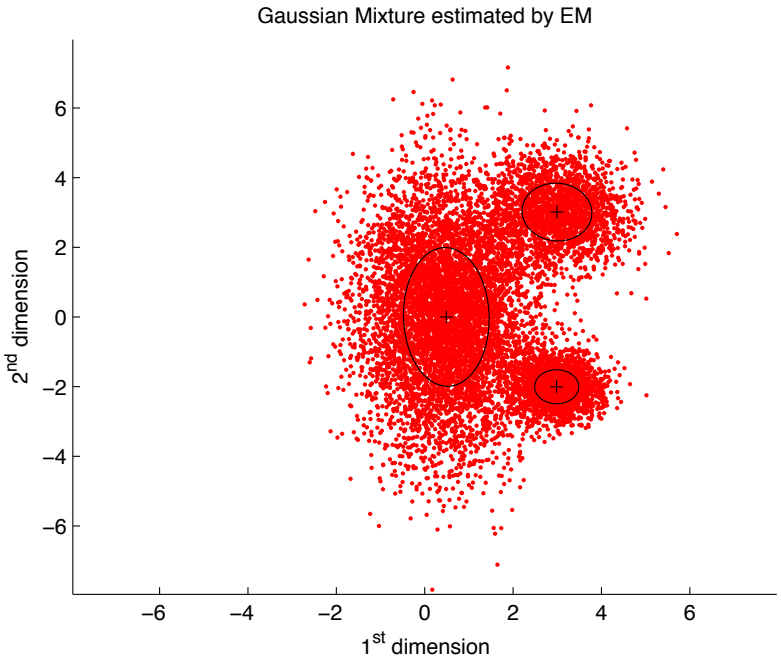

Figure 2: Example of GM EM output, 10'000 samples, 3 components.

Then the maximization (M) step, uses the log-likelihood calculated in the previous E step to maximize the parameters. The steps are iteratively applied until a halting condition is met. The halting condition can either be reaching a maximum number of iterations or that the variation in log-likelihood is arbitrarily small.

2.1.1 Gaussian Mixture

Gaussian Mixture is a EM algorithm variant for determining parameters in Gaussian mixture models and the clustering of the data. Figure 2 illustrates the output of a sample EM GM algorithm. The samples are coming from a three Gaussian sources. The crosses represent the estimated means and the ellips are the respective covariances.

The detailed steps of the algorithm are presented below. Some mathematical aspects were left out to emphasize more the structure of the algorithm and focus on how the algorithm could be implemented. The approach is inspired largely by the example that can be found on Wikipedia[1] and also from [1, 7, 5, 2] which present more formal approaches.

The data $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is composed of n independent observations each of dimension d. The latent variables $\mathbf{z} = (z_1, z_2, \ldots, z_n)$ determine from which com-

[1] http://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm#Example:_Gaussian_mixture

ponent the observation originates. We then have

$$X_i|(Z_i = j) \sim \mathcal{N}_d(\mu_j, \Sigma_j)$$

where

$$\mathrm{P}(Z_i = k) = \tau_k$$

and

$$\sum_k \tau_k = 1$$

with $0 < k < K$ where K is the number of Gaussian components.

The algorithm should estimate the unknown parameters :

$$\theta = (\tau, \mu, \Sigma)$$

The likelihood function to be maximized is :

$$L(\theta, \mathbf{x}, \mathbf{z}) = P(\mathbf{x}, \mathbf{z}|\theta) = \prod_{i=1}^{n} \sum_{j=1}^{K} I(z_i = j)\tau_j f(x_i; \mu_j, \Sigma_j)$$

where f is the multivariate normal distribution :

$$f(x_i; \mu_j, \Sigma_j) = \frac{1}{\sqrt{(2\pi)^d \left|\Sigma_j\right|}} e^{-\frac{1}{2}(x_i - \mu_j)^T \Sigma_j^{-1}(x_i - \mu_j)}$$

The GM EM algorithm then consists of the application of the steps described in Algorithm 1.

Different strategies exist for guessing the initial parameter $\theta^{(0)}$. In this work the *Kmeans* algorithm was used.

2.2 Implementations

The sequential and parallel versions work in the same way: methods that can be used are the same, only the type changes. Therefore the algorithm only had to be implemented in two fundamentally different ways: one using Scala generic collections and the other using the Menthor framework. Although very different in design, both implementations had to share as much code and logic as possible to allow fair comparisons in the benchmarks.

To allow this, a simple class hierarchy was designed. `Gaussian` is an abstract class that implements basic initialization procedures and declares the functions that have to be implemented for the algorithm to run, namely the `em` function. The `Gaussian` class is extended by `GaussianCollections` and `GaussianMenthor` that each implement the `em` method. The inner-workings of both implementations are detailed hereafter. It is important to get insights into their implementation to understand which operations could take advantage of multi-core environments.

Both variants use Scalala[2] for linear algebra and mathematical operations.

[2]"Scalala is a high performance numeric linear algebra library for Scala, with rich Matlab-like operators on vectors and matrices; a library of numerical routines; support for plotting." https://github.com/scalala/Scalala

Algorithm 1 GM EM Algorithm

1. Start with an initial guess for the parameter $\theta^{(0)}$
2. Expectation step: at the t^{th} step, compute

$$T_{j,i}^{(t)} = P(Z_j = j | X_i = x_i; \theta^{(t)}) = \frac{\tau_i^{(t)} f(x_i; \mu_j, \Sigma_j)}{\sum_{k=1}^{K} \tau_k^{(t)} f(x_i; \mu_j, \Sigma_j)}$$

$$Q(\theta | \theta^{(j)}) = E\left[L(\theta, \mathbf{x}, \mathbf{z})\right]$$

3. Maximization Step: determine the new estimate θ^{t+1}as the maximizer of $Q(\theta|\theta^t)$ over θ

$$\tau_j^{(t+1)} = \frac{1}{n} \sum_{i=1}^{n} T_{j,i}^{(t)}$$

$$\mu_k^{(t+1)} = \frac{\sum_{i=1}^{n} T_{k,i}^{(t)} x_i}{\sum_{i=1}^{n} T_{k,i}^{(t)}}$$

$$\Sigma_k^{(t+1)} = \frac{\sum_{i=1}^{n} T_{k,i}^{(t)} (x_i - \mu_k^{(t+1)})(x_i - \mu_k^{(t+1)})^T}{\sum_{i=1}^{n} T_{k,i}^{(t)}}$$

2.2.1 Implementation using Collections

In this implementation, the input data is stored in a generic sequence of `DenseVectors`. Each vector represents a sample. The implementation of the EM is pretty straightforward. A loop applies the E and M step iteratively. Between each iteration, the variation in log-likelihood is computed. Whenever the variation is arbitrary small or if the maximum number of iterations is reached, the algorithm stops and returns the current estimates.

Expectation In the E step, some parameters used later in the calculations are first initialized. Then a `map` function is applied on the input data. The `map` computes the expectation value corresponding to each sample.

Maximization In the M step, the current estimated weights are first computed by summing the expectation data with a `reduce` function. Then the estimated means are computed by multiplying each point with its corresponding estimate using a `map`. The new collection is then reduced with a summing function. Finally for each component, the estimated covariances are computed, once more by applying a `map` on the data, and then the estimates are zipped together. The resulting collection is then also summed.

2.2.2 Parallel vs. sequential

It is important to notice that almost all operations are done collection-wise using `map` or `reduce` functions. It means that they could potentially be run in parallel.

To allow the code to be very flexible, only `GenSeq` was used in the method signatures. During run-time we take advantage of polymorphism. If the input sequence is parallel, so will be the output, and all operations are done in parallel. No subsequent changes in the code are needed.

To allow easy benchmarking with the parallel and sequential version of the collections, the `GaussianCollections` was simply sub-classed with `GaussianSequential` and `GaussianParallel` which provide the right input types. This is illustrated in figure 3. In the final code the choice was made to use `Arrays` and `ParArrays` as input since they are easy to use and known for allowing good performance when the elements are accesed out-of-order.

2.2.3 Implementation using Menthor

For the Menthor implementation, a graph structure had to be designed to hold the data. The choice was made to have one sample per vertex. Since there are no relations between the samples, the vertices were not connected to each other. This means that the entire graph is unconnected and that no vertex has any neighbor to send messages to. Given this choice, the algorithm had to be designed to use a

```
class GaussianSequential(initStrategy: GaussianInit)
                        (dataIn: GenSeq[DenseVector[Double]],
                         gaussianComponents: Int)
   extends GaussianCollections(initStrategy)(dataIn.seq, gaussianComponents)

class GaussianSequential(initStrategy: GaussianInit)
                        (dataIn: GenSeq[DenseVector[Double]],
                         gaussianComponents: Int)
   extends GaussianCollections(initStrategy)(dataIn.seq, gaussianComponents)

abstract class GaussianCollections(initStrategy: GaussianInit)
                                  (dataIn: GenSeq[DenseVector[Double]],
                                   gaussianComponents: Int)
   extends Gaussian(initStrategy)(dataIn, gaussianComponents) {

  /** The implementation of the EM algorithm */
  def em(...) {
    ...
  }
}
```

Figure 3: GaussianCollections sub-classes

succession of sub-steps as enforced by Menthor. Algorithm 2 gives an overview of the implementation's eight sub-steps.

Algorithm 2 Menthor sub-steps

1. Compute the expectation at each vertex
2. Crunch step to sum the estimated expectation at each vertex
3. One vertex sets the new estimated weights in the shared data
4. Crunch step to sum the estimated means
5. One vertex sets the new estimated means in the shared data
6. Crunch step to sum the estimated covariances
7. One vertex sets the new estimated covariances in the shared data
8. One vertex computes the likelihood and decides if the iterations should stop

In the implementation of the algorithm, some data from the model such as the estimated weights, means and covariances are to be used across all vertices. These objects were stored in one `SharedData` object to avoid redundant copies.

```
class VertexValue(
  val point: DenseVector[Double] = null,
  var exp: DenseVector[Double] = null,
  private val estMeans: DenseMatrix[Double] = null,
  private val estCovariances: Array[DenseMatrix[Double]] = null) {
  def means = estMeans
  def covariances = estCovariances
}
```

Figure 4: VertexValue

```
class RealVertexValue(point: DenseVector[Double])
   extends VertexValue(point = point) {
  override def means = point.asCol * exp.asRow
  override def covariances = {
    (0 until gaussianComponents).toArray map(k => {
      val delta = point.asCol - CurrentData.means(::, k)
      (delta * delta.t) :* exp(k) :/ CurrentData.weights(k)
    })
  }
}
```

Figure 5: RealVertexValue

crunch A limitation was that the crunch step only allows for `(Data, Data) => Data)` function to be applied on all vertices. However in this implementation it is necessary to apply different operations involving different types of data when the crunch step computes the estimated means or covariances. In these cases, the input and output of the function are not of the same type.

To overcome this issue a simple wrapper class was implemented. `RealVertexValue` (see Figure 5) objects hold the sample and the current expected value in each vertex. It also provides methods `means` and `covariances` that return the part of the mean and covariance that must be computed at each vertex before being summed altogether. This class is also sub-classed with `VertexValue` (see figure 4) that is only used during the crunch step. It helps carrying the current sum. When `means` and `covariances` are called on this object it simply returns the running sum. Figure 6 illustrates the use of both classes. `x` and `y` can be either of type `RealVertexValue` or `VertexValue` but when they are summed, they get wrapped in a `VertexValue` object. This was made possible to do in a readable way by taking advantage of Scala's default parameters.

2.2.4 Menthor improvement: crunchToOne

The Menthor architecture was not at all optimized for this particular implementation, which could introduce significant overhead. Indeed, in the original implementation of Menthor, the result of the crunch step is sent as a message to all vertices. However here, only one vertex needs the result to store it in a shared data structure. The framework was modified to allow this behavior by implementing the `crunchToOne` function. To make this possible, the entire structure of the Menthor framework had to be slightly altered. The results of the crunchToOne step had to be wrapped in a marker object for the foremen and workers to know it only had to be delivered to a specific vertex.

The only difference at the end with `crunch`, is that for the next step only the first vertex of the graph receives the results, saving the sending of $n-1$ redundant messages (where n is the number of vertices).

2.2.5 Notes on testing

To be sure the algorithm was correctly implemented in Scala, its outputs where carefully compared to a MATLAB version of the algorithm upon which the implementation was based. Extensive unit tests were also written to support the development and maintenance of the code base.

2.3 Experimental setup

Scala code benchmarking is delicate since there are several actors that can affect the running time that can come into play. Care had to be taken to avoid problems with the following mechanisms.

Garbage collection During the lifetime of an application running on the JVM garbage collection is invoked periodically by the JVM. Its task is to reclaim the memory used by objects that are no longer referenced. Every time the garbage collector is called, there is an impact on performance, since a part of the CPU is dedicated to that task. To avoid this problem, the GC can be called manually when the code is not being benchmarked. It lowers the probability of being called later on by the JVM.

HotSpot optimizations HotSpot is the JVM developed by Oracle. When the bytecode of a Scala application is loaded in the JVM, it is not optimized yet. During the run of the JVM the "hot spots" will be localized and optimized allowing the application to run faster. It typically takes a few runs of the

```
crunch((x,y) => new VertexValue(exp = x.exp + y.exp))
```

Figure 6: Crunch usage

```
class BenchmarkedClass extends TicToc {
  def runAlgo(...) {
    // Initialization code
    tic
    // Code to be benchmarked
    toc("description")
    // Other code
  }
}
```

Figure 7: TicToc usage

same code for the performance to become stable. Other implementations of the JVM implement similar mechanisms.

Benchmark code The code used for the benchmarking can itself introduce overhead when invoked. Therefore, the number of operations used when benchmarking have to have a very small footprint. The benchmarking code should also be implemented to only take into account the desired parts of the code and not the application as a whole.

Other applications If other processes are running alongside the application, they can randomly impact performance and introduce variance in the results.

The detailed techniques used to minimize the errors introduced by these issues are detailed below.

2.3.1 Benchmark code

To time the code the `TicToc` trait provided with Menthor was used. `TicToc` makes it possible to time various parts of the code with little intrusion and minimal overhead. It also provides basic functions to display and output the benchmarking results to a file.

The only parts of the code that were benchmarked were the execution of the EM algorithm. To do this, the `Gaussian` class implemented the TicToc trait. Both subclasses could then delimit the parts of the code that were to be timed (example in figure 7). The time to read the data from the file, initialize parameters etc. were not taken into account.

2.3.2 Benchmark run

To reduce the impact of the time it takes for HotSpot to optimize the code during run time, the EM algorithm was run multiple times in a row. Typically, after only a few runs, the optimizations are done and the speed is constant. Therefore each benchmark run consisted of the the iterative execution of the program 15 times in

a row on the same JVM. The first five (usually biased by HotSpot) results were discarded and the average of the ten other was made.

Between each run the Garbage collector was called. Calling it before running the algorithm lowers the probability of the garbage collector running during the execution of the code that is being benchmarked.

2.3.3 Launch scripts

The application was always launched remotely to avoid unnecessary processes (GUI, other applications etc.) running on the machines. A script took care of packaging the latest version of the code into a JAR and executing it with the different subsets of available cores and data sets. The execution of the code could be monitored remotely by using `top` on Linux to make sure no other user or process was interfering.

2.3.4 Machines and configuration

The benchmarks were run on machines with distinct configurations to have different comparison bases.

MTCQUAD The MTCQUAD server has 4 Dual-Core AMD Opteron 8220 SE running at 2.8Ghz which gives a total of 8 physical cores.

ICBC07 The ICBC07 machines run on an Intel Core i7 920 at 2.66Ghz. It has 4 real cores and supports hyper-threading[3].

All tests were carried out with Scala 2.9.2. The MTCQUAD machine was running Oracle JVM 1.6 64bit in server mode. The ICBC07 machines were running OpenJDK 1.6 32bits in server mode.

2.3.5 Core usage

On both machines the cores could not be physically switched off. Their use had to be prevented by the operating system so the JVM only has access to a subset of the available cores. When running the benchmarks in a Linux environment the `taskset` command was used to make this possible, by setting the CPU affinity of a new process. Figure 8 illustrates the use of `taskset` to execute the application that was previously packed into a JAR on cores 0 and 4. On Windows, a similar command exists: `start`.

[3] "Intel Hyper-Threading Technology uses processor resources more efficiently, enabling multiple threads to run on each core. As a performance feature, Intel HT Technology also increases processor throughput, improving overall performance on threaded software." source: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

2.3.6 Data sets

Two datasets were used for the benchmarking. The first one had 50'000 points coming from six Gaussian sources. With the default parameters, it converged after six iterations. The second dataset is much larger, with 600'000 points coming from five sources. It converges after eight iterations.

3 Results

3.1 General remarks

Figures 9 and 10 show the plots of all the implementations running respectively on the ICBC07 and MTCQUAD machines.

Plots at the top of each figure represent the average running time in milliseconds of each implementation depending on the number of cores.

Plots at the bottom show the speedup factor for each implementation. Each series are normalized to the run with one core. They are used to asses separately the speedup of each implementation.

3.2 ICBC07

Important note The CPU has only four physical cores. Intel's hyper-threading makes the OS see it as eight distinct cores. The hardware tries to optimize the core usage by interleaving the simultaneous execution of the eight threads transparently. In the benchmarks the cores were always combined in pairs. For example when two cores were used it was the two cores associated to the same physical core.

One can notice that the running time when only one core is available is the fastest with the sequential implementation. The Parallel implementation is around 5% slower whereas the Menthor implementation takes around twice as much time to run.

We also see that the running time of the Menthor implementation is always around 1.7 - 2 times longer than the parallel collections implementation with the small dataset. With the large data set the numbers are 1.4 - 1.7. It only gets faster than the sequential implementation when it has more than 5 cores (3 physical cores).

The running time of the sequential implementation improves slightly when more cores are available.

The speedup plots indicate that the speed of both the parallel and Menthor implementations scale with the number of cores. When all cores are used the speedups

```
$ taskset -c 0,4 java -jar em.jar
```

Figure 8: `taskset` usage limiting the execution of em.jar to core ids 0 and 4

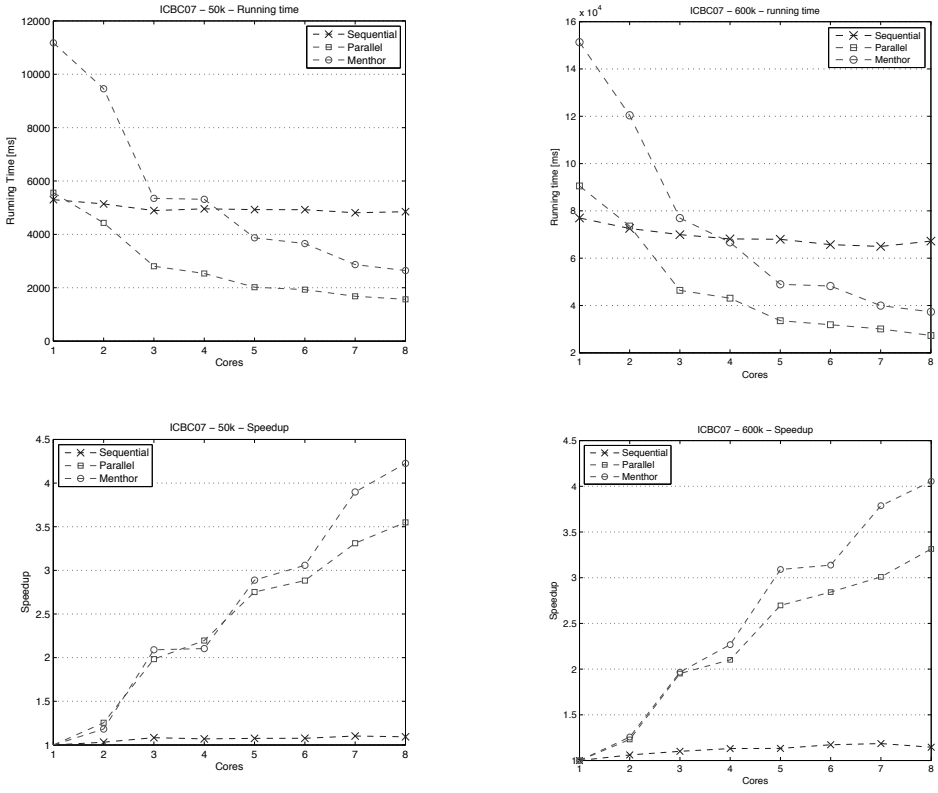

Figure 9: Run on ICBC07 machines

are respectively 3.5 and 4.2 for the 50k and 3.3 and 4.1 for the 600k run compared to the use of one core. If the cores are grouped in pairs to only compare the speedup when the cores are fully used we get the speedup from 1 to 4 physical cores to be respectively 2.8 and 3.6 for the 50k dataset and 2.7 and 3.2 for the 600k dataset.

One can also notice that steps appear in the speedup plots for every pair of cores that is added.

3.3 MTCQUAD

The results coming from the MTCQUAD machine share similarities with the outcome coming from the ICBC07 machine. The running time of the Menthor implementation is always significantly slower than with the parallel collections. When only one core is available for the 50k dataset, it takes twice as much time to complete, when all cores are available this number goes down to 1.5. These numbers are respectively 1.7 and 1.2 for the larger dataset. When three cores or more are available Menthor beats the running time of the sequential implementation.

The execution time of the sequential implementation is also nearly constant, with the 600k dataset it shows a slight improvement when it is slightly decays with

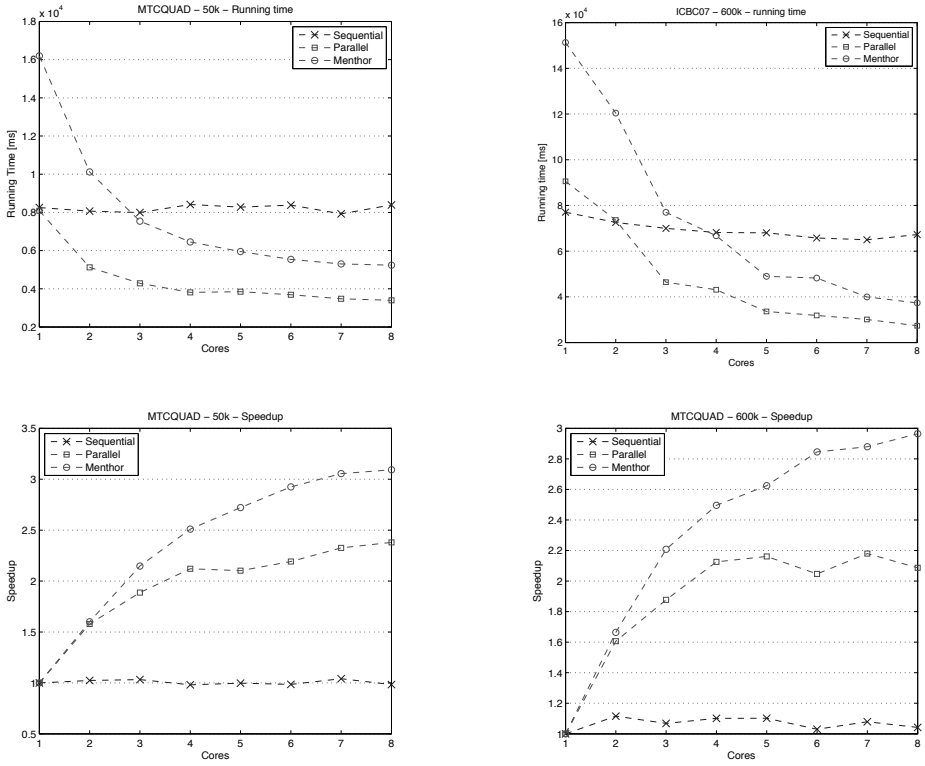

Figure 10: Run on MTCQUAD machine

the other dataset.

Both implementations show important speedups when more cores are available. One can notice that the relative speedup each time a core is added decays inversely proportional for the Menthor implementation. When both cores from the first CPU are used, the speed improved by around 60% for both datasets. When a third core is added, it is improved further by around 30% compared to two cores, then 15% with the fourth and so on.

For the parallel collections implementation the speedup grows quickly until two CPUs (four cores) are used. When all the CPUs are used, the variation is only of +12% for the 50k dataset and for the 600k dataset the performance drops by a couple of percent.

One can also notice the small drop in performance going from 5 to 6 cores and 7 to 8 cores with the parallel collections implementation when running with the large dataset.

3.4 crunchToOne

The outcome of the benchmarks comparing the Menthor implementation using `crunch` and `crunchToOne` is illustrated in figure 11. For the 50k dataset the the

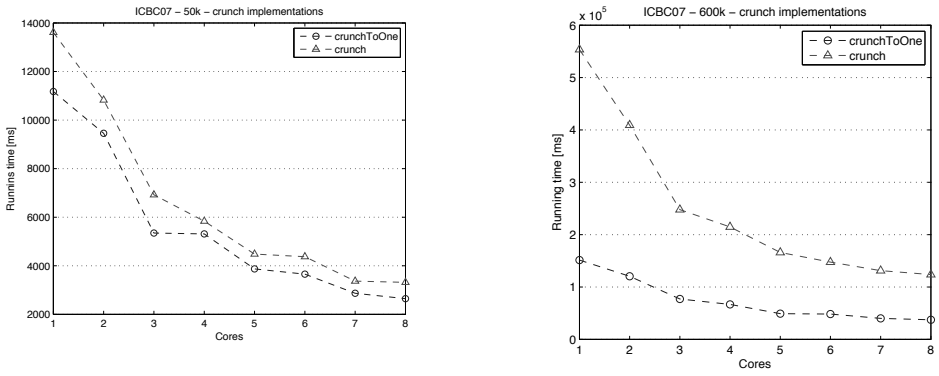

Figure 11: crunch vs. crunchToOne running on ICBC07

implementation using `crunchToOne` runs on average about 20% faster than the other. For the 600k, set the difference is much bigger: the implementation using the `crunch` function take over three times more time to execute.

4 Discussion

4.1 Running time and speedup

4.1.1 General observations

On both machines the parallel and Menthor framework implementations run, as expected, faster the more cores are available. Nevertheless, due to hardware differences, the scaling behaves differently. These differences are detailed below.

The Menthor framework implementation runs slower than the parallel implementation with all configurations and datasets. Several explanations are possible. First the steps in the Menthor framework are synchronous. Hence, there is waiting between the substeps until they are all completed. Regardless of whether or not the wait is necessary. Many more objects are instantiated during the run which is time expensive. Empty lists are created at each vertex during each substep for example. The actors might also introduce overhead since there are more actor threads than available cores. On the other hand, the Menthor implementation scales slightly better than the parallel version. This must come from the fact that it makes better use of the CPU time. Each actor performs relatively small operations so there is less overall waiting time.

4.1.2 ICBC07

The results align well with the hypothesis. The speedup grows nearly linearly. This is possible since allmost all operations in the application can be done in parallel. (c.f. part about the implementation).

The fact that the Menthor implementation scales slightly better with the larger dataset might come from the architecture. Smaller collections, and less expensive operations are used in the Menthor implementation. When using the parallel collection, the arrays have to be recursively split before they can be processed separately.

The steps in the speedup graph come from the hyper-threading mechanism. When the second core in each pair of virtual pair is added, a slight improvement is possible, since the use of the CPU is optimized.

4.1.3 MTCQUAD

The general observations are still true for the run on the MTCQUAD machine. But a surprising result is that the application does not scale to take full advantage of the eight physical cores, that are available. Even when allowed to use all cores it only had a maximum speedup of 3.1 with the Menthor framework. This must be explained by the machine's architecture. It is composed of four distinct Dual-core CPUs. The architecture is not capable of properly balancing the load or might me limited by memory acces or bus speed. There is likely more latency when communicating between two processors on different chips than communicating between cores on the same chip.

4.2 Implementations

4.2.1 Collections

The code was first written based on a MATLAB implementation of the algorithm by keeping a much resemblance in the structure. The problem however was that the code was very procedural and relied heavily on the Scalala library by using `DenseMatrix` and `DenseVectors` to hold all the data.

The code was then iteratively modified to rely more on the Scala collections to hold the data. There was an important shift from procedural to a more functional code. However the use of the Scalala library was not completely abandoned, since it provided elegant solutions for some linear algebra operations.

Once the sequential version using Arrays was working, it was, as expected, very easy to make it ready to accept parallel collections objects as inputs. Only the method signatures were altered to use `GenSeq` instead of explicit concrete types in addition to sub-classing `GaussianCollections`. No other code had to be changed.

Programmers should have a very easy time making their sequential code take advantage of parallel speedups. Developers coming from procedural backgrounds can also take advantage of these collections when using `for` loops or other constructs - they are not constrained to use functional constructs.

4.2.2 Menthor

The implementation using the Menthor framework was, at first, more complicated. There was a learning curve for adapting to the framework's structure and design. In contrast with the other implementation, it put more emphasis on the processing of each sample instead of taking input data as a collection which is not usual when coming from a procedural programming background.

It was also a difficult to take ful advantage of the crunch function. An another approach was tried by using "master nodes" that would take care of reducing the results coming from "child nodes". This however would not have been as scalable and easy to maintain as an implementation using the crunch step. The optimal number and size of partitions would have been necessary to find out etc. In the end, as previously mentioned, the crunch function combined with a wrapper object was used. However a method with the same signature as fold Left (`foldLeft[B](z: B)(f: (B, A) => B): B`) for example would have made the development easier.

Once these problems were solved, the implementation was easy. The code is makes also the mathematical steps and formulas appear more clearly which is good for code maintenance.

If another algorithm was now to be developed in addition it would not need as much time for adapting since the learning curve was already passed.

4.2.3 Improvement: crunchToOne

The results strongly support the implementation of the `crunchToOne` function. The running time of the smaller dataset are sped up by 20%, which is already non-negligible. As for the 600k dataset the speed improvements are drastic. The reason is that the ratio of messages that don't have to be sent scales linearly with the number of vertices. When, for the smaller dataset, the ratio of saved messages is 1:50'000, it becomes 1:600'000 for the larger one. The waiting time between the substeps is also reduced.

4.3 Conclusion

Overall, the work shows very encouraging results that align with the hypothesis. Developers should not refrain from using parallel collections whenever possible since it provides tangible speedups for real applications. This is consistent with the benchmarks in [6]. Other works go into more details on what size the collections should be to have an optimal outcome.

For a more specific machine learning application, the Menthor framework was also a good solution. It provides good speedup and is relatively easy to use. The EM GM algorithm however is certainly not the best suited for this kind of framework since it does not originally rely on graphs. However the framework is flexible enough to be used anyway.

Regarding the hardware, it appears that the use of configurations with a single CPU with multiple cores is preferable than to have multiple CPUs with less cores.

4.3.1 Future work

Future work could investigate how the implementations scale when even larger datasets are provided to verify is the trend is confirmed. It would be also interesting to run the same benchmarks on other machines with different hardware (more CPUs, more cores / CPU etc.)

5 Acknowledgments

I would like to thank HEATHER MILLER, doctoral assistant at LAMP for her guidance, insights and support throughout the project. I also would like to thank DR. PHILIPP HALLER for once reviewing the code and for sharing valuable insights. I also would like to thank PROF. MARTIN ODERSKY for allowing me to do this semester project at LAMP and for his inspiring talks and lectures. Special thanks also go to my family and friends for their support throughout my studies.

6 Cited work

[1] Jeff Bilmes. A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Technical report, 1998.

[2] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

[3] Philipp Haller and Heather Miller. Parallelizing Machine Learning- Functionally: A Framework and Abstractions for Parallel Graph Processing. In *2nd Annual Scala Workshop*, 2011.

[4] Philipp Haller and Martin Odersky. Event-Based Programming without Inversion of Control. In David E. Lightfoot and Clemens A. Szyperski, editors, *Modular Programming Languages*, Lecture Notes in Computer Science, pages 4–22, 2006.

[5] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2009.

[6] Aleksandar Prokopec, Tiark Rompf, Philip Bagwell, and Martin Odersky. A Generic Parallel Collection Framework. Technical report, 2010.

[7] Jason D. M. Rennie. A short tutorial on using expectation- maximization with mixture models, 2004.