HPSC_Project_Report

December 15, 2023

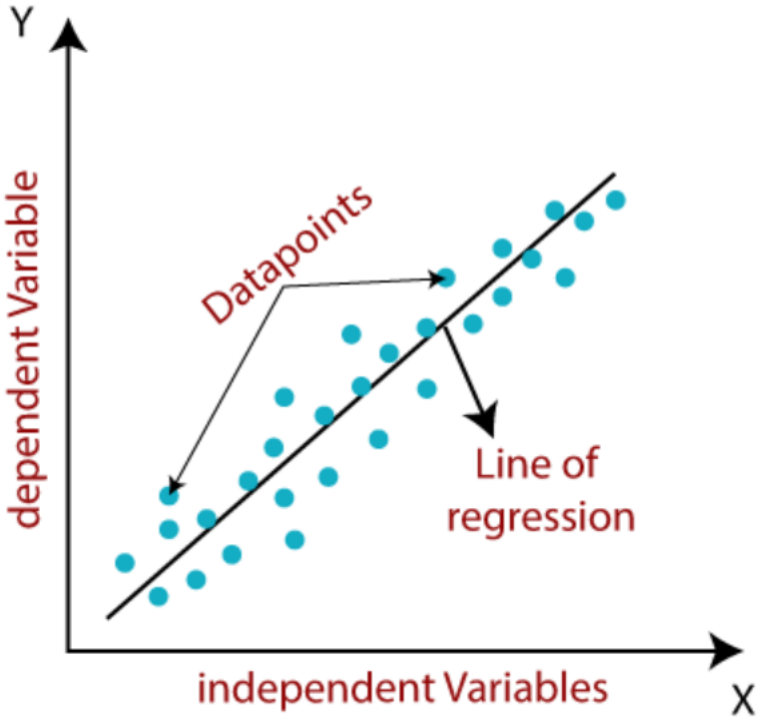
1 Parallel Implementation of the Standard Gradient Descent Algorithm to Optimize a Linear Regression Model Using C Language.

Group: Michael Soricelli, Vaishnavi Paineni, Mythri Krpet Jayaram

2 Introduction to Linear Regression Algorithm

Linear Regression is one of the simplest Machine Learning algorithm and it uses statistical methods to perform predictive analysis. It makes predictions for continous or numeric variables. This algorithm shows the linear relationship between two variables, a dependent variable x and one or more independent variables such as y. This means it shows how the value of the dependent variable x changes according to the value of the independent variable y.

The linear regression model provides a sloped straight line representing the relation between the variable. This can be represented as follows:



This can be mathematically expressed as:

$$y = a_0 + a_1 x + \epsilon \tag{1}$$

where, y= Dependent Variable (Target Variable) x= Independent Variable (predictor Variable) a_0= intercept of the line (Gives an additional degree of freedom) a_1 = Linear regression coefficient (scale factor to each input value). ϵ = random error

The values for x and y variables are training datasets for Linear Regression model representation

2.0.1 What is a Cost Function?

- The different values for weights or coefficient of lines (a_0, a_1) gives the different line of regression , and the cost function is used to estimate the values of the coefficient for the best fit line.
- The cost function optimizes the regression coefficients or weights. It measures how a linear regression model is performing.
- This cost function is used to find the accuracy of the mapping function, which maps the input variable to the output variable. This mapping function is also known as the Hypothesis Function.

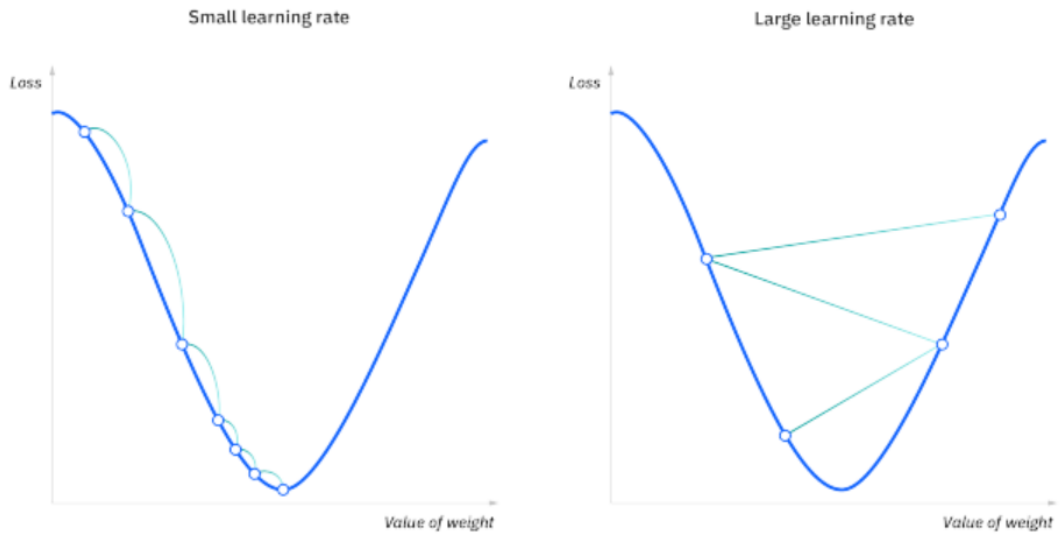
For the linear regression model, we used the mean squared error cost function which can be represented as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(actual - predicted)^2 \tag{2}$$

3 Introduction & Working of the Gradient Descent Algorithm

Gradient Descent is a widely used optimization algorithm to train machine learning models by the means of minimizing errors between the anctual and expected results. Apart from this, the gradient descent algorithm is also used in neural networks. Similar to finding the line of best fit in linear regression, the goal of gradient descent is to minimize the cost function, or the error between predicted and actual y. In order to do this, it requires two data points—a direction and a learning rate. These factors determine the partial derivative calculations of future iterations, allowing it to gradually arrive at the local or global minimum (i.e. point of convergence).

- Learning rate (also referred to as step size or the alpha) is the size of the steps that are taken to reach the minimum. This is typically a small value, and it is evaluated and updated based on the behavior of the cost function.
- The cost (or loss) function measures the difference, or error, between actual y and predicted y at its current position. This improves the machine learning model's efficacy by providing feedback to the model so that it can adjust the parameters to minimize the error and find the local or global minimum. It continuously iterates, moving along the direction of steepest descent (or the negative gradient) until the cost function is close to or at zero. At this point, the model will stop learning.



4 Part 1: Serial Code Implementation(serial.c)

For this project we have decided to parallelize a Gradient Descent Algorithm to optimize the mean squared error cost function for a linear regression model within a C program. The task of the C program is to take a set of data and fit a linear regression line to the data by using a standard

Gradient descent algorithm. For the sake of this project, we are creating dummy data by setting values to random doubles from 0 to 1 and adding them together to get the "target" value in the data. After running gradient descent with this approach, all the weights should converge to 1. We did this so that we could easily identify whether or not our parallel code was correctly running later on.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
double prediction(double *X, double *W, double B, int dim)
{
  double sum = 0.0;
  //calculate f(x_i) = W_1*X_1 + W_2*X_2 .... W_i*X_i + B
  for(int i = 0; i < dim; i++)
  {
    sum += X[i] * W[i];
  }
  sum += B;
  return sum;
}
//caclulate derivate of Mean squared error loss function with respect to W_i
double derivative(int N, int M, double *W, double B, int w_i, double X[N][M], double Y[N]) {
  double dldw = 0.0;
  for(int i = 0; i < N; i++) {
    //extract row i from X
    double row[M];
    for(int j = 0; j < M; j++) {
      row[j] = X[i][j];
    }
    double pred = prediction(row, W, B, M);
    dldw += -2 * row[w_i] * (Y[i] - pred);
  }
  return dldw/N;
}
int main(int argc, char **argv)
{
  srand(time(NULL));
  const double LEARNING_RATE = 0.001;
  //inititalize global variables, n(size of training data), m(number of features)
  //initialize X(feature data array [n X m]), Y(target data array[N x 1], W(weights for linear
  int N = atoi(argv[1]);
  int M = atoi(argv[2]);
  int iterations = atoi(argv[3]);
  double Y[N];
  double X[N][M];
  double W[M];
  for(int i = 0; i < M; i++)
```

```
  {
    W[i] = 0.0;
  }
  double B = 0.0;
  //intialize training data
  for(int i = 0; i < N; i++)
  {
    Y[i] = 0.0;
    for(int j = 0; j < M; j++)
    {
        X[i][j] = ((double) rand())/((double) RAND_MAX);
        //printf("training data point = %e\n", trainingData[i][j]);
    Y[i] += X[i][j];
    }
    //Y[i] = ((double) rand())/((double) RAND_MAX);
    //printf("target = %e\n", Y[i]);
  }
  //run gradient descent for x iterations
  for(int i = 0; i < iterations; i++) {
    //double pred = prediction(X, W, B, M);
    for(int i = 0; i < M; i++) {
      double dldw_i = derivative(N, M, W, B, i, X, Y);
      //update Weight with derivative
      W[i] = W[i] - LEARNING_RATE * dldw_i;
    }
  }
  for(int i = 0; i < M; i++) {
    printf("Weight %i is %e\n", i, W[i]);
  }
  return 0;
}
```

Output for the Serial Code:

```
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$ vim serial.c
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$ gcc -Wall -std=c99 -o serial serial.c
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$ time ./serial 1000 4 100000
Weight 0 is 1.000000e+00
Weight 1 is 1.000000e+00
Weight 2 is 1.000000e+00
Weight 3 is 1.000000e+00

real    0m10.965s
user    0m10.956s
sys     0m0.001s
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$
```

5 Part 2: Parallel Code Implementation(parallel.c)

To parallelize our code we created a parallel region on the for loop for training iterations where we run the gradient descent. We utilized OMP to create the parallel region and in the output we can see the time for execution has reduced compared to the serial code.

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>
double prediction(double *X, double *W, double B, int dim)
{
  double sum = 0.0;
  //calculate f(x_i) = W_1*X_1 + W_2*X_2 .... W_i*X_i + B
  for(int i = 0; i < dim; i++)
  {
    sum += X[i] * W[i];
  }
  sum += B;
  return sum;
}
//caclulate derivate of Mean squared error loss function with respect to W_i
double derivative(int N, int M, double *W, double B, int w_i, double X[N][M], double Y[N]) {
  double dldw = 0.0;
  for(int i = 0; i < N; i++) {
    //extract row i from X
    double row[M];
    for(int j = 0; j < M; j++) {
      row[j] = X[i][j];
    }
    double pred = prediction(row, W, B, M);
    dldw += -2 * row[w_i] * (Y[i] - pred);
  }
  return dldw/N;
}
int main(int argc, char **argv)
{
  srand(time(NULL));
  double start = omp_get_wtime();
  const double LEARNING_RATE = 0.001;
  //inititalize global variables, n(size of training data), m(number of features),
  //initialize X(feature data array [n X m]), Y(target data array[N x 1], W(weights for linear
  int N = atoi(argv[1]);
  int M = atoi(argv[2]);
  int iterations = atoi(argv[3]);
  int threads = atoi(argv[4]); //number of threads
  omp_set_num_threads(threads);
  double Y[N];
  double X[N][M];
  double W[M];
  for(int i = 0; i < M; i++)
  {
    W[i] = 0.0;
```

```
  }
  double B = 0.0;
  double dldw_i;
  //intialize training data
  for(int i = 0; i < N; i++)
  {
    Y[i] = 0.0;
    for(int j = 0; j < M; j++)
    {
        X[i][j] = ((double) rand())/((double) RAND_MAX);
        //printf("training data point = %e\n", trainingData[i][j]);
    Y[i] += X[i][j];
    }
    //Y[i] = ((double) rand())/((double) RAND_MAX);
    //printf("target = %e\n", Y[i]);
  }
  //run gradient descent for x iterations
#pragma omp parallel for private(dldw_i)
    for(int i = 0; i < iterations; i++) {
    //double pred = prediction(X, W, B, M);
    for(int j = 0; j < M; j++) {
            dldw_i = derivative(N, M, W, B, j, X, Y);
            //update Weight with derivative
            W[j] = W[j] - LEARNING_RATE * dldw_i;
    }
  }
  for(int i = 0; i < M; i++) {
    printf("Weight %i is %e\n", i, W[i]);
  }

  //printf("%e\n", omp_get_wtime() - start);
  return 0;
}
```

Output for the Parallel Code:

```
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$ vim parallel.c
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$ gcc -Wall -std=c99 -fopenmp -o parallel parall
el.c
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$ time ./parallel 1000 4 100000
Weight 0 is 1.000000e+00
Weight 1 is 1.000000e+00
Weight 2 is 1.000000e+00
Weight 3 is 1.000000e+00

real    0m2.510s
user    0m19.846s
sys     0m0.001s
(base) vaishpai@Vaish:/mnt/c/users/vaish/desktop/dsc520-2023-vaishnavipaineni/project$
```

6 Part 3: Running the Parallel Code on Anvil to check Scaling and Speed-up

To check the performance of our parallel implementation of gradient descent, we utilized up to 128 cores and 512 cores on Anvil for this project. On Anvil we performed strong and weak scaling tests for our code. When running the parallelized code, we got our expected output and we saw speedup when we ran the program with more cores, the speed up gains would taper off as the amount of cores grew in size.

6.0.1 Parallel Code for Anvil(parallel_anvil.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>
double prediction(double *X, double *W, double B, int dim)
{
  double sum = 0.0;
  //calculate f(x_i) = W_1*X_1 + W_2*X_2 .... W_i*X_i + B
  for(int i = 0; i < dim; i++)
  {
    sum += X[i] * W[i];
  }
  sum += B;
  return sum;
}
//caclulate derivate of Mean squared error loss function with respect to W_i
double derivative(int N, int M, double *W, double B, int w_i, double X[N][M], double Y[N]) {
  double dldw = 0.0;
  for(int i = 0; i < N; i++) {
    //extract row i from X
    double row[M];
    for(int j = 0; j < M; j++) {
      row[j] = X[i][j];
    }
    double pred = prediction(row, W, B, M);
    dldw += -2 * row[w_i] * (Y[i] - pred);
  }
  return dldw/N;
}
int main(int argc, char **argv)
{
  srand(time(NULL));
  //double start = omp_get_wtime();
  const double LEARNING_RATE = 0.001;
  //inititalize global variables, n(size of training data), m(number of features),
  //initialize X(feature data array [n X m]), Y(target data array[N x 1], W(weights for linear
```

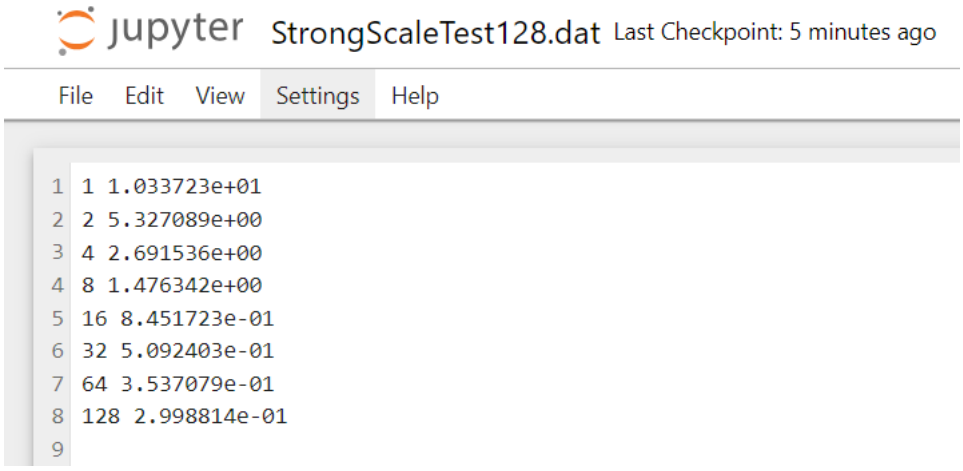
```
  int N = atoi(argv[1]);
  int M = atoi(argv[2]);
  int iterations = atoi(argv[3]);
  int threads = atoi(argv[4]); //number of threads
  omp_set_num_threads(threads);
  double Y[N];
  double X[N][M];
  double W[M];
  for(int i = 0; i < M; i++)
  {
    W[i] = 0.0;
  }
  double B = 0.0;
  double dldw_i;
  //intialize training data
  for(int i = 0; i < N; i++)
  {
    Y[i] = 0.0;
    for(int j = 0; j < M; j++)
    {
        X[i][j] = ((double) rand())/((double) RAND_MAX);
        //printf("training data point = %e\n", trainingData[i][j]);
    Y[i] += X[i][j];
    }
    //Y[i] = ((double) rand())/((double) RAND_MAX);
    //printf("target = %e\n", Y[i]);
  }
  //run gradient descent for x iterations
#pragma omp parallel for private(dldw_i)
    for(int i = 0; i < iterations; i++) {
    //double pred = prediction(X, W, B, M);
    for(int j = 0; j < M; j++) {
            dldw_i = derivative(N, M, W, B, j, X, Y);
            //update Weight with derivative
            W[j] = W[j] - LEARNING_RATE * dldw_i;
    }
  }
/*  for(int i = 0; i < M; i++) {
    printf("Weight %i is %e\n", i, W[i]);
  }
*/
  printf("%e\n", omp_get_wtime() - start);
  return 0;
}
```

6.0.2 Anvil Job Script for Strong Scale Testing(128 cores) (anvilstrong128.job)

```
#!/bin/bash
```

```
#-------------------------------------------------
#SBATCH -J myjob # Job name
#SBATCH -o myjob.o%j # Name of stdout output file
#SBATCH -e myjob.e%j # Name of stderr error file
#SBATCH -p shared # Queue (partition) name
#SBATCH -N 1 # Total # of nodes (must be 1 for serial)
#SBATCH -n 128 # Total # of cores to use
#SBATCH -t 00:05:00 # Run time (hh:mm:ss)
#SBATCH -A SEE230009 # Name of class allocation
#SBATCH --mail-user=vpaineni@umassd.edu
#SBATCH --mail-type=all # Send email at begin and end of job

echo "Hello"

# The "&" used below allows a program to run in the background.
# For example, all four runs of the trap program will be running
# at the same time. This is a simple form of parallel computing.
# at the same time. This is a simple form of parallel computing.

./parallel 1000 4 100000 1 >> StrongScaleTest1.dat
./parallel 1000 4 100000 2 >> StrongScaleTest1.dat
./parallel 1000 4 100000 4 >> StrongScaleTest1.dat
./parallel 1000 4 100000 8 >> StrongScaleTest1.dat
./parallel 1000 4 100000 16 >> StrongScaleTest1.dat
./parallel 1000 4 100000 32 >> StrongScaleTest1.dat
./parallel 1000 4 100000 64 >> StrongScaleTest1.dat
./parallel 1000 4 100000 128 >> StrongScaleTest1.dat

echo "Goodbye"
```

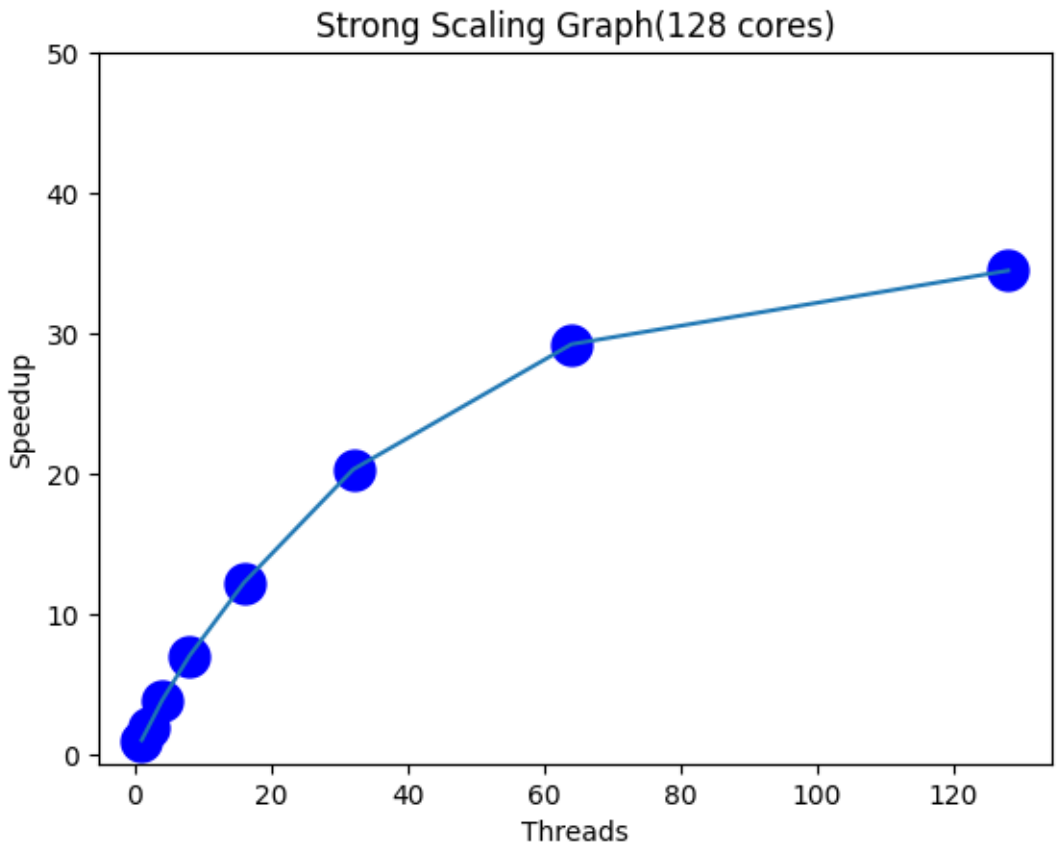
Dat File for Strong Scale Testing:



Plot for the Strong Scale Test:

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Loading data from file
strongScale128 = np.loadtxt("StrongScaleTest128.dat")

# Extracting the reference time for one processor
T_p1 = strongScale128[0, 1]

# Initializing an array for speedup values
speedup = np.zeros(8)

# Calculating speedup for each case
for i in range(8):
    speedup[i] = T_p1 / strongScale128[i, 1]

# Extracting the number of threads
threads = strongScale128[:, 0]

# Plotting the results
# Plotting the results
plt.plot(threads, speedup, 'bo', markersize=15)
plt.plot(threads, speedup)

# Formatting the plot
plt.yticks([50, 40, 30, 20, 10, 0])
plt.xlabel('Threads')
plt.ylabel('Speedup')
plt.title('Strong Scaling Graph(128 cores)')

# Saving the plot as a PDF file
#plt.savefig("strongscaling.pdf", bbox_inches="tight")
```

[ ]:
```
Text(0.5, 1.0, 'Strong Scaling Graph(128 cores)')
```

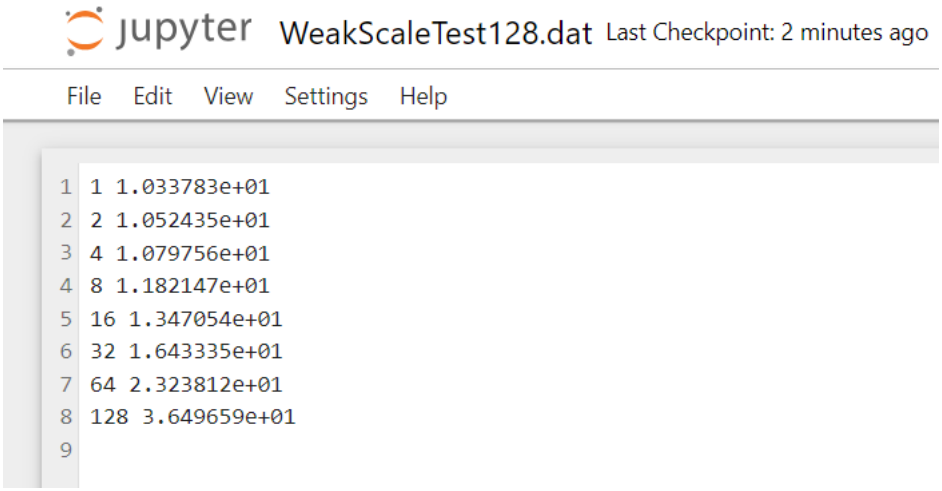


6.0.3 Anvil Job Script for Weak Scale Testing(128 cores) (anvilweak128.job)

```
#!/bin/bash
#----------------------------------------------------
#SBATCH -J myjob # Job name
#SBATCH -o myjob.o%j # Name of stdout output file
#SBATCH -e myjob.e%j # Name of stderr error file
#SBATCH -p shared # Queue (partition) name
#SBATCH -N 1 # Total # of nodes (must be 1 for serial)
#SBATCH -n 128 # Total # of cores to use
#SBATCH -t 00:10:00 # Run time (hh:mm:ss)
#SBATCH -A SEE230009 # Name of class allocation
#SBATCH --mail-user=vpaineni@umassd.edu
#SBATCH --mail-type=all # Send email at begin and end of job

echo "Hello"

# The "&" used below allows a program to run in the background.
# For example, all four runs of the trap program will be running
# at the same time. This is a simple form of parallel computing.
```

```
# at the same time. This is a simple form of parallel computing.

./parallel 1000 4 100000 1 >> WeakScaleTest1.dat
./parallel 1000 4 200000 2 >> WeakScaleTest1.dat
./parallel 1000 4 400000 4 >> WeakScaleTest1.dat
./parallel 1000 4 800000 8 >> WeakScaleTest1.dat
./parallel 1000 4 1600000 16 >> WeakScaleTest1.dat
./parallel 1000 4 3200000 32 >> WeakScaleTest1.dat
./parallel 1000 4 6400000 64 >> WeakScaleTest1.dat
./parallel 1000 4 12800000 128 >> WeakScaleTest1.dat

echo "Goodbye"
```

Dat File for Weak Scale Testing:

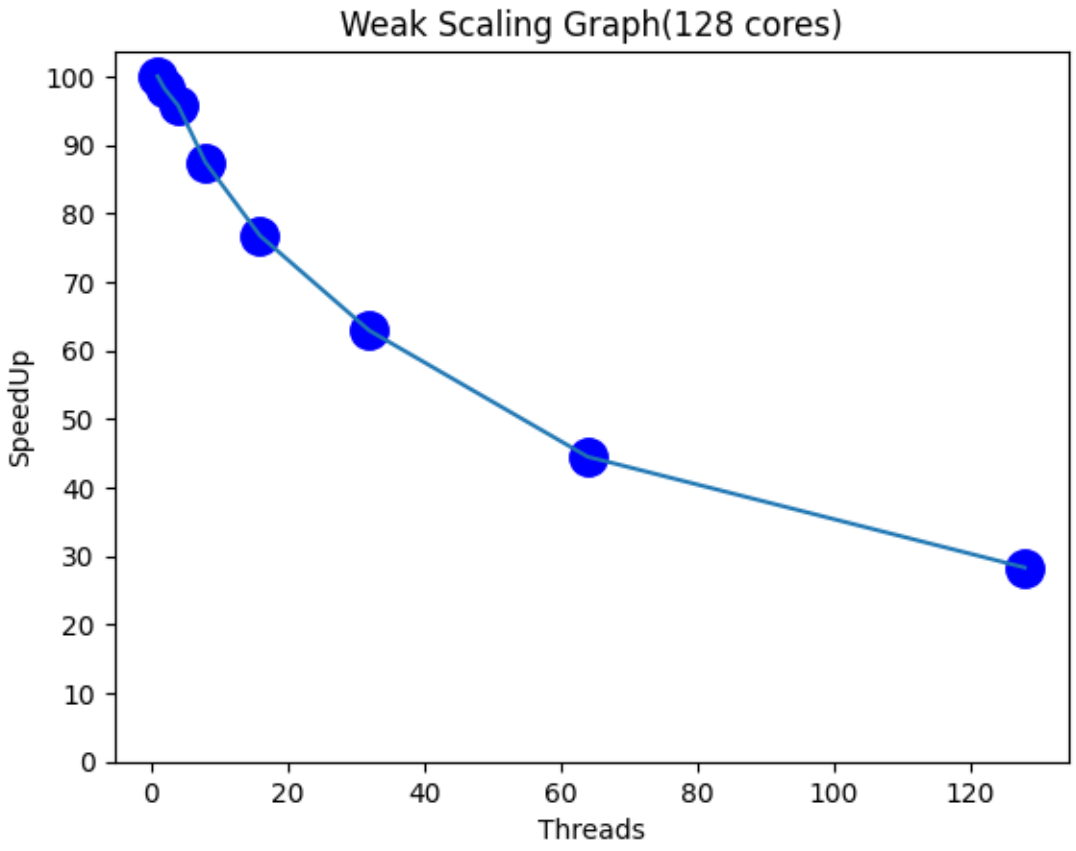


Plot for the Weak Scale Test:

[ ]:
```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt

# Loading data from file
weakScale128 = np.loadtxt("WeakScaleTest128.dat")

# Extracting the reference time for one processor
T_p1 = weakScale128[0, 1]

# Initializing an array for speedup values
speedup = np.zeros(8)

# Calculating speedup for each case
for i in range(8):
    speedup[i] = T_p1 / weakScale128[i, 1] * 100
```

```
# Extracting the number of threads
threads = weakScale128[:, 0]

# Plotting the results
plt.plot(threads, speedup, 'bo', markersize=14)
plt.plot(threads, speedup)

# Adding labels, legend, and formatting
plt.xlabel('Threads')
plt.ylabel('SpeedUp')
plt.yticks([100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0])
plt.title('Weak Scaling Graph(128 cores)')

# Saving the plot as a PDF file
#plt.savefig("weak.pdf", bbox_inches="tight")
```

[ ]: Text(0.5, 1.0, 'Weak Scaling Graph(128 cores)')



6.0.4 Anvil Job Script for Strong Scale Testing(512 cores) (anvilstrong512.job)

```
#!/bin/bash
#----------------------------------------------------
#SBATCH -J myjob # Job name
#SBATCH -o myjob.o%j # Name of stdout output file
#SBATCH -e myjob.e%j # Name of stderr error file
#SBATCH -p wholenode # Queue (partition) name
#SBATCH -N 4 # Total # of nodes (must be 1 for serial)
#SBATCH -n 512 # Total # of cores to use
#SBATCH -t 00:05:00 # Run time (hh:mm:ss)
#SBATCH -A SEE230009 # Name of class allocation
#SBATCH --mail-user=vpaineni@umassd.edu
#SBATCH --mail-type=all # Send email at begin and end of job

echo "Hello"

# The "&" used below allows a program to run in the background.
# For example, all four runs of the trap program will be running
# at the same time. This is a simple form of parallel computing.
# at the same time. This is a simple form of parallel computing.

./parallel 1000 4 100000 1 >> StrongScaleTest1.dat
./parallel 1000 4 100000 2 >> StrongScaleTest1.dat
./parallel 1000 4 100000 4 >> StrongScaleTest1.dat
./parallel 1000 4 100000 8 >> StrongScaleTest1.dat
./parallel 1000 4 100000 16 >> StrongScaleTest1.dat
./parallel 1000 4 100000 32 >> StrongScaleTest1.dat
./parallel 1000 4 100000 64 >> StrongScaleTest1.dat
./parallel 1000 4 100000 128 >> StrongScaleTest1.dat
./parallel 1000 4 100000 256 >> StrongScaleTest1.dat
./parallel 1000 4 100000 512 >> StrongScaleTest1.dat

echo "Goodbye"
```

Dat file for Strong Scale Testing:

jupyter StrongScaleTest512.dat Last Checkpoint: 28 seconds ago

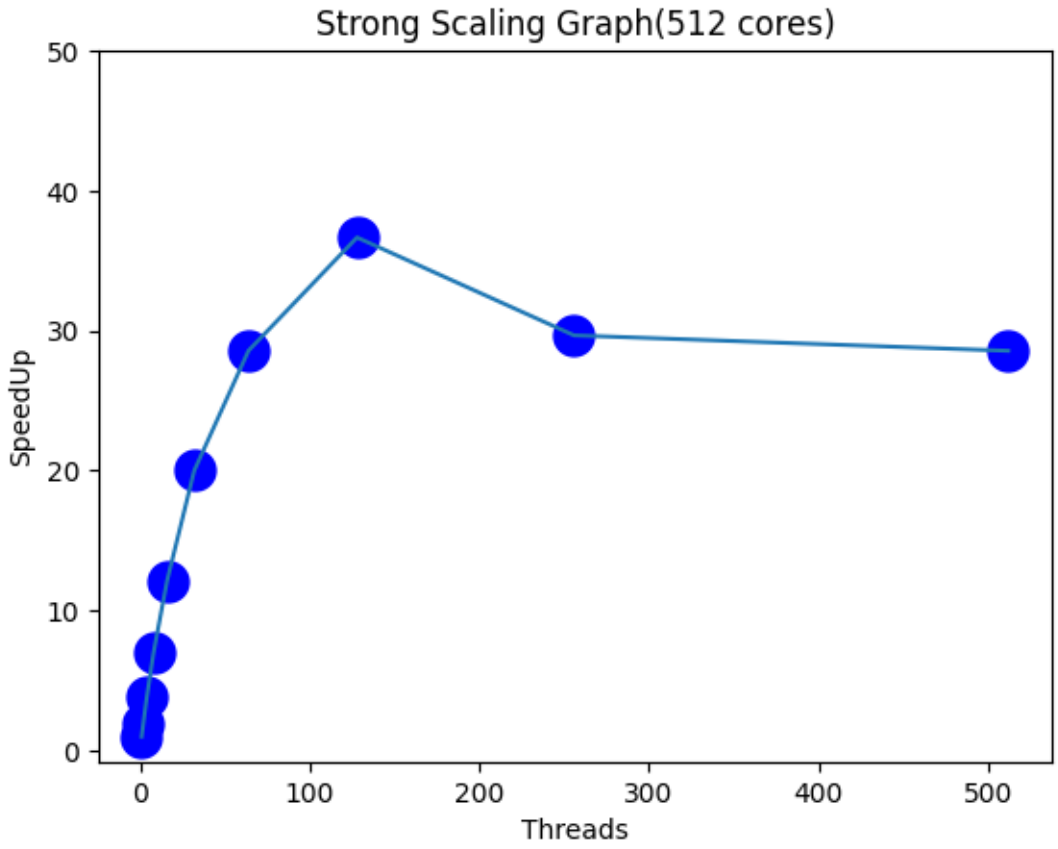
File Edit View Settings Help

```
1 1.033633e+01
2 5.256316e+00
3 4.716893e+00
4 8.478107e+00
5 16 8.514352e-01
6 32 5.166481e-01
7 64 3.617432e-01
8 128 2.819300e-01
9 256 3.484150e-01
10 512 3.618785e-01
11
```

Plot for strong scale testing with 512 cores:

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Loading data from file
strongScale512 = np.loadtxt("StrongScaleTest512.dat")

# Extracting the reference time for one processor
T_p1 = strongScale512[0, 1]

# Initializing an array for speedup values
speedup = np.zeros(strongScale512.shape[0])

# Calculating speedup for each case
for i in range(strongScale512.shape[0]):
    speedup[i] = T_p1 / strongScale512[i, 1]

# Extracting the number of threads
threads = strongScale512[:, 0]

# Plotting the results
plt.plot(threads, speedup, 'bo', markersize=15)
plt.plot(threads, speedup)

# Formatting the plot
plt.yticks([50, 40, 30, 20, 10, 0])
plt.xlabel('Threads')
plt.ylabel('SpeedUp')
plt.title('Strong Scaling Graph(512 cores)')
```

```
# Saving the plot as a PDF file
#plt.savefig("strongscaling.pdf", bbox_inches="tight")

# Show the plot
plt.show()
```

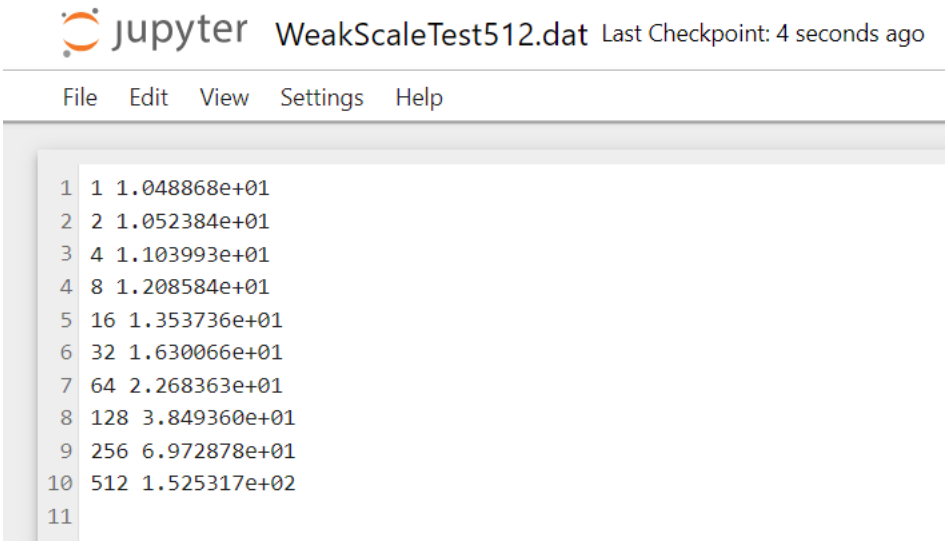


6.0.5 Anvil Job Script for Weak Scale Testing(512cores) (anvilweak512.job)

```
#!/bin/bash
#----------------------------------------------------
#SBATCH -J myjob # Job name
#SBATCH -o myjob.o%j # Name of stdout output file
#SBATCH -e myjob.e%j # Name of stderr error file
#SBATCH -p wholenode # Queue (partition) name
#SBATCH -N 4 # Total # of nodes (must be 1 for serial)
#SBATCH -n 512 # Total # of cores to use
#SBATCH -t 00:10:00 # Run time (hh:mm:ss)
#SBATCH -A SEE230009 # Name of class allocation
#SBATCH --mail-user=vpaineni@umassd.edu
#SBATCH --mail-type=all # Send email at begin and end of job
```

```
echo "Hello"

# The "&" used below allows a program to run in the background.
# For example, all four runs of the trap program will be running
# at the same time. This is a simple form of parallel computing.
# at the same time. This is a simple form of parallel computing.

./parallel 1000 4 100000 1 >> WeakScaleTest1.dat
./parallel 1000 4 200000 2 >> WeakScaleTest1.dat
./parallel 1000 4 400000 4 >> WeakScaleTest1.dat
./parallel 1000 4 800000 8 >> WeakScaleTest1.dat
./parallel 1000 4 1600000 16 >> WeakScaleTest1.dat
./parallel 1000 4 3200000 32 >> WeakScaleTest1.dat
./parallel 1000 4 6400000 64 >> WeakScaleTest1.dat
./parallel 1000 4 12800000 128 >> WeakScaleTest1.dat
./parallel 1000 4 25600000 256 >> WeakScaleTest1.dat
./parallel 1000 4 51200000 512 >> WeakScaleTest1.dat

echo "Goodbye"
```

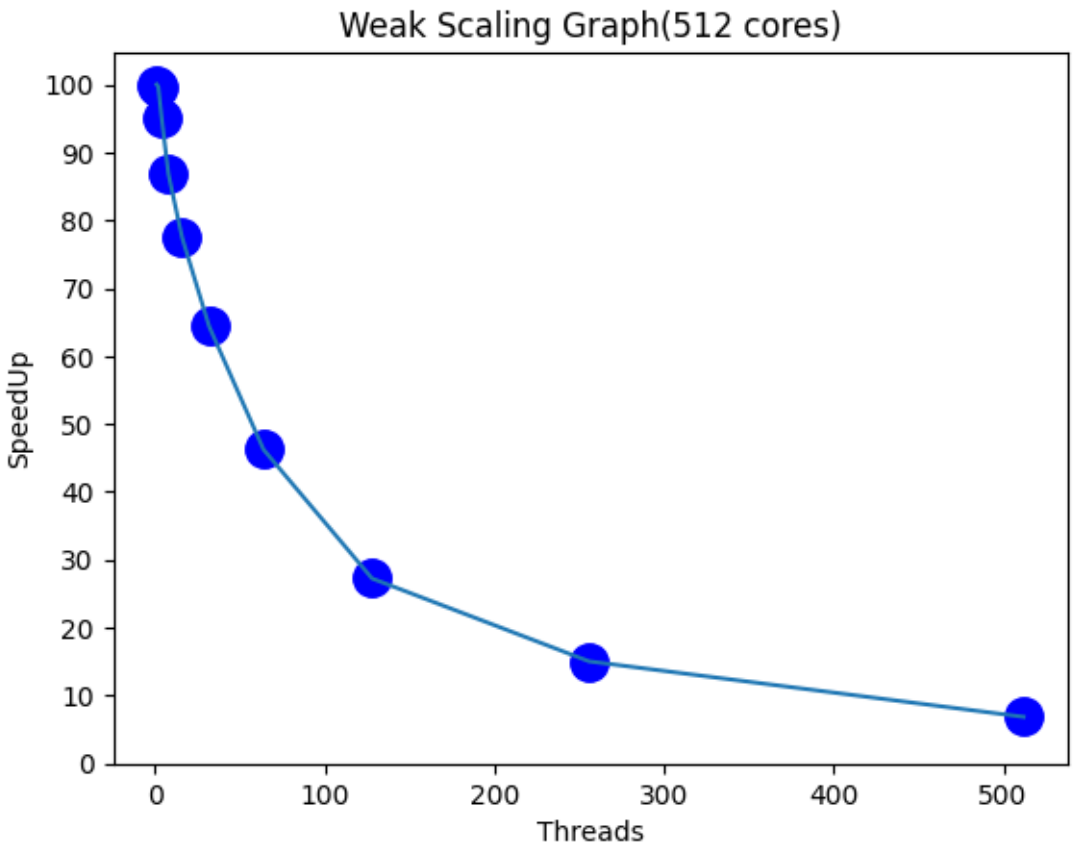
Dat file for Weak Scale Testing:



Plot for weak scale testing with 512 cores:

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Loading data from file
weakScale512 = np.loadtxt("WeakScaleTest512.dat")

# Extracting the reference time for one processor
```

```
T_p1 = weakScale512[0, 1]

# Initializing an array for speedup values
speedup = np.zeros(weakScale512.shape[0])

# Calculating speedup for each case
for i in range(weakScale512.shape[0]):
    speedup[i] = T_p1 / weakScale512[i, 1] * 100

# Extracting the number of threads
threads = weakScale512[:, 0]

# Plotting the results
plt.plot(threads, speedup, 'bo', markersize=14)
plt.plot(threads, speedup)

# Adding labels, legend, and formatting
plt.xlabel('Threads')
plt.ylabel('SpeedUp')
plt.yticks([100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0])
plt.title('Weak Scaling Graph(512 cores)')

# Saving the plot as a PDF file
#plt.savefig("weak.pdf", bbox_inches="tight")

# Show the plot
plt.show()
```



When we ran our code using more than 128 cores(upto 512 cores) we did not get any practical speed up. This may due to the fact that we were using omp for parallelization instead of mpi, because we needed to use more than 1 compute node to run the code with over 128 cores.

7 Conclusion

In this project we were able to create a parallelized implementation of a linear regression model utlizing a standard gradient descent algorithm for optimization. More specifically, we utilized omp to parallelize a training loop where we performed gradient descent optimization based on a mean squared error cost function to train the linear regression model. To evaluate performance of the parellization we performed a strong scaling test and weak scaling test with up to 128 cores. We noticed noticable speedup gains up to about 64 cores, then the performance gains would begin to taper off. In the future, we could take the work from this project to improve performance for other machine learning applications that utilize general gradient descent optimization algorithms.

8 References

1. https://www.geeksforgeeks.org/how-to-implement-weighted-mean-square-error-in-python/#
2. https://www.javatpoint.com/linear-regression-in-machine-learning
3. https://www.ibm.com/topics/gradient-descent#:~:text=Gradient%20descent%20is%20an%20optimization,e