High Performance Computing

Ionospheric Radar Returns Classification using SGD

Details-

Sujoy Datta (CED18I063)
Sem 7(4th year)- CSE Dual Degree

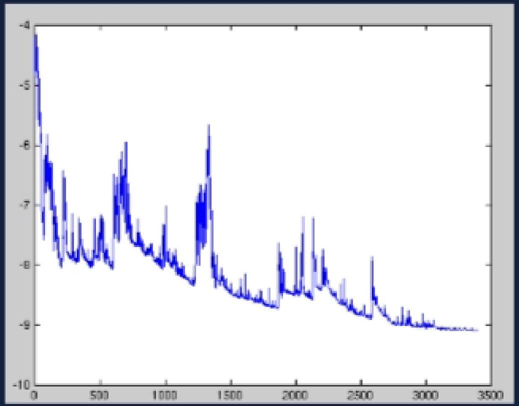
HPC

Relevance

The Ionosphere is at the horizon of the atmosphere and outer space. Interestingly, Ionosphere exploration falls into the category of Solar System Exploration.

The ionosphere is the ionized part of the Earth's atmosphere from 48 km to 965 km, which includes the thermosphere and parts of the mesosphere and exosphere.

In Ionospheric research, we need to classify the signals as useful(good) or useless(bad) for further analysis.



Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

Addressable Problem Statement

Classifying signals based on high frequency antenna responses and determining line of best fit using parallel computing.

Evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions. Parallelising this operation can improve computational time.

Stochastic Gradient Descent Algorithm

Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum. Evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions. Parallelising this operation can improve computational time.

Gradient descent is an iterative algorithm that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function. Iteration mechanism is sigmoid.

For a dataset with r feature dependencies the line of best fit can be determined using the stochastic gradient descent algorithm. The value of constants [b][0...r] can be calculated using a fixed learning rate and predicted initial values. The sum-minimization problem also arises for empirical risk minimization.

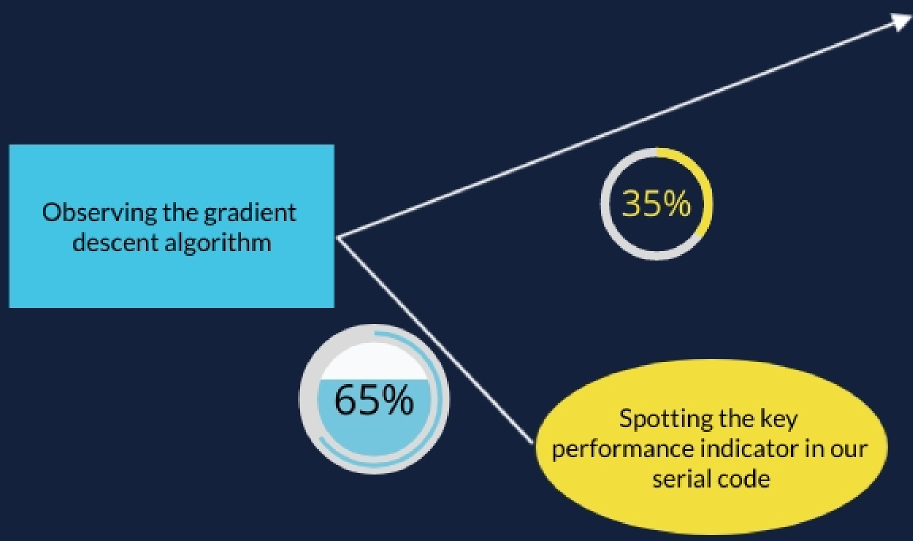
$$p(\mathbf{x}) = \frac{1}{1 + \exp(-f(\mathbf{x}))}$$

$$f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$$

Why use SDG for Ionospheric Regression?

Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties.

So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.



Classification of Radar Returns

Received signals were processed using an autocorrelation function whose arguments are the time of a pulse and the pulse number. There were 17 pulse numbers for the Goose Bay system. Instances in this database are described by 2 attributes per pulse number, corresponding to the complex values returned by the function resulting from the complex electromagnetic signal.

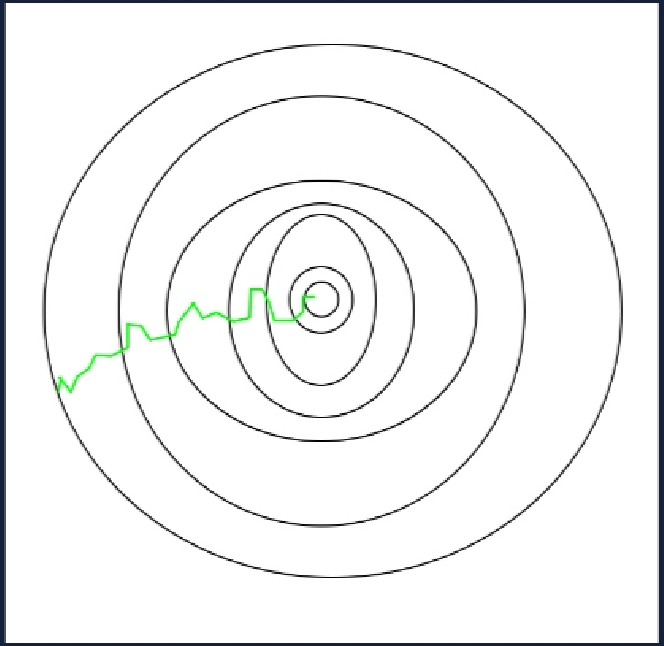
The Goose Bay Laboratory has 34 high frequency antennas that detect the probability of free electrons present in the atmosphere. For the ionospheric classification we can use the exploratory data analysis to determine the most influential data features.

$$\textit{for } i \textit{ in range}(m):$$

$$\theta_j = \theta_j - \alpha(\hat{y}^i - y^i)x_j^i$$

Algorithm

This is the crux of the algorithm. The general idea is to start with a random point (in our parabola example start with a random "x") and find a way to update this point with each iteration such that we descend the slope.



Path taken by Stochastic Gradient Descent

The steps of the algorithm are

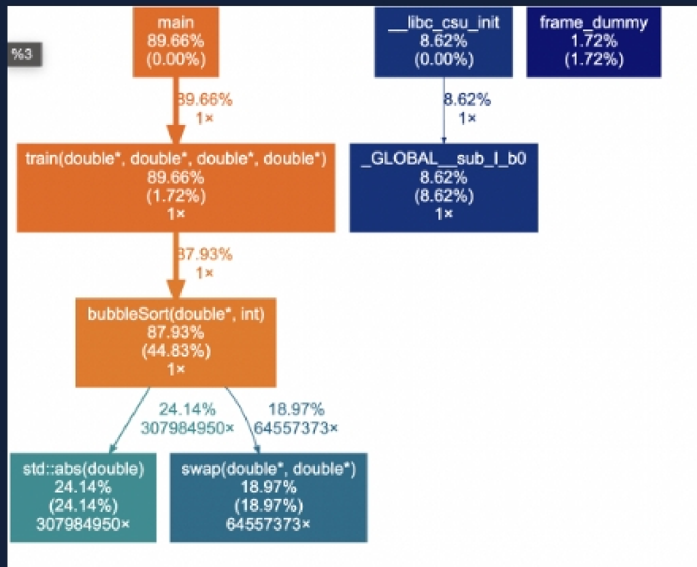
- Find the slope of the objective function with respect to each parameter/feature. In other words, compute the gradient of the function.
- Pick a random initial value for the parameters. (To clarify, in the parabola example, differentiate "y" with respect to "x". If we had more features like x1, x2 etc., we take the partial derivative of "y" with respect to each of the features.)
- Update the gradient function by plugging in the parameter values.
- Calculate the step sizes for each feature as : step size = gradient * learning rate.
- Calculate the new parameters as : new params = old params -step size
- Repeat steps 3 to 5 until gradient is almost 0.

Serial Code Block

```
for (i = 0; i < 10530; i++)
{
    idx = i % 10;//for accessing index after every batch
    p[i] = -(b[0][i] + b[1][i] * x1[idx] + b[2][i] * x2[idx] + b[3][i] * x3[idx]);//making the prediction
    pred[i] = 1 / (1 + pow(e, p[i])); //calculating final prediction applying sigmoid
    err[i] = y[idx] - pred[i]; //calculating the error
    for(int j=0;j<100000;j++)
    {
        b[0][i]=b[0][i] - alpha * err[i] * pred[i] * (1 - pred[i]) * 1.0;     //updating b0
        b[1][i]=b[1][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x1[idx]; //updating b1
        b[2][i]=b[2][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x2[idx]; //updating b2
        b[3][i]=b[3][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x3[idx]; //updating b3
    }
    //cout << "\tB0= " << b[0][i] << " " << "\t\tB1= " << b[1][i] << " " << "\t\tB2= " << b[2][i] << "\t\tB3= " << b[3][i] << "\t\tError=" << err << endl;
    error[i]=err[i];
}
bubbleSort(error,i);
```

Stochastic gradient descent is a variant of stochastic gradient descent in which you estimate the gradient of the cost function for each observation and update the decision variables accordingly. This can help you find the global minimum, especially if the objective function is convex.

Functional Profiling



It displays the executed code in a visual diagram where each node corresponds to a function or method and their relations show the code flow.

Line Profiling

```
17550:   62:        b0 = b0 - alpha
17550:   63:        b1 = b1 + alpha
17550:   64:        b2 = b2 + alpha
17550:   65:        b3 = b3 + alpha
    -:   66:
```

Profiling allows us to learn where the program spent its time and which functions called which other functions while it was executing. This information can show which pieces of the program are slower than expected

Parallelising SGD

The preprocessor directive #pragma is used to provide the additional information to the compiler in C/C++ language. This is used by the compiler to provide some special features.

Based on the inferences derived from profiling outputs and obtaining the critical section of the code, it is indeed important to parallelise the section using the pragma directive.

```
start=omp_get_wtime();
#pragma omp parallel for
for (i = 0; i < 10530; i++)
{
    idx = i % 10;//for accessing index after every batch
    p[i] = -(b[0][i] + b[1][i] * x1[idx] + b[2][i] * x2[idx] + b[3][i] * x3[idx]);//making the prediction
    pred[i] = 1 / (1 + pow(e, p[i])); //calculating final prediction applying sigmoid
    err[i] = y[idx] - pred[i]; //calculating the error
    for(int j=0;j<100000;j++)
    {
        b[0][i]=b[0][i] - alpha * err[i] * pred[i] * (1 - pred[i]) * 1.0;     //updating b0
        b[1][i]=b[1][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x1[idx]; //updating b1
        b[2][i]=b[2][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x2[idx]; //updating b2
        b[3][i]=b[3][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x3[idx]; //updating b3
    }
    //cout << "\tB0= " << b[0][i] << " " << "\t\tB1= " << b[1][i] << " " << "\t\tB2= " << b[2][i] << "\t\tB3= " << b[3][i] << "\t\tError=" << err << endl;
    error[i]=err[i];
}
bubbleSort(error,i);
//custom sort based on absolute error difference
end=omp_get_wtime();
```

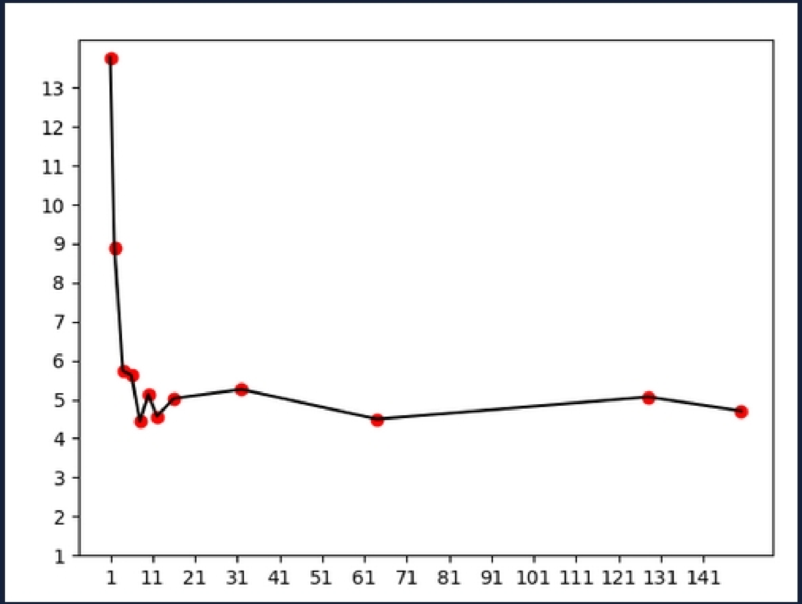
Observational Tabulation

# Threads	Execution Time	Speed Up	Parallelization Fraction
1	13.78	1	
2	8.8952	1.54915010342657	70.8969521044993
4	5.7365	2.40216159679247	77.8277697145622
6	5.6258	2.44942941448327	71.0089985486212
8	**4.44**	**3.1036036036036**	**77.4621604810284**
10	5.1394	2.68124683815231	69.6710208030963
12	4.5685	3.01630732187808	72.9238685842459
16	5.0275	2.74092491297862	67.7503628447025
32	5.2587	2.62041949531253	63.8329509808512
64	4.4976	3.0638562789043	68.4306217891124
128	5.0651	2.72057807348325	63.741083162863
150	4.7076	2.92718157872377	66.2793076241221

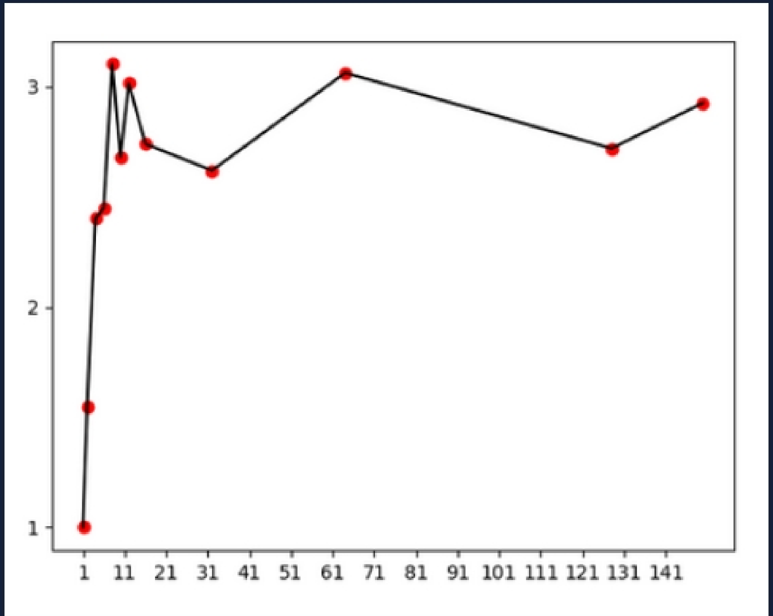
Speed up can be found using the following formula,
$S(n)=T(1)/T(n)$
where, $S(n)$ = Speedup for thread count ‘n’
$T(1)$ = Execution Time for Thread count ‘1’ (serial code)
$T(n)$ = Execution Time for Thread count ‘n’ (serial code)

Parallelization Fraction can be found using the following formula,
$S(n)=1/((1 - p) + p/n)$
where, $S(n)$ = Speedup for thread count ‘n’
n = Number of threads
p = Parallelization fraction

Threads vs Execution Time



#Threads vs Speed-up



Inferences-

- At thread count 8 maximum speedup is observed as the maximum number of parallel threads supported by the hardware is 8.
- If thread count is more than 8 then the execution time increases slightly and tapers out after 32 threads.

Ionospheric Radar Returns Classification using SGD

High Performance Computing Project Report

Problem Statement: Classifying signals based on high frequency antenna responses and determining line of best fit using parallel computing.

Faculty Guide: Dr. Noor Mahammad

By,

Sujoy Datta
CED18I063

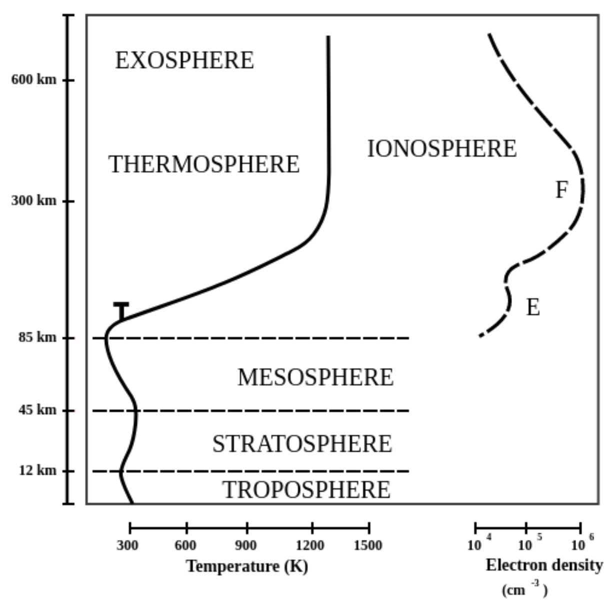
Code Repository- https://github.com/sujoyyyy/Ionospheric-Radar-Returns-Classification-using-SGD

Introduction

The Ionosphere is at the horizon of the atmosphere and outer space. Interestingly, Ionosphere exploration falls into the category of Solar System Exploration. The ionosphere is the ionized part of the Earth's atmosphere from 48 km to 965 km, which includes the thermosphere and parts of the mesosphere and exosphere.

Reasons to classify signals for the Ionosphere-

1. It houses all the charged particles of the Earth's atmosphere.
2. It is the boundary between Earth's atmosphere and space.
3. The orbital drag is felt in this region.
4. The favourite hangout place for our Earth Orbiting satellites.
5. It's influenced by fluctuations in our weather conditions on Earth.
6. Radio and GPS signals are disrupted by radiation in the Ionosphere.
7. Influenced by weather conditions in space.

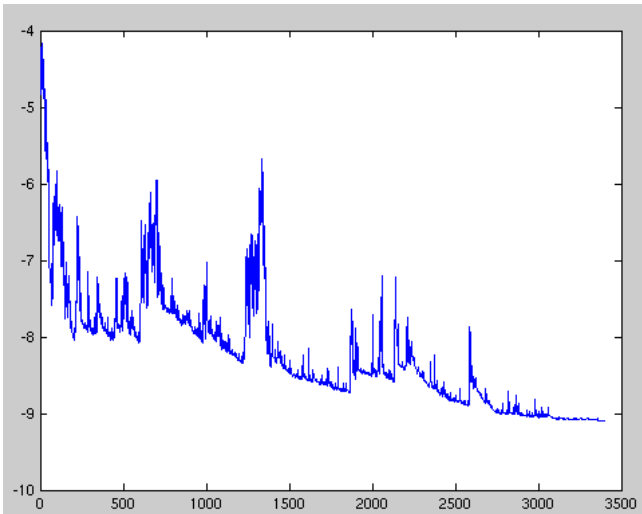


In Ionospheric research, we need to classify the signals as useful(good) or useless(bad) for further analysis. Often in such analysis manual intervention is necessary and it's a painful time-consuming task. The John Hopkins Applied Physics Laboratory has made the data collected from Goose Bay, Labrador radar in the UCI machine learning repository.

This problem from the Geophysics domain can be mapped into a binary classification problem in Machine learning.

Stochastic Gradient Descent Algorithm

Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum. Evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions. Parallelising this operation can improve computational time.



Fluctuations in the total objective function as gradient steps with respect to mini-batches.

Gradient descent is an iterative algorithm that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function. Iteration mechanism is sigmoid.

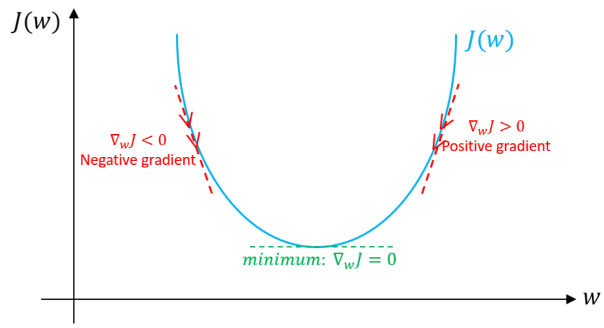
Mathematical Formula-

$$p(\mathbf{x}) = \frac{1}{1+\exp(-f(\mathbf{x}))}$$

$$f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$$

For a dataset with r feature dependencies the line of best fit can be determined using the stochastic gradient descent algorithm. The value of constants [b][0...r] can be calculated using a fixed learning rate and predicted initial values. The sum-minimization problem also arises for empirical risk minimization.

Why use SDG for Ionospheric Regression?



Relevance

1. Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties. So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.
2. It can be regarded as a stochastic approximation of gradient descent optimization. In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm.
3. Especially in high-dimensional optimization problems this reduces the computational burden, achieving faster iterations in trade for a lower convergence rate. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minimum and with significantly shorter training time.

Classification of Radar Returns

This radar data is collected by a system in Goose Bay, Labrador. The system consists of a phased array of 16 High-frequency antennas with a total transmission power of 6.4 kilowatts. The targets were free electrons in the ionosphere.

Received signals were processed using an autocorrelation function whose arguments are the time of a pulse and the pulse number. There were 17 pulse numbers for the Goose Bay system. Instances in this database are described by 2 attributes per pulse number, corresponding to the complex values returned by the function resulting from the complex electromagnetic signal.

The Goose Bay Laboratory has 34 high frequency antennas that detect the probability of free electrons present in the atmosphere. For the ionospheric classification we can use the exploratory data analysis to determine the most influential data features.

1st high freq antenna	2nd high freq antenna	3rd high freq antenna	Signal classification
0.42267	-0.54487	0.18641	g
-0.16626	-0.06288	-0.13738	b
0.60436	-0.2418	0.56045	g
0.25682	1	-0.32382	b
-0.05707	-0.59573	-0.04608	g
0	0	-0.00039	b
-0.04262	-0.81318	-0.13832	g
1	1	0	b
0.45114	-0.72779	0.38895	g
0.16595	0.24086	-0.08208	b
0.30996	-0.89093	0.22995	g
1	-1	1	b
0.68714	-0.64537	0.64727	g
1	0.88428	1	b
1	0.32492	1	g
1	0.23188	0	b

Received signals were processed using an autocorrelation function whose arguments are the time of a pulse and the pulse number. There were 34 high freq antennas for the Goose Bay system.

Algorithm

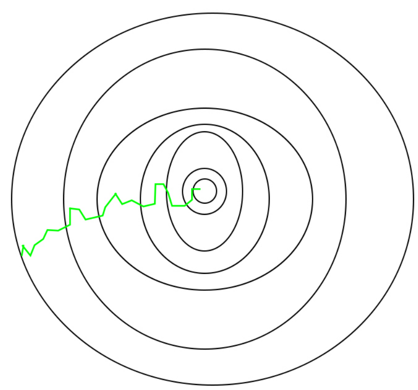
This is the core part of the problem. The general idea is to start with a random point (in our parabola example start with a random "x") and find a way to update this point with each iteration such that we descend the slope.

$$for\ i\ in\ range\,(m):$$

$$\theta_j = \theta_j - \alpha\,(\hat{y}^i - y^i)\,x_j^i$$

The steps of the algorithm are

1. Find the slope of the objective function with respect to each parameter/feature. In other words, compute the gradient of the function.
2. Pick a random initial value for the parameters. (To clarify, in the parabola example, differentiate “y” with respect to “x”. If we had more features like x1, x2 etc., we take the partial derivative of “y” with respect to each of the features.)
3. Update the gradient function by plugging in the parameter values.
4. Calculate the step sizes for each feature as : step size = gradient * learning rate.
5. Calculate the new parameters as : new params = old params -step size
6. Repeat steps 3 to 5 until the gradient is almost 0.



Path taken by Stochastic Gradient Descent

Softwares

1. C/C++, programming language
2. OpenMP, API for shared-memory parallel programming
3. MPI, High performance Message Passing library
4. Cuda C/C++, API for utilizing CUDA-enabled GPU for computation

---

Core Meaning

Stochastic gradient descent algorithms are a modification of gradient descent. In stochastic gradient descent, you calculate the gradient using just a random small part of the observations instead of all of them. In some cases, this approach can reduce computation time.

Online stochastic gradient descent is a variant of stochastic gradient descent in which you estimate the gradient of the cost function for each observation and update the decision variables accordingly. This can help you find the global minimum, especially if the objective function is convex.

Serial Code (C++)

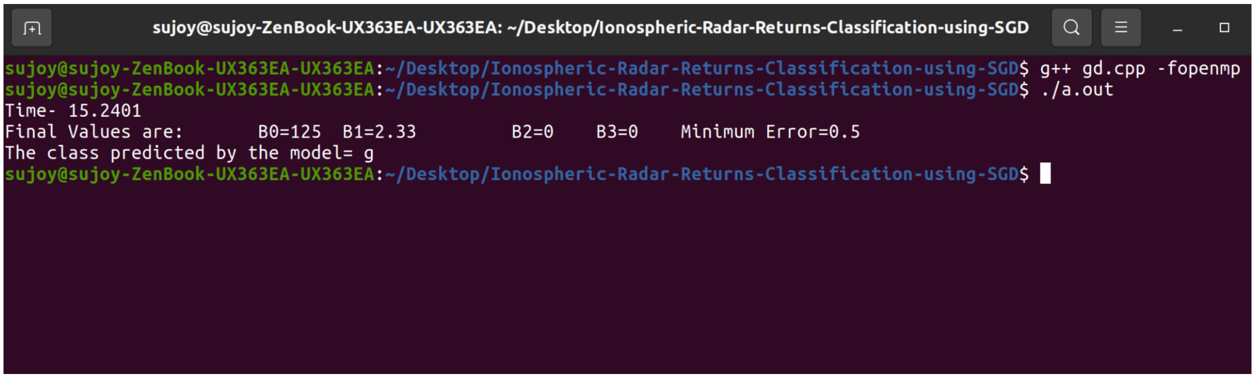
```
#include <bits/stdc++.h>
#include <iostream>
#include <string>
#include <iomanip>
#include <omp.h>
#include <sstream>
#include <fstream>
using namespace std;

//Variables for obtaining line of best fit
double b[4][10530]={};
//Swapping function
void swap(double *xp, double *yp)
{
    double temp = *xp;
    *xp = *yp;
    *yp = temp;
}


// A function to implement bubble sort
void bubbleSort(double arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
    //Absolute swapping mechanism
    for (j = 0; j < n-i-1; j++)
        if (abs(arr[j]) > abs(arr[j+1]))
            swap(&arr[j], &arr[j+1]);
}
```

```
//Training using the obtained data set
void train(double *x1,double *x2,double *x3,double *y)
{
    double start,end;
    int i,idx;double error[10530]; // for storing the error values
    //Since there are 351 values in our dataset and we want to run for 50 batches
so total for loop run 17550 times
    double err[10530]={0};          // for calculating error on each stage
    double alpha = 0.01; // initializing our learning rate
    double e = 2.718281828;
    double p[10530]={0},pred[10530]={0};

    start=omp_get_wtime();
    #pragma omp parallel for
    for (i = 0; i < 10530; i++)
    {
        idx = i % 10;//for accessing index after every batch
        p[i] = -(b[0][i] + b[1][i] * x1[idx] + b[2][i] * x2[idx] + b[3][i] *
x3[idx]);//making the prediction
        pred[i] = 1 / (1 + pow(e, p[i])); //calculating final prediction applying
sigmoid
        err[i] = y[idx] - pred[i]; //calculating the error
        for(int j=0;j<100000;j++)
        {
            b[0][i]=b[0][i] - alpha * err[i] * pred[i] * (1 - pred[i]) * 1.0;
//updating b0
            b[1][i]=b[1][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x1[idx];
//updating b1
            b[2][i]=b[2][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x2[idx];
//updating b2
            b[3][i]=b[3][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x3[idx];
//updating b3
        }
        //cout << "\tB0= " << b[0][i] << " " << "\t\tB1= " << b[1][i] << " " <<
"\t\tB2= " << b[2][i] << "\t\tB3= " << b[3][i] << "\t\tError=" << err << endl;
        error[i]=err[i];
    }
    bubbleSort(error,i);
    //custom sort based on absolute error difference
    end=omp_get_wtime();
    //Time Taken
    cout<<end-start<<endl;
    //cout << "Final Values are: " << "\tB0=" << b[0][10529] << " " << "\tB1=" <<
b[1][10529] << " " << "\tB2=" << b[2][10529] << "\tB3=" << b[3][10529] <<"\tMinimum
Error=" << abs(error[0])<<endl;
}
```

```
//Testing the trained Stochastic Model
void test(double test1, double test2, double test3)
{
    //make prediction
    double pred = b[0][10529] + b[1][10529] * test1 + b[2][10529] * test2 +
b[3][10529]*test3;
    char ch;

    //cout << "The value predicted by the model= " << pred << endl;
    if (pred > 0.5)
    {
        pred = 1;
        ch='g';
    }
    else
    {
        pred = 0;
        ch='b';
    }
    //cout << "The class predicted by the model= " << ch<<endl;
}

int main()
{
    //Input dataset arrays
    double x1[1053];
    double x2[1053];
    double x3[1053];
    double y[1053];

    //Reading the data file
    FILE* fp = fopen("ionosphere_data.csv", "r");
    char buffer[1024]; int i=0;
    int row = 0; int column = 0;
    while (fgets(buffer,1024, fp))
    {
       column = 0;
       row++;
       if (row == 1)
           continue;

       // Splitting the data
       char* value = strtok(buffer, ",");

       while (value)
       {
           // Column 1
```

```
            if (column == 0)
            {
                x1[i]=stod(value);
            }
            // Column 2
            if (column == 1)
            {
                x2[i]=stod(value);
            }
            // Column 3
            if (column ==2)
            {
                x3[i]=stod(value);
            }
            // Column 4
            if (column == 3)
            {
                 if (value=="g")
                 {
                     y[i]=1.0;
                 }
                 else
                 {
                     y[i]=0.0;
                 }
                 i++;
            }
            value = strtok(NULL, ",");
            column++;
        }
}

    //Close the file
    fclose(fp);


    //Training Phase
    train(x1, x2,x3, y);

    //Testing Phase
    double test1=0.35346, test2=0.69387, test3=0.68195;
    test(test1, test2, test3);

    return 0;
}
```

Output Snapshot-



Functional Profiling-

Profiling allows us to learn where the program spent its time and which functions called which other functions while it was executing. This information can show which pieces of the program are slower than expected, and might be candidates for rewriting to make the program execute faster. It can also tell which functions are being called more or less often than expected. This may help to spot bugs that had otherwise been unnoticed.

Since the profiler uses information collected during the actual execution of the program, it can be used on programs that are too large or too complex to analyze by reading the source. However, how the program is run will affect the information that shows up in the profile data. If we don't use some feature of the program while it is being profiled, no profile information will be generated for that feature.

Steps to profile-

Step 1. Enable profiling during compilation (use -pg option)
$ gcc -pg -o TestGprof TestGprof.c

Step 2. Execute the binary so that profiling data is generated
$ ./TestGprof

Step 3. If the profiling is enabled then on executing the program, file gmon.out will be generated.
$ ls
gmon.out TestGprof TestGprof.c

Step 4. Obtain the profiling results in a txt file
$ gprof ./a.out | grep -v std | grep -v static | grep -v cxx > analysis.txt

---

1. **Flat Profile**
   (Excluding the built-in STL functions)

% time	**Cumulative seconds**	**Self Seconds**	**Calls**	**Self ms/call**	**Total ns/call**	**Name**
1.79	0.26	0.01	1	260.72	501.38	bubbleSort(double*, int)
18.80	0.50	0.11	64557373	0.00	0.00	swap(double*, double*)
8.06	0.55	0.05	1	45.12	45.12	_GLOBAL__sub_I_b0
46.56	0.56	0.26	1	10.03	511.40	train(double*, double*, double*, double*)
0.90	0.56	0.01				frame_dummy
0.00	0.56	0.00	1	0.00	0.00	test(double, double, double)

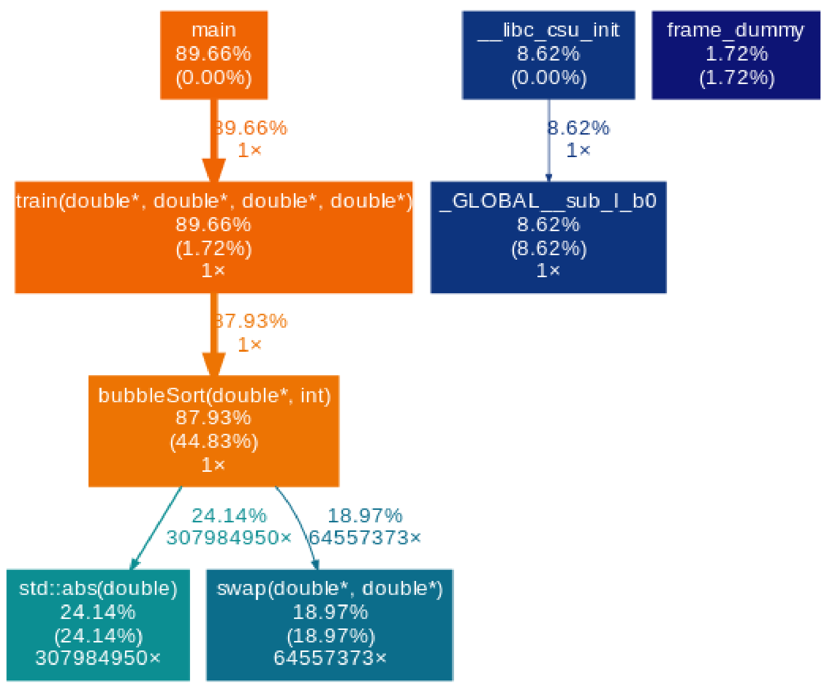
Terminology-

1. % time- the percentage of the total running time of the program used by this function.
2. Cumulative seconds- a running sum of the number of seconds accounted for by this function and those listed above it.
3. self seconds- the number of seconds accounted for by this function alone. This is the major sort for this listing.
4. Calls- the number of times this function was invoked, if this function is profiled, else blank.

5. self ms/call- the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
6. Total ms/call- the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.
7. Name- the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

2. Functional Graph Diagram-

It displays the executed code in a visual diagram where each node corresponds to a function or method and their relations show the code flow.



Line Profiling Output-

Line profiling is a process where we analyze time taken by various parts of our code for every line of the code. It can help us better understand the time/space complexity of our code.

```
sujoydatta@sujoydatta-VirtualBox:~/Desktop/HPC Project$ ls
analysis.txt  gd.cpp  gd.gcda  gd.gcno  gmon.out  gprof2dot.py  ionosphere_data.csv  main.gprof  output.svg  sgd
sujoydatta@sujoydatta-VirtualBox:~/Desktop/HPC Project$ gcov -b -c gd.cpp
File 'gd.cpp'
Lines executed:95.71% of 70
Branches executed:100.00% of 110
Taken at least once:60.00% of 110
Calls executed:91.55% of 71
Creating 'gd.cpp.gcov'
```

Profile-

```
        -:   38://Training using the obtained data set
function _Z5trainPdS_S_S_ called 1 returned 100% blocks executed 100%
        1:   39:void train(double *x1,double *x2,double *x3,double *y)
        -:   40:{
        -:   41:    double error[17550]; // for storing the error values
        -:   42:    double err;          // for calculating error on each stage
        1:   43:    double alpha = 0.01; // initializing our learning rate
        1:   44:    double e = 2.718281828;
        -:   45:
        -:   46:    /*Training Phase*/
    17551:   47:    for (int i = 0; i < 17550; i++)
branch  0 taken 17550 (fallthrough)
branch  1 taken 1
        -:   48:    { //Since there are 350 values in our dataset and we want to
run for 50 batches so total for loop run 17550 times
        -:   49:
        -:   50:        //for accessing index after every batch
    17550:   51:        int idx = i % 50;
        -:   52:
        -:   53:        //making the prediction
    17550:   54:        double p = -(b0 + b1 * x1[idx] + b2 * x2[idx] + b3 *
x3[idx]);
        -:   55:
        -:   56:        //calculating final prediction applying sigmoid
    17550:   57:        double pred = 1 / (1 + pow(e, p));
        -:   58:
    17550:   59:        err = y[idx] - pred; //calculating the error
        -:   60:
        -:   61:        //obtaining the line of best fit
    17550:   62:        b0 = b0 - alpha * err * pred * (1 - pred) * 1.0;
//updating b0
    17550:   63:        b1 = b1 + alpha * err * pred * (1 - pred) * x1[idx];
//updating b1
    17550:   64:        b2 = b2 + alpha * err * pred * (1 - pred) * x2[idx];
//updating b2
    17550:   65:        b3 = b3 + alpha * err * pred * (1 - pred) * x3[idx];
//updating b3
```

```
        -:   66:
        -:   67:
        -:   68:        //printing values for each training step
    17550:   69:        cout << "\tB0= " << b0 << " " << "\t\tB1= " << b1 << " " <<
"\t\tB2= " << b2 << "\t\tB3= " << b3 << "\t\tError=" << err << endl;
blocks executed 72%
```

Main Function-

```
        1:  102:int main()
        -:  103:{
        -:  104:    //Input dataset arrays
        -:  105:    double x1[351];
        -:  106:    double x2[351];
        -:  107:    double x3[351];
        -:  108:    double y[351];
        -:  109:
        -:  110:    //Reading the data file
        1:  111:    FILE* fp = fopen("ionosphere_data.csv", "r");
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        1:  112:    char buffer[1024]; int i=0;
        1:  113:    int row = 0; int column = 0;
      353:  114:    while (fgets(buffer,1024, fp))
call    0 returned 353
branch  1 taken 353 (fallthrough)
branch  2 taken 0 (throw)
branch  3 taken 352 (fallthrough)
branch  4 taken 1
        -:  115:    {
      352:  116:        column = 0;
      352:  117:        row++;
      352:  118:        if (row == 1)
branch  0 taken 1 (fallthrough)
branch  1 taken 351
        1:  119:            continue;
        -:  120:
        -:  121:        // Splitting the data
      351:  122:        char* value = strtok(buffer, ",");
call    0 returned 351
        -:  123:
     1755:  124:        while (value)
branch  0 taken 1404 (fallthrough)
branch  1 taken 351
        -:  125:        {
```

```
        -:  126:            // Column 1
     1404:  127:            if (column == 0)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
        -:  128:            {
      351:  129:                x1[i]=stod(value);
call    0 returned 351
call    1 returned 351
branch  2 taken 351 (fallthrough)
branch  3 taken 0 (throw)
call    4 returned 351
branch  5 taken 351 (fallthrough)
branch  6 taken 0 (throw)
call    7 returned 351
call    8 returned 351
call    9 never executed
call   10 never executed
        -:  130:            }
        -:  131:            // Column 2
     1404:  132:            if (column == 1)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
        -:  133:            {
      351:  134:                x2[i]=stod(value);
call    0 returned 351
call    1 returned 351
branch  2 taken 351 (fallthrough)
branch  3 taken 0 (throw)
call    4 returned 351
branch  5 taken 351 (fallthrough)
branch  6 taken 0 (throw)
call    7 returned 351
call    8 returned 351
call    9 never executed
call   10 never executed
        -:  135:            }
        -:  136:            // Column 3
     1404:  137:            if (column ==2)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
        -:  138:            {
      351:  139:                x3[i]=stod(value);
call    0 returned 351
call    1 returned 351
branch  2 taken 351 (fallthrough)
branch  3 taken 0 (throw)
call    4 returned 351
```

```
branch  5 taken 351 (fallthrough)
branch  6 taken 0 (throw)
call    7 returned 351
call    8 returned 351
call    9 never executed
call   10 never executed
        -:  140:            }
        -:  141:            // Column 4
     1404:  142:            if (column == 3)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
        -:  143:            {
      351:  144:                if (value=="g")
branch  0 taken 0 (fallthrough)
branch  1 taken 351
        -:  145:                {
    #####:  146:                    y[i]=1.0;
        -:  147:                }
        -:  148:                else
        -:  149:                {
      351:  150:                    y[i]=0.0;
        -:  151:                }
      351:  152:                i++;
        -:  153:            }
     1404:  154:            value = strtok(NULL, ",");
call    0 returned 1404
     1404:  155:            column++;
        -:  156:        }
        -:  157:}
        -:  158:
        -:  159:    //Close the file
        1:  160:    fclose(fp);
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        -:  161:
        -:  162:
        -:  163:    double start,end;
        1:  164:    start=omp_get_wtime();
call    0 returned 1
        -:  165:    //Training Phase
        1:  166:    train(x1, x2,x3, y);
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        1:  167:    end=omp_get_wtime();
call    0 returned 1
```

```
        -:  168:
        -:  169:    //Testing Phase
        1:  170:    double test1=0.5131, test2=-0.00015, test3=0.52099;
        1:  171:    test(test1, test2, test3);
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        -:  172:
        -:  173:    //Time Taken
        1:  174:    cout<<"Time "<<end-start<<" seconds"<<endl;
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
call    3 returned 1
branch  4 taken 1 (fallthrough)
branch  5 taken 0 (throw)
call    6 returned 1
branch  7 taken 1 (fallthrough)
branch  8 taken 0 (throw)
call    9 returned 1
branch 10 taken 1 (fallthrough)
branch 11 taken 0 (throw)
        1:  175:    return 0;
        -:  176:}
```

Processor Utilization Report-

1. **Hardware Profile of the System-**

   CPU name: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
   CPU type: Intel Coffee Lake processor
   Hardware Thread Topology
   Sockets: 1
   Cores per socket: 4
   Threads per core: 2

HWThread	Thread	Core	Socket	Available
0	0	0	0	*
1	0	1	0	*
2	0	2	0	*
3	0	3	0	*
4	1	0	0	*

5	1	1	0	*
6	1	2	0	*
7	1	3	0	*

Socket 0: (0 4 1 5 2 6 3 7)

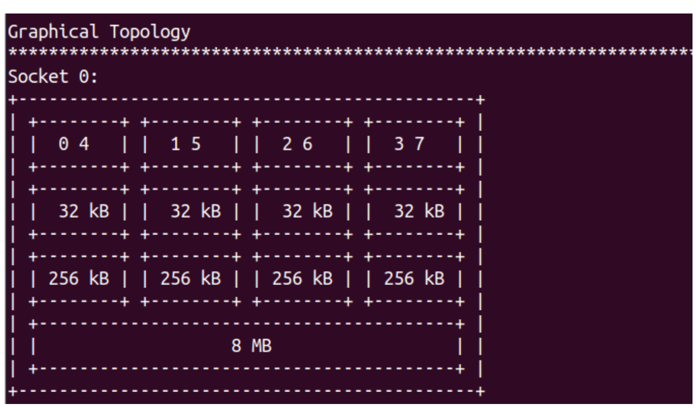
****
Cache Topology
Level: 1
Size: 32 kB
Cache groups: (0 4) (1 5) (2 6) (3 7)

Level: 2
Size: 256 kB
Cache groups: (0 4) (1 5) (2 6) (3 7)

Level: 3
Size: 8 MB
Cache groups: (0 4 1 5 2 6 3 7)

NUMA domains: 1

Domain: 0
Processors: (0 1 2 3 4 5 6 7)
Distances: 10
Free memory: 1349.73 MB
Total memory: 7766.82 MB



```
Graphical Topology
********************************************************************************
Socket 0:
+---------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ |
| |  0 4   | |  1 5   | |  2 6   | |  3 7   | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| |  32 kB | |  32 kB | |  32 kB | |  32 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| | 256 kB | | 256 kB | | 256 kB | | 256 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +-----------------------------------------+ |
| |                  8 MB                   | |
| +-----------------------------------------+ |
+---------------------------------------------+
```

2. Architectural Capability-

1. This architecture has 27 counters.
2. This architecture has 439 events.

```
This architecture has 27 counters.
Counter tags(name, type<, options>):
FIXC0, Fixed counters, KERNEL|ANYTHREAD
FIXC1, Fixed counters, KERNEL|ANYTHREAD
FIXC2, Fixed counters, KERNEL|ANYTHREAD
PMC0, Core-local general purpose counters,
```

```
This architecture has 439 events.
Event tags (tag, id, umask, counters<, options>):
TEMP_CORE, 0x0, 0x0, TMP0
PWR_PKG_ENERGY, 0x2, 0x0, PWR0
PWR_PP0_ENERGY, 0x1, 0x0, PWR1
PWR_PP1_ENERGY, 0x4, 0x0, PWR2
PWR_DRAM_ENERGY, 0x3, 0x0, PWR3
```

3. Monitoring Caches-

1. Case 1- L3 Cache [0-7 cores]

Group 1: L3

Event	Counter	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
INSTR_RETIRED_ANY	FIXC0	98305699554	81387712	103496606	31923816	24018154651	68597554	65555357	47036763
CPU_CLK_UNHALTED_CORE	FIXC1	30695796387	86328407	131835164	52459426	7478095382	83394893	66159656	64977378
CPU_CLK_UNHALTED_REF	FIXC2	20407955136	61207200	90802464	35598432	4994850048	56085024	45338592	45989568
L2_LINES_IN_ALL	PMC0	1198680	2825178	6534439	2083470	1074602	3012674	1277323	2254762
L2_TRANS_L2_WB	PMC1	927276	279294	587728	218016	240579	332654	210443	395596

Event	Counter	Sum	Min	Max	Avg
INSTR_RETIRED_ANY STAT	FIXC0	122721852013	31923816	98305699554	1.534023e+10
CPU_CLK_UNHALTED_CORE STAT	FIXC1	38659046693	52459426	30695796387	4.832381e+09
CPU_CLK_UNHALTED_REF STAT	FIXC2	25737826464	35598432	20407955136	3217228308
L2_LINES_IN_ALL STAT	PMC0	20261128	1074602	6534439	2532641
L2_TRANS_L2_WB STAT	PMC1	3191586	210443	927276	398948.2500

Metric	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Runtime (RDTSC) [s]	11.0604	11.0604	11.0604	11.0604	11.0604	11.0604	11.0604	11.0604
Runtime unhalted [s]	13.3251	0.0375	0.0572	0.0228	3.2462	0.0362	0.0287	0.0282
Clock [MHz]	3464.8840	3249.0807	3344.5902	3394.7043	3448.8780	3425.3254	3361.5107	3254.7088
CPI	0.3122	1.0607	1.2738	1.6433	0.3114	1.2157	1.0092	1.3814
L3 load bandwidth [MBytes/s]	6.9360	16.3476	37.8108	12.0558	6.2181	17.4325	7.3911	13.0469
L3 load data volume [GBytes]	0.0767	0.1808	0.4182	0.1333	0.0688	0.1928	0.0817	0.1443
L3 evict bandwidth [MBytes/s]	5.3656	1.6161	3.4008	1.2615	1.3921	1.9249	1.2177	2.2891
L3 evict data volume [GBytes]	0.0593	0.0179	0.0376	0.0140	0.0154	0.0213	0.0135	0.0253
L3 bandwidth [MBytes/s]	12.3016	17.9637	41.2116	13.3173	7.6101	19.3574	8.6088	15.3360
L3 data volume [GBytes]	0.1361	0.1987	0.4558	0.1473	0.0842	0.2141	0.0952	0.1696

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	88.4832	11.0604	11.0604	11.0604
Runtime unhalted [s] STAT	16.7819	0.0228	13.3251	2.0977
Clock [MHz] STAT	26943.6821	3249.0807	3464.8840	3367.9603
CPI STAT	8.2077	0.3114	1.6433	1.0260
L3 load bandwidth [MBytes/s] STAT	117.2388	6.2181	37.8108	14.6548
L3 load data volume [GBytes] STAT	1.2966	0.0688	0.4182	0.1621
L3 evict bandwidth [MBytes/s] STAT	18.4678	1.2177	5.3656	2.3085
L3 evict data volume [GBytes] STAT	0.2043	0.0135	0.0593	0.0255
L3 bandwidth [MBytes/s] STAT	135.7065	7.6101	41.2116	16.9633
L3 data volume [GBytes] STAT	1.5010	0.0842	0.4558	0.1876

2. **Case 2- L2 Cache [0-7 cores]**

Group 1: L2

Event	Counter	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
INSTR_RETIRED_ANY	FIXC0	898218	97540563440	13285243	6175188	107985589	24813082183	12533164	18170945
CPU_CLK_UNHALTED_CORE	FIXC1	3361203	30602293807	25842985	9898091	107403316	7774181003	19864119	23284273
CPU_CLK_UNHALTED_REF	FIXC2	2273952	19132067328	16276128	6284640	67615776	4901264928	12335424	14722176
L1D_REPLACEMENT	PMC0	38745	18012010	326042	85085	1050273	893757	232385	248648
L1D_M_EVICT	PMC1	7875	14994694	90568	21679	313097	118488	58680	61454
ICACHE_64B_IFTAG_MISS	PMC2	124454	87896	1423383	423592	3468914	382606	800952	863575

Event	Counter	Sum	Min	Max	Avg
INSTR_RETIRED_ANY STAT	FIXC0	122512693970	898218	97540563440	1.531409e+10
CPU_CLK_UNHALTED_CORE STAT	FIXC1	38566128797	3361203	30602293807	4.820766e+09
CPU_CLK_UNHALTED_REF STAT	FIXC2	24152840352	2273952	19132067328	3019105044
L1D_REPLACEMENT STAT	PMC0	20886945	38745	18012010	2.610868e+06
L1D_M_EVICT STAT	PMC1	15666535	7875	14994694	1.958317e+06
ICACHE_64B_IFTAG_MISS STAT	PMC2	7575372	87896	3468914	946921.5000

Metric	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Runtime (RDTSC) [s]	10.4595	10.4595	10.4595	10.4595	10.4595	10.4595	10.4595	10.4595
Runtime unhalted [s]	0.0015	13.2821	0.0112	0.0043	0.0466	3.3742	0.0086	0.0101
Clock [MHz]	3405.6456	3685.3445	3658.2854	3628.7500	3659.7858	3654.5380	3710.2334	3643.9859
CPI	3.7421	0.3137	1.9452	1.6029	0.9946	0.3133	1.5849	1.2814
L2D load bandwidth [MBytes/s]	0.2371	110.2128	1.9950	0.5206	6.4265	5.4688	1.4219	1.5214
L2D load data volume [GBytes]	0.0025	1.1528	0.0209	0.0054	0.0672	0.0572	0.0149	0.0159
L2D evict bandwidth [MBytes/s]	0.0482	91.7503	0.5542	0.1327	1.9158	0.7250	0.3591	0.3760
L2D evict data volume [GBytes]	0.0005	0.9597	0.0058	0.0014	0.0200	0.0076	0.0038	0.0039
L2 bandwidth [MBytes/s]	1.0468	202.5008	11.2586	3.2452	29.5680	8.5349	6.6819	7.1816
L2 data volume [GBytes]	0.0109	2.1181	0.1178	0.0339	0.3093	0.0893	0.0699	0.0751

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	83.6760	10.4595	10.4595	10.4595
Runtime unhalted [s] STAT	16.7386	0.0015	13.2821	2.0923
Clock [MHz] STAT	29046.5686	3405.6456	3710.2334	3630.8211
CPI STAT	11.7781	0.3133	3.7421	1.4723
L2D load bandwidth [MBytes/s] STAT	127.8041	0.2371	110.2128	15.9755
L2D load data volume [GBytes] STAT	1.3368	0.0025	1.1528	0.1671
L2D evict bandwidth [MBytes/s] STAT	95.8613	0.0482	91.7503	11.9827
L2D evict data volume [GBytes] STAT	1.0027	0.0005	0.9597	0.1253
L2 bandwidth [MBytes/s] STAT	270.0178	1.0468	202.5008	33.7522
L2 data volume [GBytes] STAT	2.8243	0.0109	2.1181	0.3530

3. **Case 2- FLOPS_DP [0-7 cores]**

Group 1: FLOPS_DP

Event	Counter	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
INSTR_RETIRED_ANY	FIXC0	1155327425	1430635557	86129506027	1745284423	1634056837	1508731802	37329706220	1689094437
CPU_CLK_UNHALTED_CORE	FIXC1	1079640028	1311164075	27233076199	1462686568	1345025129	1238310475	12688213878	1396950716
CPU_CLK_UNHALTED_REF	FIXC2	824874336	964091520	18740633472	1108744224	1027159008	952072896	9027135168	1065090912
FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE	PMC0	2707168	2579641	450974	2759279	2271787	3620891	1028062	2377571
FP_ARITH_INST_RETIRED_SCALAR_DOUBLE	PMC1	16513866	20444664	30360707645	21552740	20970458	19150283	10028820694	25864951
FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE	PMC2	51	55	10	150	87	41	22	55

Event	Counter	Sum	Min	Max	Avg
INSTR_RETIRED_ANY STAT	FIXC0	132622342728	1155327425	86129506027	16577792841
CPU_CLK_UNHALTED_CORE STAT	FIXC1	47755067068	1079640028	27233076199	5.969383e+09
CPU_CLK_UNHALTED_REF STAT	FIXC2	33709801536	824874336	18740633472	4213725192
FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE STAT	PMC0	17795373	450974	3620891	2.224422e+06
FP_ARITH_INST_RETIRED_SCALAR_DOUBLE STAT	PMC1	40514025301	16513866	30360707645	5.064253e+09
FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE STAT	PMC2	471	10	150	58.8750

Metric	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Runtime (RDTSC) [s]	11.7585	11.7585	11.7585	11.7585	11.7585	11.7585	11.7585	11.7585
Runtime unhalted [s]	0.4686	0.5691	11.8194	0.6348	0.5838	0.5374	5.5068	0.6063
Clock [MHz]	3015.7183	3133.5625	3348.2048	3039.6214	3017.1181	2996.8081	3238.5464	3021.9967
CPI	0.9345	0.9165	0.3162	0.8381	0.8231	0.8208	0.3399	0.8270
DP [MFLOP/s]	1.8649	2.1775	2582.1073	2.3023	2.1699	2.2445	853.0773	2.6041
AVX DP [MFLOP/s]	1.734921e-05	1.870993e-05	3.401805e-06	0.0001	2.959571e-05	1.394740e-05	7.483972e-06	1.870993e-05
Packed [MUOPS/s]	0.2302	0.2194	0.0384	0.2347	0.1932	0.3079	0.0874	0.2022
Scalar [MUOPS/s]	1.4044	1.7387	2582.0306	1.8330	1.7834	1.6286	852.9024	2.1997
Vectorization ratio	14.0846	11.2042	0.0015	11.3500	9.7747	15.9014	0.0103	8.4186

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	94.0680	11.7585	11.7585	11.7585
Runtime unhalted [s] STAT	20.7262	0.4686	11.8194	2.5908
Clock [MHz] STAT	24811.5763	2996.8081	3348.2048	3101.4470
CPI STAT	5.8161	0.3162	0.9345	0.7270
DP [MFLOP/s] STAT	3448.5478	1.8649	2582.1073	431.0685
AVX DP [MFLOP/s] STAT	0.0002	3.401805e-06	0.0001	2.614974e-05
Packed [MUOPS/s] STAT	1.5134	0.0384	0.3079	0.1892
Scalar [MUOPS/s] STAT	3445.5208	1.4044	2582.0306	430.6901
Vectorization ratio STAT	70.7453	0.0015	15.9014	8.8432

Inferences-

1. **Extent of Parallelism**

Task parallelism (also known as function parallelism and control parallelism) is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing tasks—concurrently performed by processes or threads—across different processors. In contrast to data parallelism which involves running the same task on different components of data, task parallelism is distinguished by running many different tasks at the same time on the same data.

A common type of task parallelism is multi-threading which consists of moving a single set of data through a series of separate tasks where each task can execute independently of the others.

In the above serial code we look into the following aspects-

a. Task parallelism- choosing the task with the maximum wall clock serial time. Task parallelism emphasizes the distributed (parallelized) nature of the processing (i.e. threads), as opposed to the data (data parallelism). Most real programs fall somewhere on a continuum between task parallelism and data parallelism.

   In the training function above, a considerable distributed nature can be observed.

Critical Code to parallelise-

```
 for (i = 0; i < 10530; i++)
   {
       idx = i % 10;//for accessing index after every batch
       p[i] = -(b[0][i] + b[1][i] * x1[idx] + b[2][i] * x2[idx] + b[3][i] * x3[idx]);//making
the prediction
       pred[i] = 1 / (1 + pow(e, p[i])); //calculating final prediction applying sigmoid
       err[i] = y[idx] - pred[i]; //calculating the error
       for(int j=0;j<100000;j++)
       {
          b[0][i]=b[0][i] - alpha * err[i] * pred[i] * (1 - pred[i]) * 1.0;     //updating b0
          b[1][i]=b[1][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x1[idx]; //updating b1
          b[2][i]=b[2][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x2[idx]; //updating b2
          b[3][i]=b[3][i] + alpha * err[i] * pred[i] * (1 - pred[i]) * x3[idx]; //updating b3
       }
       //cout << "\tB0= " << b[0][i] << " " << "\t\tB1= " << b[1][i] << " " << "\t\tB2= "
<< b[2][i] << "\t\tB3= " << b[3][i] << "\t\tError=" << err << endl;
       error[i]=err[i];
   }
```

b. Motivational Pseudocode to push parallelism-

```
program:
...
if CPU = "a" then
   do task "A"
else if CPU="b" then
   do task "B"
end if
...
end program
```

2. **Functional modularity**

   Profiling is normally useful only after we've detected that something is wrong with our program. However, it could still be performed before that even happens to catch possible unseen bugs, which would, in turn, help chip away the time spent debugging the application at a later stage.

   Observations-

   a. On observing the **total ms/call** value for all the rows in the **flat profile**, it is quite evident that the most amount of time spent is on this functional module. The thread level parallelism implemented on this function can assist in saving the corresponding serial execution time frame.

46.56	0.56	0.26	1	10.03	511.40	train(double*, double*, double*, double*)

   b. **Line Profiling** can help avoid that **crash and burn outcome**, since it provides a fairly accurate view of what our program is doing(line-by-line), no matter the load. So, if we profile it with a very light load, and the result is that we're spending 80 percent of our time doing some kind of I/O operation, it might raise a flag for us.

   Code piece-

```
17550:   62:      b0 = b0 - alpha * err * pred * (1 - pred) * 1.0;    //updating b0
17550:   63:      b1 = b1 + alpha * err * pred * (1 - pred) * x1[idx]; //updating b1
17550:   64:      b2 = b2 + alpha * err * pred * (1 - pred) * x2[idx]; //updating b2
17550:   65:      b3 = b3 + alpha * err * pred * (1 - pred) * x3[idx]; //updating b3
```

3. **Processor Utilisation-**

   The least upper bound imposed upon processor utilization by the requirement for real-time guaranteed service can approach in big order of magnitudes for large task sets. It is desirable to find ways to improve this situation, since the practical costs of switching between tasks must still be counted.

   a. Cache comparison- All processors rely on L1 cache, this is usually located on the die of the processor and is very fast memory (and expensive). L2 cache is slower, bigger and cheaper than L1 cache.

Older processors used L2 cache on the motherboard, nowadays it tends to be built into the processor. L3 cache is slower, bigger and cheaper than L2 cache.

b. Many processes have known functions and durations(**CPI_STATS**) and can be given a permanent, fixed priority. In process control applications, where the duration and function of each process is known at design time, it may be appropriate to assign processes fixed priorities and place them in a static order. In this case, the scheduler begins at the top of the queue of processes and searches down for the first runnable process.

c. In designing schedulers for cache blocks and processors, it is important to determine the desirable criteria for a scheduler. The low-level scheduler basically determines which task has the highest priority and then selects that task for execution; it also maintains the ready queue.

Parallelising SGD

The preprocessor directive #pragma is used to provide the additional information to the compiler in C/C++ language. This is used by the compiler to provide some special features.

Approach-

Based on the inferences derived from profiling outputs and obtaining the critical section of the code, it is indeed important to parallelise the section using the pragma directive.

Code(C++)

```
#include <bits/stdc++.h>
#include <iostream>
#include <string>
#include <iomanip>
#include <omp.h>
#include <sstream>
#include <fstream>
using namespace std;

//Variables for obtaining line of best fit
```

```
double b[4][10530]={};
//Swapping function
void swap(double *xp, double *yp)
{
    double temp = *xp;
    *xp = *yp;
    *yp = temp;
}


// A function to implement bubble sort
void bubbleSort(double arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
    //Absolute swapping mechanism
    for (j = 0; j < n-i-1; j++)
        if (abs(arr[j]) > abs(arr[j+1]))
            swap(&arr[j], &arr[j+1]);
}


//Training using the obtained data set
void train(double *x1,double *x2,double *x3,double *y)
{
    double start,end;
    int i,idx;double error[10530]; // for storing the error values
    //Since there are 351 values in our dataset and we want to run
for 50 batches so total for loop run 17550 times
    double err[10530]={0};          // for calculating error on each
stage
    double alpha = 0.01; // initializing our learning rate
    double e = 2.718281828;
    double p[10530]={0},pred[10530]={0};

    start=omp_get_wtime();
    #pragma omp parallel for
    for (i = 0; i < 10530; i++)
    {
```

```
        idx = i % 10;//for accessing index after every batch
        p[i] = -(b[0][i] + b[1][i] * x1[idx] + b[2][i] * x2[idx] +
b[3][i] * x3[idx]);//making the prediction
        pred[i] = 1 / (1 + pow(e, p[i])); //calculating final
prediction applying sigmoid
        err[i] = y[idx] - pred[i]; //calculating the error
        for(int j=0;j<100000;j++)
        {
            b[0][i]=b[0][i] - alpha * err[i] * pred[i] * (1 -
pred[i]) * 1.0;     //updating b0
            b[1][i]=b[1][i] + alpha * err[i] * pred[i] * (1 -
pred[i]) * x1[idx]; //updating b1
            b[2][i]=b[2][i] + alpha * err[i] * pred[i] * (1 -
pred[i]) * x2[idx]; //updating b2
            b[3][i]=b[3][i] + alpha * err[i] * pred[i] * (1 -
pred[i]) * x3[idx]; //updating b3
        }
        //cout << "\tB0= " << b[0][i] << " " << "\t\tB1= " << b[1][i]
<< " " << "\t\tB2= " << b[2][i] << "\t\tB3= " << b[3][i] <<
"\t\tError=" << err << endl;
        error[i]=err[i];
    }
    bubbleSort(error,i);
    //custom sort based on absolute error difference
    end=omp_get_wtime();
    //Time Taken
    cout<<end-start<<endl;
    //cout << "Final Values are: " << "\tB0=" << b[0][10529] << " "
<< "\tB1=" << b[1][10529] << " " << "\tB2=" << b[2][10529] << "\tB3="
<< b[3][10529] <<"\tMinimum Error=" << abs(error[0])<<endl;
}

//Testing the trained Stochastic Model
void test(double test1, double test2, double test3)
{
    //make prediction
    double pred = b[0][10529] + b[1][10529] * test1 + b[2][10529] *
test2 + b[3][10529]*test3;
    char ch;
```

```
    //cout << "The value predicted by the model= " << pred << endl;
    if (pred > 0.5)
    {
        pred = 1;
        ch='g';
    }
    else
    {
        pred = 0;
        ch='b';
    }
    //cout << "The class predicted by the model= " << ch<<endl;
}

int main()
{
    //Input dataset arrays
    double x1[1053];
    double x2[1053];
    double x3[1053];
    double y[1053];

    //Reading the data file
    FILE* fp = fopen("ionosphere_data.csv", "r");
    char buffer[1024]; int i=0;
    int row = 0; int column = 0;
    while (fgets(buffer,1024, fp))
    {
       column = 0;
       row++;
       if (row == 1)
           continue;

       // Splitting the data
       char* value = strtok(buffer, ",");

       while (value)
       {
```

```
            // Column 1
            if (column == 0)
            {
                x1[i]=stod(value);
            }
            // Column 2
            if (column == 1)
            {
                x2[i]=stod(value);
            }
            // Column 3
            if (column ==2)
            {
                x3[i]=stod(value);
            }
            // Column 4
            if (column == 3)
            {
                 if (value=="g")
                 {
                     y[i]=1.0;
                 }
                 else
                 {
                     y[i]=0.0;
                 }
                 i++;
            }
            value = strtok(NULL, ",");
            column++;
        }
}

    //Close the file
    fclose(fp);


    //Training Phase
    train(x1, x2,x3, y);
```

```
    //Testing Phase
    double test1=0.35346, test2=0.69387, test3=0.68195;
    test(test1, test2, test3);


    return 0;
}
```

Observational Tabulation

# Threads	Execution Time	Speed Up	Parallelization Fraction
1	13.78	1	
2	8.8952	1.54915010342657	70.8969521044993
4	5.7365	2.40216159679247	77.8277697145622
6	5.6258	2.44942941448327	71.0089985486212
8	**4.44**	**3.1036036036036**	**77.4621604810284**
10	5.1394	2.68124683815231	69.6710208030963
12	4.5685	3.01630732187808	72.9238685842459
16	5.0275	2.74092491297862	67.7503628447025
32	5.2587	2.62041949531253	63.8329509808512
64	4.4976	3.0638562789043	68.4306217891124
128	5.0651	2.72057807348325	63.741083162863
150	4.7076	2.92718157872377	66.2793076241221

Theoretical Background

1. Speed up can be found using the following formula,

   S(n)=T(1)/T(n)

   where, S(n) = Speedup for thread count 'n'

   T(1) = Execution Time for Thread count '1' (serial code)

   T(n) = Execution Time for Thread count 'n' (serial code)

2. Parallelization Fraction can be found using the following formula,

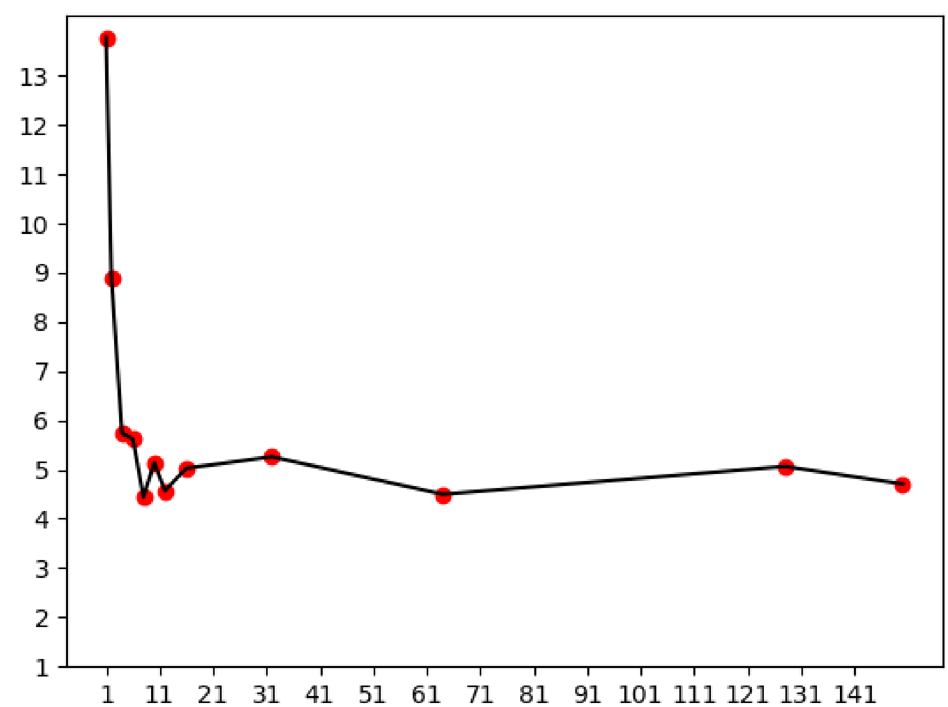
   S(n)=1/((1 - p) + p/n)

   where, S(n) = Speedup for thread count 'n'

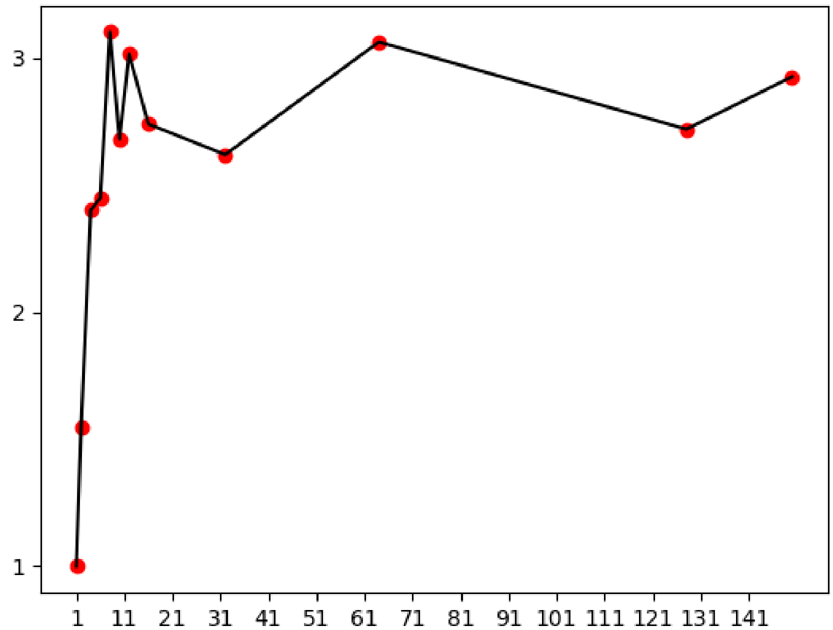
   n = Number of threads

   p = Parallelization fraction

Graphical Analysis



 # Threads vs Execution Time



#Threads vs Speed-up

Inferences-

- At thread count 8 maximum speedup is observed as the maximum number of parallel threads supported by the hardware is 8.
- If thread count is more than 8 then the execution time increases slightly and tapers out after 32 threads.

References-

1. https://www.appliedaicourse.com/ Includes a better overview of the larger problem statement in thought.
2. UCI Machine Learning Repository: Ionosphere Data Set- This radar data was collected by a system in Goose Bay, Labrador. This system consists of a phased array of 16 high-frequency antennas with a total transmitted power on the order of 6.4 kilowatts.

3. https://www.jhuapl.edu/Content/techdigest/pdf/V10-N03/10-03-Sigillito_Class.pdf - Distributed Stochastic Neighborhood Embedding

---