HPC Project Report: Parallel Implementation of Logical Regression

Rajdeep Pinge 201401103, Aditya Joglekar 201401086

November 7, 2016

Part I

Hardware Details:

We have tested our codes on two machines having following two different architectures. This is done because certain parallel implementations of code give better performance for some specific number of processors.

1.
   - **CPU model:** Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
   - **Memory Information:**
     (a) L1d cache: 32K
     (b) L1i cache: 32K
     (c) L2 cache: 256K
     (d) L3 cache: 15360K
   - **No. of cores:** The CPU has 12 physical cores with 1 thread per core. This means, there are 12 logical cores.
   - **Compiler:** gcc version 4.8.2
   - **Precision used:** We have used double precision wherever possible so as to get more accuracy in the result.
   - **cache line size:**
     Since our problem deals with enormous amount of memory access and processing, it is important to get to know the **cache line size** of the system. It can be obtained by typing in the following commands: getconf LEVEL1_DCACHE_LINESIZE, getconf LEVEL1_ICACHE_LINESIZE, getconf LEVEL2_CACHE_LINESIZE, getconf LEVEL3_CACHE_LINESIZE
     For all the four level of cache mentioned above, the cache line length is 64 bytes.
2.
   - **CPU model:** AMD A10-5750M APU with Radeon(tm) HD Graphics
   - **memory information:**
     (a) L1d cache: 16K
     (b) L1i cache: 64K
     (c) L2 cache: 2048K
   - **No. of cores:** The CPU has 4 physical cores with 1 thread per core. This means, there are 4 logical cores.
   - **Compiler:** gcc version 4.8.5
   - **Precision used:** We have used double precision wherever possible so as to get more accuracy in the result.
   - **cache line size:**
     For level1 cache it is = 64 bytes, for level2 cache also it is = 64 bytes.

Part II

1 Context:

- **Brief description of the problem.**

  Logistic regression is a technique to build a classifier. The **input** is a **labelled dataset** i.e. a set of training samples each having some input features and a class label. e.g {0.01,2.3,7.52,3.51} and 5.

  The **features** are typically **values which can encode some information** which can help us classify the sample into one of the available classes. We assume that a training sample can belong to only one class.

  Classification is typically of **two types**, **binary (0/1, yes/no) classification** or **multiclass (one-vs all classification).** The latter is an extension of the former.

  Binary classification: There are only two possible results of a given data and the learning and predictions is done for only two classes/results.

  Multiclass classification: This is a clever **extension of the same binary classification idea** called one vs all classification. If we have k classes, we train k binary classifier by considering the input labels to be 1 if they are of the type being considered (say i^{th}) else 0. Hence classification is done by learning corresponding weights for each class and checking the response of each for a given training example

- **Algorithm description:**

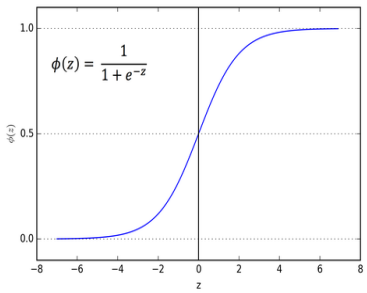
  Initially the program starts with random weights since we know nothing about the data and its characteristics. The **aim** is to **find out parameters(weights named as theta) which can accurately capture patterns pertaining to classification** by observing the given data and try and generalize it to unseen examples.

  To do this, an important quantity is the **cost associated** with a particular set of parameters (weights/theta). The **cost measures how well do the current weights predict the result with respect to the actual label (ground truth).** Lower the cost, better is the prediction.

  The **cost used for classification is the logistic regression cost.**

  This logistic regression algorithm **uses a non-linear function** which encodes some information about the likelihood of the input being of a particular class. We have chosen the 'sigmoid' function for this purpose (fig 1:).

Figure 1: The Sigmoid Function



Notice that for large input the function tends to 1 and for large negative inputs it tends to zero. The **output of the sigmoid function** when the linear combination of the input features and the weights is passed to it, is the **conditional probability of the label being 1 given the current parameters(weights)** i.e. $P(y = 1|\Theta)$ where the theta stands for the vector of all weights(parameters) used.

Figure 2: Cost function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Note: $y = 0$ or 1 always

Below (Fig 2) is the cost function used in binary classification...

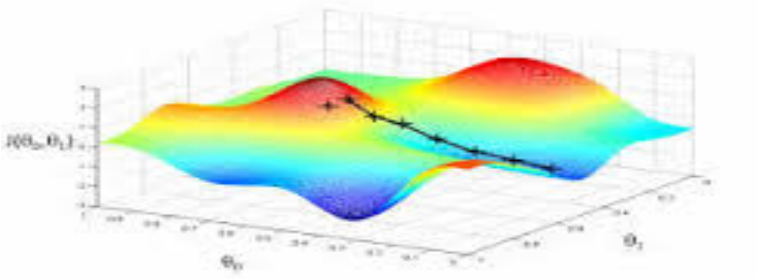
As the figure indicates(fig 2), **if the label is 1**, then the first term is applicable. This term will have a **large cost value if the hypothesis gives a low probability of the answer being one** i.e. if the prediction is wrong. Whereas if the probability is high then the value of the cost term will be lower.

This property of the cost makes it useful for learning the weights properly. If we want **weights which can correctly predict the ground truth**, it is the same as obtaining **weights which give the minimum possible cost or penalty of answering wrongly.**

Every weight vector (theta) is associated with a prediction cost which measures how good the weights are. The question now is how to minimize the cost which is a function of theta?

For this we use a **iterative minimization technique known as gradient descent.**

Figure 3: Gradient Descent



Pictorially, imagine that the cost curve w.r.t. the parameters is a mountain range. The aim is to reach the deepest valley(the minimum cost point with the best weights). At every iteration we 'feel' the ground around us and **descent in the direction of the negative gradient of the slope of the cost function.** This is a **greedy method** in which we use the fact that a function decreases fastest in the direction of the negative gradient. At every iteration we continue to move in this direction down the slope till we reach a relatively flat region. This method is **guaranteed to reach a local minimum** but it might not be the global minimum. However, in case of **binary logistic regression** it can be proven that the **shape of the cost function is convex i.e. there is unique minimum.** So in our problem we are guaranteed to reach the minimum after some time or the other. This is the **essence of the 'learning procedure'**

The figure below(Fig 4) is the **algorithm for learning weights**. Notice that it is an **inherently iterative procedure** because of its dependence on previous step **(Step dependency)**. By **convergence** means reaching an optimum. Usually instead of explicitly specifying a stop condition we simply **run this procedure sufficient number of times**. This is an important hyper parameter (number of iterations) because it will **decide the quality of the weights obtained and also determine the computational cost and hence the running time.**

Basically,for every iteration, for every weight Θ_j, we subtract the gradient of the cost function with respect to that weight multiplied to a constant factor known as the **learning rate α or the step size.** Using basic calculus we can obtain the summation term. The summation term involves computing the

Figure 4: Formula for gradient descent to adjust weights

Repeat until convergence

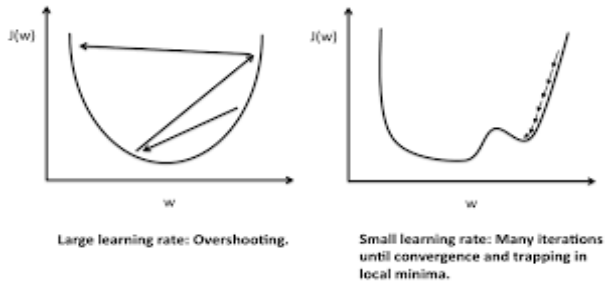
{

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}

activation(sigmoid output) for every training sample and multiplying it with the j^{th} feature of the i^{th} input sample. It is **very important to correctly set** α. If it is too small, then the algorithm will take an inordinately long time to converge. If it is too large, the algorithm may diverge instead of converging. As a rule of thumb α should be set around 0.1,0.01 +_ a degree of magnitude

Figure 5: Problems in convergence



- **Complexity of Algorithm:**

  The above formula needs a double loop one over the m training samples and one over the n weights for calculation of h_θ.

There are **two approaches** to do the above computation. For further reference let us call these methods as **outer loop over features(OLF) and outer loop over training samples(OLT).** Both give the same solutions (optimal weights) but have drastically **different work complexities.**

OLF (Outer Loop over Features): serial complexity: Step complexity is O(n*m). This is obvious because of the double loop. But notice that the in-inner loop over training samples, the sigmoid function takes n steps for every training sample (refer to the sigmoid function in the code). Thus for every feature update, the inner loop takes n steps. Thus, the complexity is actually $O(n^2m)$.

So overall, OLF for t iterations will take $O(tn^2m)$. Thus, is an expensive computation, even for 50 iterations. With 400 features, 1000 samples, it will roughly take $2 * 10^{10}$ steps.

OLT (Outer Loop over Samples): serial complexity: However in the OLT method, for every iteration, the other loop runs m times. For every input sample, the adjustment for each weight is calculated using the sigmoid function. This takes n steps. Then, the adjustments are subtracted from the current weights in n steps. Thus, the complexity of this algorithm is $O(t(mn + n))$. This is faster than the previous by n times! The reason being that the OLT code for every iteration calculates sigmoid once(requiring n steps). Thus, leading to savings by a factor of n. OLT has the added advantage of parallelizing over a larger problem size since it parallelizes loop over samples.

- **Possible speedup (theoretical).**

  Method 1: OLF As discussed above, the complexity of this method is $O(tn^2m)$. In the parallel code, we parallelize the outer loop over the columns. Hence theoretically, the new complexity should be $O((tn^2m)/p)$ where p is the number of processors. If we parallelize inner loop over rows, the complexity is same as the outer loop parallelization. Thus, theoretically we should get the maximum possible speedup of p. However,

we will soon see that this reasoning is specious as this does not take into account the overhead for maintaining the threads.

Method 2: OLT Here, the complexity was $O(t(mn + 2n))$. The outer loop over training samples has been parallelized. Hence the parallel complexity will be $O(t((mn/p) + 2n))$. Thus if we assume that m is sufficiently larger than n, then we can expect speedup close to p.

Notice, that the **speedup is completely independent of t(number of iterations)**. Mathematically it is cancelling out. Hence, the **number of iterations can't be the measure of our problem size**.

- **Optimization strategy:**

  The problem is highly granular as there is no communication needed to complete the work on each thread individually.

  This is an inherently iterative algorithm. The theta vector of the current iteration is obtained by calculating the gradient value at the previous iteration. This part simply cannot be parallelized as we can't know the weights for the i^{th} iteration without reaching the $i-1^{th}$ iteration. However at every iteration there is plenty of scope for parallelization.

  We have used the following optimization strategies to parallelize and optimize the processing time.

  1. **Case 1 - OLF:**
     (a) Simple parallelization of the inner loop over training samples. (Later, we see that this is a bad idea as the overhead is too great).
     (b) Parallalizing the outer loop over the feature columns. The effectiveness of this method depends partly on the problem size(i.e. number of feature columns in the data-set). The parallelizable work should be able to cover the overhead of maintaining the threads.
  2. **Case 2 - Using collapse with OLF:**
     (a) We noticed that it is possible to collapse the address space of the two loops into one and parallelize the two loops simultaneously. However, to confirm to OPENMP restrictions we had to put an if condition which restricts speedup.
  3. **Case 3 - OLT:**
     (a) We later realized that we **can decrease the complexity of the algorithm by a factor of n** by simply flipping the two loops. That is we parallelize over the input samples instead of over the columns. This needs a slight reworking of the gradient descent formula and we need an n-dimensional extra array to store the partial gradients obtained. This optimization drastically cuts down the time. The reason for this becomes apparent when we compare the two codes. The problem with the OLF method is that for every iteration of the outer loop over columns, the inner loop calls the costly sigmoid function n times, this is what gives this method a running time n times larger than OLT. OLT saves on this by using a little extra space(n dimensional array).
     (b) Another advantage of OLT being that the parallelization is done over training samples which is our case are more in number than the features by an order of magnitude.
     (c) Also, it is worth mentioning that in this code, to observe sufficient speedup it is important to use a reasonably large data set. This is in agreement with the observations from the HPC assignments where we used to get speedup only after the input size increased to a certain threshold.

- **Profiling Information:**

  We have found out profiling information for 3 serial implementations. There are two main functions in all the codes namely train() and sigmoid(). In every code, the train() function is called once to perform training on the data set. The calls to sigmoid depend on the complexity and implementation of train. Keeping all the parameters same, profiling tells us the following information:

  For Normal serial code: Time spent in sigmoid(): 51.68% and No. of sigmoid() calls = 6010990.

Serial code with loops swapped: Time spent in sigmoid(): 0.01% and No. of sigmoid() calls = 14990.

Serial code for multiclass classification with loops swapped: Time spent in sigmoid(): 0.31% and No. of sigmoid() calls = 5400000.

The above observations justify the decrease in total time when loops are swapped. The total calls to sigmoid are reduced when the loops are swapped and the value is stored in a local variable. So, the amount of data access and computation decreases while the resultant code has a lot of cache reuse due to the row measure matrix storage in C.

In case of multiclass code, the samples are more and hence the magnitude of data is much more than binary classification. But still the total calls to sigmoid are less than the binary serial code. Hence in some cases, even the multiclass code is faster than the binary classification

- **Problems faced in parallelization and possible solutions.**

  The fundamental limitation is we simply can't parallelize over the iterations which are inherently sequential. This is only a problem when we need to tweak the input size. The iterations cannot be used as input size as **speedup stays constant even if we increase the iterations.** So, the number of iterations are not very important as from the HPC perspective. However, it is an important parameter for the quality of the learning. Usually, with a good enough learning rate, the training accuracy increases with increase in iterations.

  In OpenMP, when we use the "parallel for" pragmas the threads created exist only for the scope of the for loop. In this code, for every iteration we need to use such pragmas, thus **in each iteration we need to spend time in creating and destroying threads**. This contributes **a lot to the overhead.** We could not find an openmp construct which allows us to reuse threads even if there scope has completed. Hence this overhead will always be there and may outweigh the obtained speedup if number of iterations are too high.

  Parallelizing the inner loop: It is a bad idea to parallelize the inner loops. This is because of the above mentioned problem of not being able to keep the threads created. This problem is more severely felt if the parallelization inside as the creation and destruction of threads happens more frequently.

  Problem size is dependent on the dataset dimensions. So **to change probelem size, the amount of data used has to be changed.**

  Other than these issues, the problem is highly granular as there is **no communication interdependency** in the parallel computations **for a given iteration**. Hence we expect a fair speedup.

2 Input. Output

The input consists of labelled a data-set. That is for every sample we have some features and a label denoting the class the label is to be classified in.

The algorithm has **two parts Learning and Prediction.** Learning involves obtaining weights which can correctly classify the training set and hopefully generalize to the new unseen samples. Prediction involves using the learnt weights to classify new unseen test samples.

Binary classification: Given a patient's medical diagnosis predict if the patient is healthy or sick (yes/no problem).

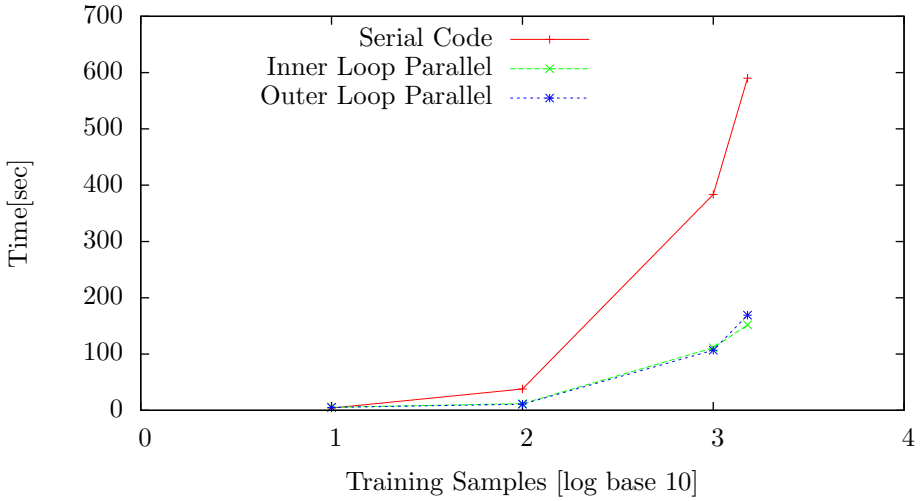
Multi-class classification: Classify MNIST handwritten images of digits 0 to 9 into 0-9 classes (standard ML problem). Input: Image parameters. Output: Number to which the image resembles.

3 A different kind of problem size

Normally, in a computational problem, the problem size is just a parameter, it can easily be changed. However, since the **input in our problem is a data set, to change the problem size we have to change the number of training samples and/or the number of features.** It is difficult to change the number of columns but we can change the problem size over a restricted range allowed by the number of training samples available. Thus we have limited points to plot the curves.

4 Problem Size vs Time (Serial, parallel) curve:

Basic Binary classification: Variation in time compared to the number of training samples iterations: 1000



Binary Swapped Loops: Variation in time compared to the number of training samples iterations: 1000

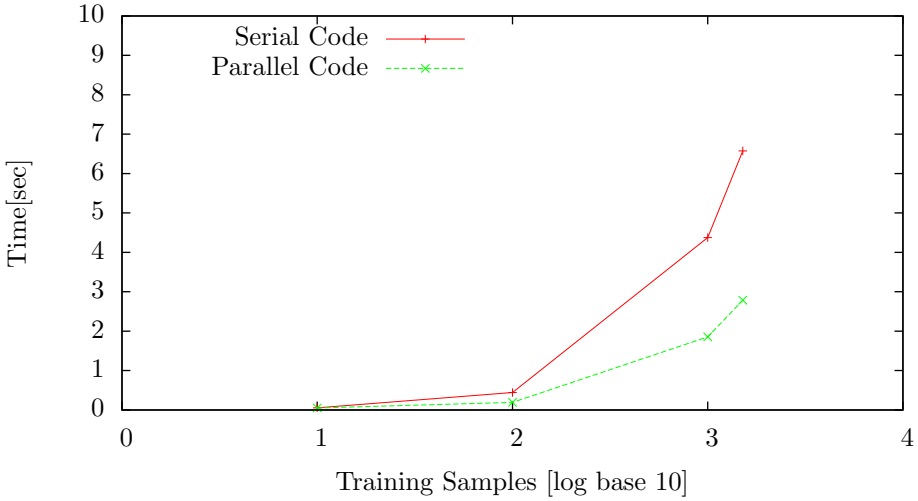
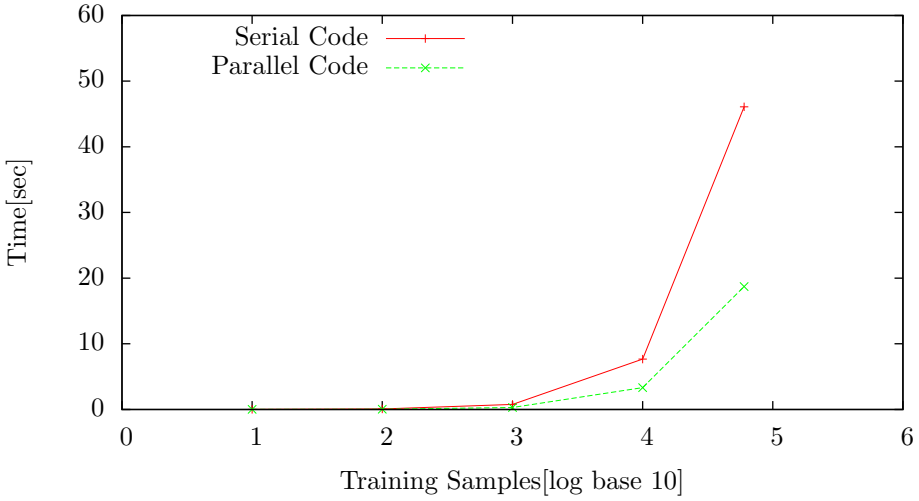


Table 1: Time taken by methods used for multiclass classification

Code	Iterations	Samples	features	Time(sec)
multiclass serial OLF	100	1499	401	1544
multiclass parallel OLF	100	1499	401	464
multiclass serial with loops swapped OLT	100	1499	401	58
multiclass parallel with loops swapped OLT	100	1499	401	23

Multiclass Swapped Loops: Variation in time compared to the number of training samples iterations: 10



Here the problem size is the number of training samples. As the number of samples increase, the parallelisable part of the code increases and hence the parallel code starts running faster than the serial code.

This can be observed as the gap between the curves is clearly increasing between x-values 1-2 to 2-3.

An important observation is the amount of time required by the Basic Binary classification code (in graph 1) for training as compared to the time taken by the Swapped Loop binary OLT code. Also the time taken by this OLF method is even more than the time taken by the Multiclass OLT method.

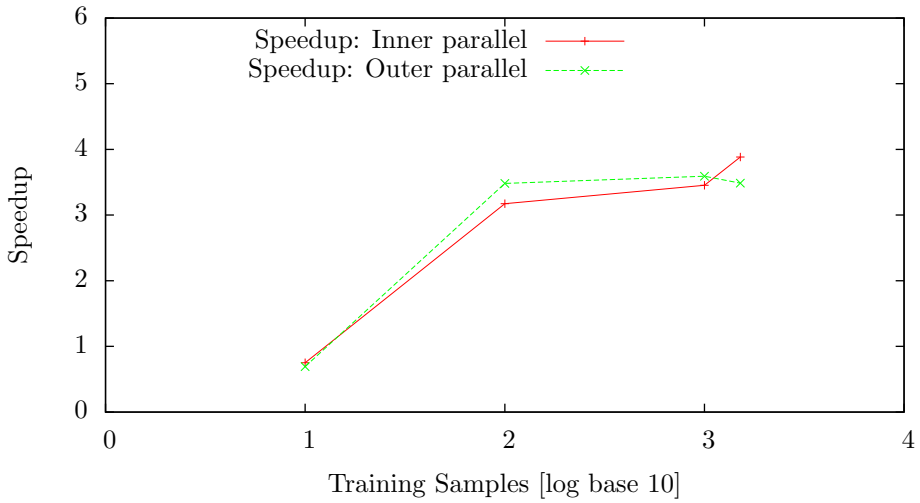
For the second graph of binary swapped loops OLT, there is significant increase in the time taken by the parallel code as the size increases. This trend is also followed by Multi-class OLT code. This is further analysed with the help of speedup.

5 Comparison of two methods

The Table 1 contains the information about time taken by different methods used in multiclass classification. As mentioned earlier, the code which uses swapped loops is much faster than the normal code for multiclass classification. This is because both of them have different work complexity for the costly sigmoid function. The sigmoid function is called n*m times in the simple code while it is called only m times and reused by soring it in a temporary variable in the swapped loop code. This makes a huge difference since an enormous amount of data access and computation is avoided. The extra data access and computation in the swapped loop code is not so costly and has a lot of cache reuse. Hence the Serial code runs much faster.

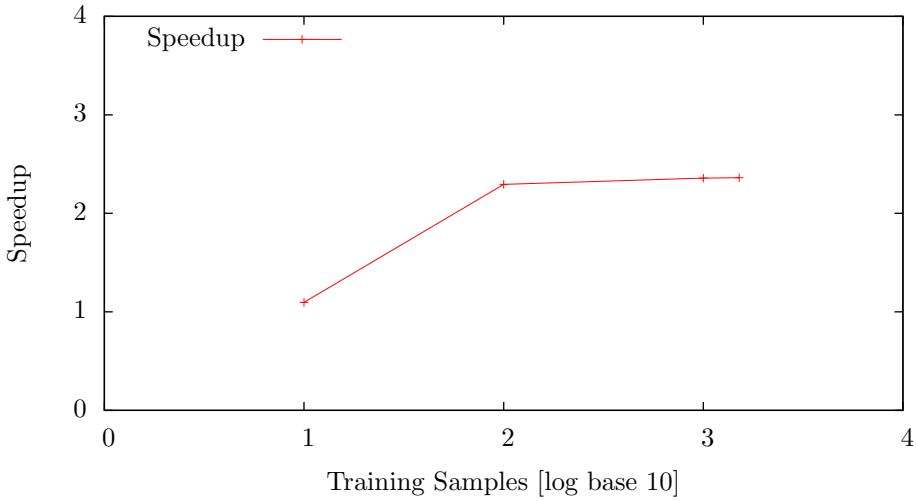
6 Problem size vs. speedup curve.

Basic Binary classification: Variation in speedup compared to the number of training samples iterations: 1000

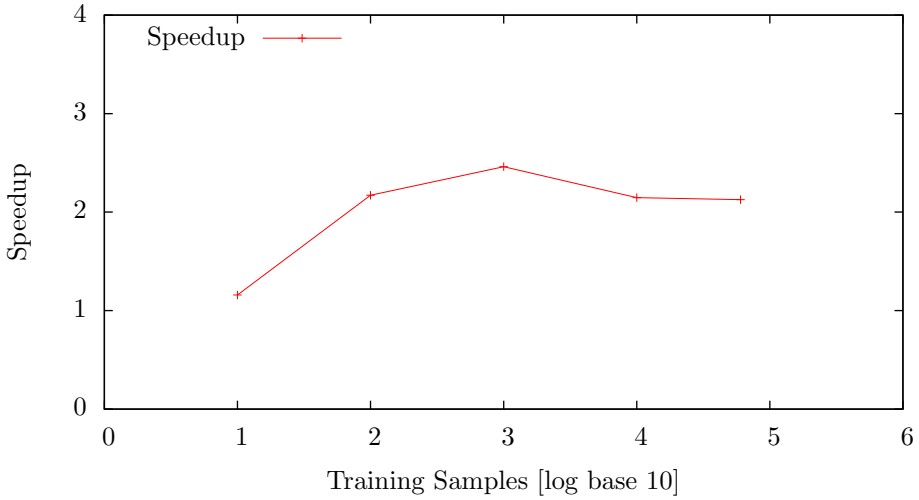


The above graph represents the usual situation of Amdahl's effect where as the problem size increases, the obtained speedup increases for a fixed number of processors. The maximum speedup is bounded by the number of threads which in this case is 4.

Binary Swapped Loops: Variation in speedup compared to the number of training samples iterations: 1000



Multiclass Swapped Loops: Variation in speedup compared to the number of training samples iterations: 10



We normally expect the speedup to go on increasing with increase in the problem size. This is so as the parallelisable part of the code increases while the overhead and non-parallelisable part stay constant. For the Binary Swapped Loop OLT curve, surprisingly, the speedup saturates after a point. This is something we had never seen before in our assignments!

The part of the code which is being parallelised is the outer loop(over rows/samples) of a double loop. Nested inside this loop is an expensive sigmoid operation which cannot be parallelized.

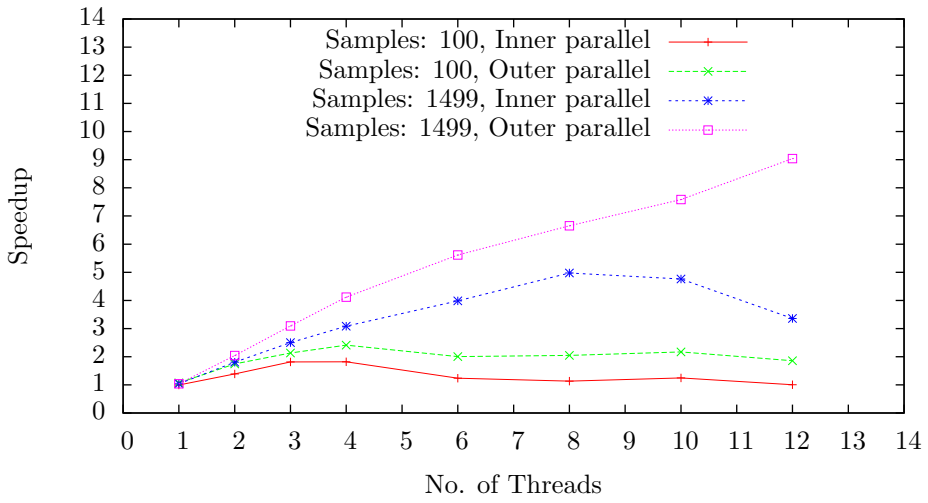
The sigmoid function cannot be parallelized because this drastically increases the parallel overhead. This is because **sigmoid parallelization is vector dot product parallelization over the columns which in our data-set is in hundred's** (not a great magnitude for parallization [from our assignment experience, this should normally be 10^6]). Moreover, this sigmoid is called **inside** in the inner loop. The profiling information tells us that it is the function called the maximum number of times. Thus, the parallelization of the sigmoid is a bad idea as it is called each time we loop over the iterations.

The above has major repercussions. Normally, when we increase the input size, the non-parallelizable serial fraction of the code stays constant(as number of processors and iterations stay constant). Here,the problem is that when we increase the number of rows, the parallelisable work does increase however so does the number of times sigmoid(the part which can't be parallized as discussed above). Hence the advantage in increasing the problem size is poisoned by the increase in the serial part. Hence, the speedup does not increase indeterminately. These restrictions cause it to saturate.

7 No. of cores vs. speedup curve

7.1 Observations taken on the HPC cluster

Basic Binary classification: Variation in speedup compared to the number of threads



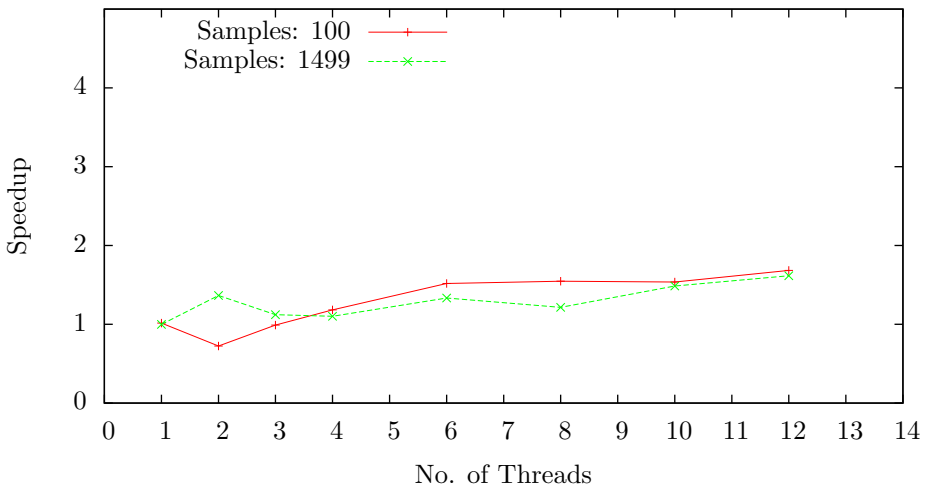
For Number of samples = 1499, note that the inner loop parallel OLF code has lesser speedup than the outer parallel version of OLF code. This is because of the larger overhead incurred for managing threads inside a double loop than doing the same for the outer loop. For inner parallel OLF, all the threads are created and destroyed in iteration of outer loop this has a lot of overhead.

The graph agrees with our theoretical expectations. We see that **for a given input size, there is an optimal number of threads for maximum speedup after which it starts decreasing because of overhead.** Here the optimum speedup is for 8 threads after which it goes on decreasing.

For lesser problem size (=100 samples), The **speedup increases till 4 threads** but afterwards the speedup of both the methods decreases. This may be attributed to the fact that the **problem size is lesser and hence the amount of processing that each processor does is less. Therefore the overhead due to higher number of threads is much more than gain due to parallelization.**

For larger input size the optimal point (no. of threads) is further right i.e. it can afford running a larger number of threads.

Binary Swapped Loops: Variation in speedup compared to the number of threads



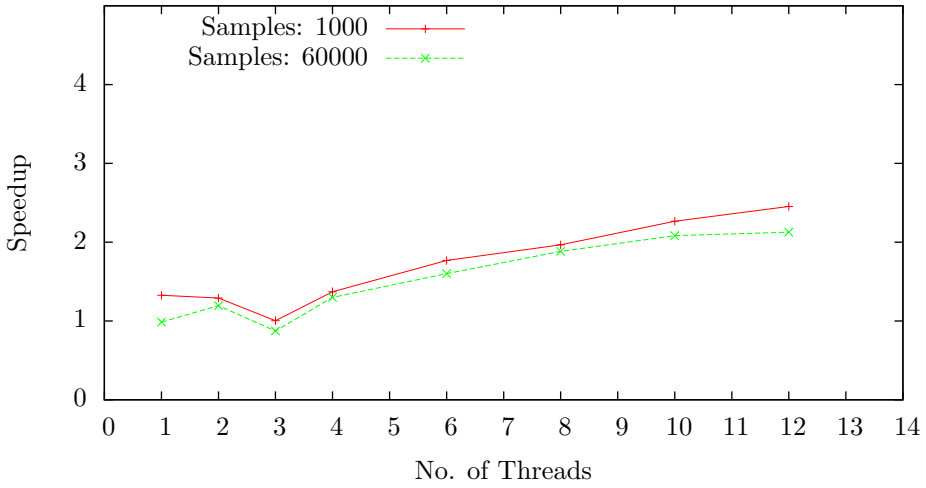
The speedup in the above graph increases very slowly as the number of threads are increased. This is

because of the nature of the problem. As number of threads increase, the **continuous spawning and merging of all threads for each iteration increases the overhead.** Similarly the inherent sequential nature of the problem and small amount of possible parallelizable code put restriction on the speedup obtained.

Here the OLT method is used. The speedup obtained is around x1.5. The two graphs show typical behaviour of **greater speedup for greater input size** till 4 threads. However, if we increase the threads beyond this point, the graph starts behaving in an unusual manner. **The one with the larger input size starts giving lesser speedup than the one with the lesser input size!**.

This is highly unusual. However, on thinking a little we might attribute it to a peculiarity of our OLT code. As mentioned before (see explanation for speedup saturation on Pg 9), when we **increase the input size, not only does the parallelisable part increase but so does the non-parallelizable and expensive sigmoid computation. This trade-off is okay till thread 4 but beyond that, the rate of increase in speedup for larger input starts decreasing and the smaller problem performs better.**

Multiclass Swapped Loops: Variation in speedup compared to the number of threads



This graph for multiclass classification really puzzled us! The speedup was decreasing, instead of increasing, with increasing input size. This seems to be contradictory to the Amdahl's effect.

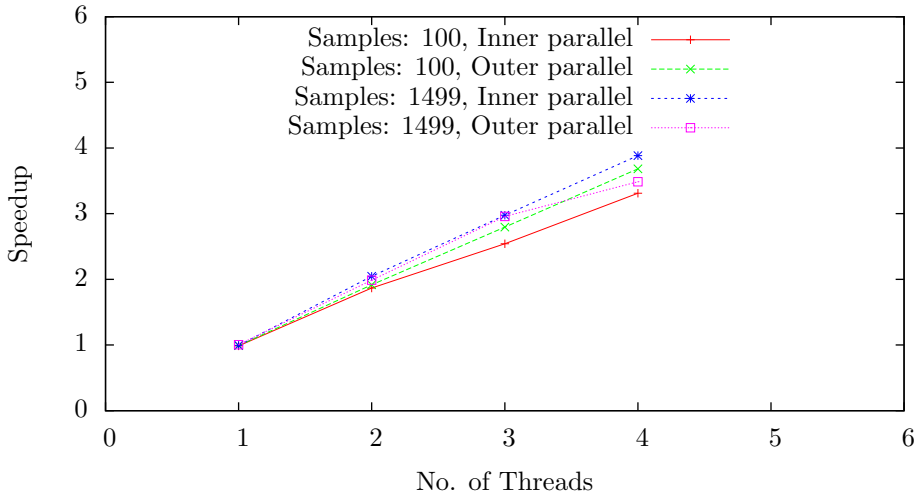
Again, looking at the code can provide an explanation to this anomaly. In multiclass classification, there is an additional loop for training k binary classifiers for each of the k classes. In this loop, we need to convert our original input into binary with respect to each class.

This is the only extra work we do in the multiclass code and suspiciously it is a function of the input size. It also has a lot of data access.

We thus feel that the work done inside this encoding (which has a lot of data access) leads to crippling overhead so much so that the **Amdahl's effect is reversed.**

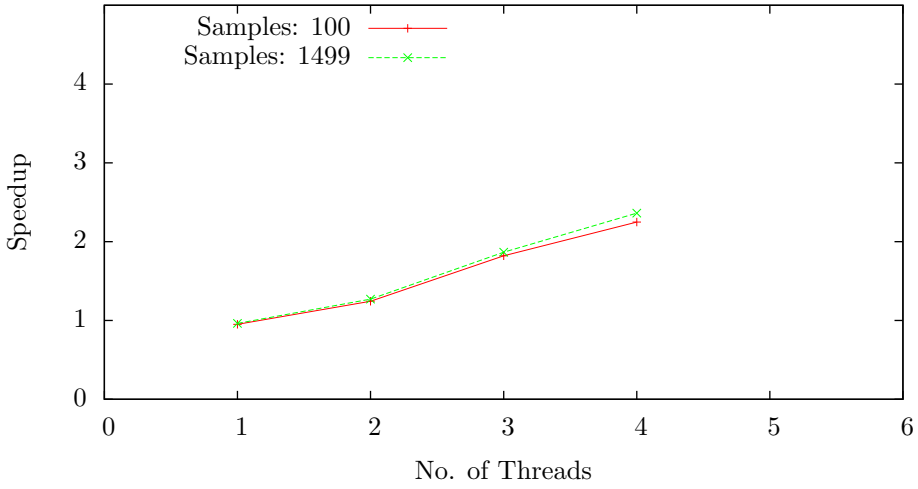
7.2 Observations taken on the Laptop with Quad Core Processor

Binary Swapped Loops: Variation in speedup compared to the number of threads

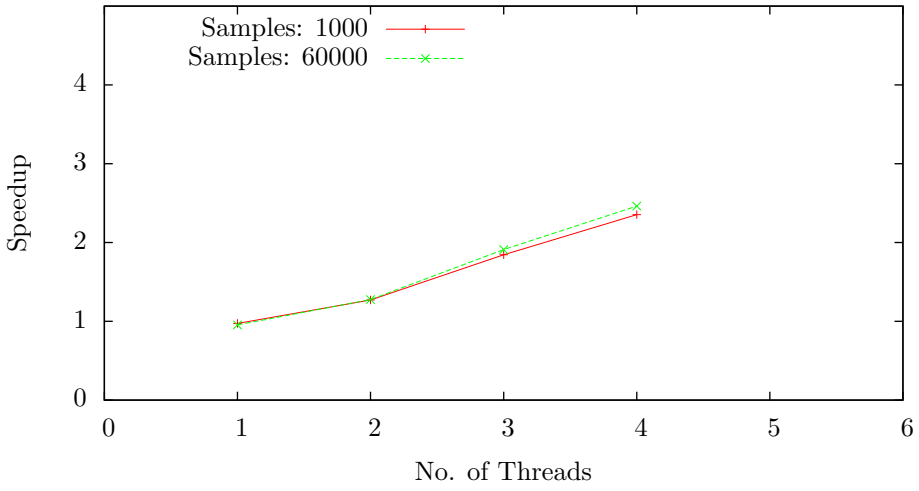


The above graph is similar to the results given by the cluster for 4 cores. But this is very misleading because the whole nature of the curves changes when run on larger number of threads

Binary Swapped Loops: Variation in speedup compared to the number of threads



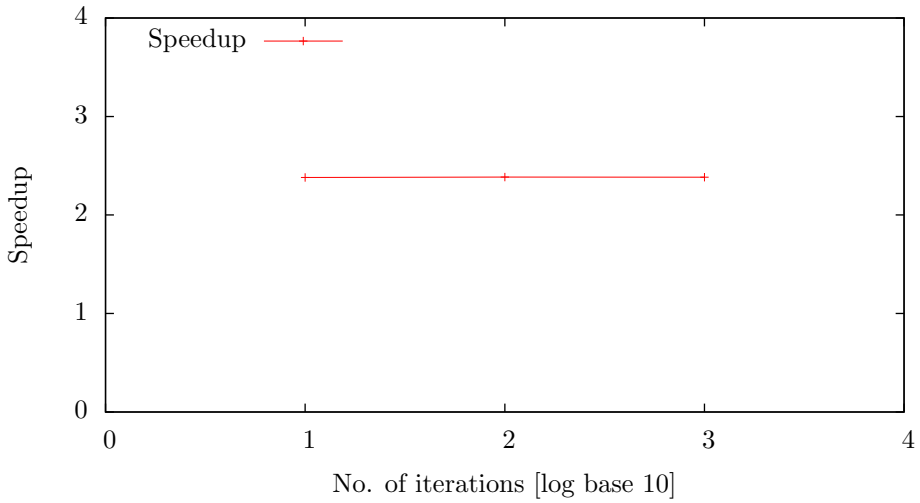
Multiclass Swapped Loops: Variation in speedup compared to the number of threads



The above two graphs nearly match with the observations taken on HPC cluster and the general trend. As the number of threads increase, the speedup increases. But these also don't give the proper nature of the graph for a larger number of threads.

8 No. of iterations vs Speedup curve

Variation in speedup compared to the number of iterations samples: 60000



When we had theoratically derived the speedup(see. **Section: Possible speedup(theoretical)**).

We had predicted that the speedup will not increase with increasing the number of iterations. This claim stands vindicated when we observe the above graph.

9 Peformance metric calculations

Experimentally determined speedup using Karp Flatt metric and Gustafsson's law:

Calculation for multi-class code with 60000 samples,785 columns,100 iterations.

Experimentally determined Speedup: 2.4

Hence according to Karp-Flatt metric, e (experimentally determined non-parallelisable part in parallel code) is

$$e\Gamma((1/\psi) - (1/p))/(1 - (1/p))$$

= 0.22

This gives a theoretical speedup of 3.34 (Gustafson's law). The empirical speedup obtained is less than this.

Note we cannot give an iso-efficiency analysis for this code. **We have observed that this code is not scalable, ie the speedup up does not scale up linearly even when we increase input size.** The reasons for this behaviour have also been discussed.

10 Conclusion

The serial part of the code is concerned with the machine learning part of the problem i.e. **finding the optimal weights which can help predict the classification for unseen data**. This can be done by finding out the optimal hyper parameters.

We were fairly successful in solving the machine learning aspect of the problem. **We were able to achieve training accuracy close to 100% and testing accuracy of around 90%(mutli-class) and 95%(binary)**.

The method to learn weights is **inherently sequential** at the highest level and cannot be parllelized. We however can parallelize the code for each iteration.

We focused on **parallelising the double loop of the train function**. The double loop can be written in two ways. **We implemented two methods for this. OLF(Outer Loop over Features) and OLT(Outer Loop over Training Samples)**.

We realized that **OLT is much more efficient than OLF**. So much so that the serial code of OLT is faster than the parallel code of OLF. **The speedup obtained in OLT however is less than OLF but the absolute time is much lesser**.

We also added **minor improvements like regularization(machine learning) and loop collapsing(HPC)**. These do not lead to any significant improvements.

1. **Binary Classification:**

   OLT:The best speedup achieved is around x2.4-x2.5 for 4 threads. **The speedup does not increase beyond this point. Unusually, beyond this point for larger input we get lesser speedup.**. This was unprecedented and we have attributed it to the non serializable part which increases with increasing problem size.

   OLF: **Even though in absolute terms this method is far slower than OLT, the speedup obtained here is much better than the former**. However, it will be unwise to think that this code is better, because speedup after-all is the comparison between a code's serial and parallel version. In this case we **get more speedup because OLF's serial code performs even worse**. Here we were able to get a scalable speedup which was increasing with increase in number of threads.

2. **Multi class classification**:

   We decided to implement multiclass using OLT, as it is much faster than OLF.

   Multiclass speedup shows similar speedup (max x2.5) like binary OLT method. **However, a surprising observation is that it does not seem to follow Amdahl's effect**. In the speedup vs processor graph (Page 11), the speedup decreases for increasing input size!

   We propose that this is due to the increase in the non-serializable encoding procedure which is necessary for Multi-class code to work. It is a function of input size and thus counteracts the gain obtained by parallelization.

We think there are several reasons why a perfect speedup(approx. p) is not possible here

1. A major problem, the adjust weights function is iterative at the highest level of abstraction. This **cannot be parallelized because of step dependency**. So, there is a significant portion of the code which is serial in nature.

2. The code requires **huge amount of data access**, for each iteration, we have to go over all training samples, and update each weight.

3. The inner loop which has expensive functions, cannot be parallalized without drastically increasing the overhead.

However, machine learning algorithms typically take a long time to run(hours,days,weeks). Hence even a speedup of x2.5, if consistent is extremely significant.

11 Future Scope

1. The problem has huge amount of computations and equally huge amount of data access. Hence this problem can be further extended to distributed computing systems.

2. The accuracy obtained in prediction over the available data sets is 90%. In practice, the makers of the data set have obtained an accuracy of 94%. So still there is some scope for improvement in parallel algorithm.

3. The state of the art classification model are neural networks which build on the logistic regression idea. Parallel implementation of neural networks will be a challenging extension of this problem.

12 References

- MNIST dataset samples = 60000, features = 785 http://pjreddie.com/projects/mnist-in-csv/
- C Library for Linear Algebra Support: GNU Scientific Library https://www.gnu.org/software/gsl/
- This is a standard machine learning problem. For the basic theory part of the problem, we referred to the Online course conducted by Prof. Andrew Ng, Stanford University. The complete code has been written by us with the help of this theory material.