PARALLELIZING REGRESSION USING MPI

Rishabh Upadhyay
NA20B052
Email: na20b052@smail.iitm.ac.in

ABSTRACT

This project proposes a parallelized implementation of Regression (linear and logistic) using Message Passing Interface (MPI) for distributed computing. By leveraging data parallelism, we aim to significantly reduce training time and enable efficient big data analytics. By distributing the training data across multiple processors, each processor can compute weight updates for its assigned data chunk independently, significantly reducing training time. This approach is particularly beneficial for large datasets where sequential processing becomes impractical. Processors only need to exchange minimal information (weight updates) after each training episode on their assigned data, minimizing communication overhead. These factors combined enable significant speedups in the training phase. This project aims to get fully functional MPI-based parallelized linear regression implementation and evaluate performance comparing training times and scalability analysis.
.

INTRODUCTION

Regression in machine learning is a statistical method used for predicting continuous outcomes. It involves finding the relationship between independent variables and a dependent variable. The goal is to build a model that can accurately predict the value of the dependent variable based on the values of the independent variables. Regression techniques include linear regression, polynomial regression, and support vector regression, among others. The core idea of regression is to find a mathematical function that best describes the relationship between the independent variables and the dependent variable. This function is typically represented as a line (in the case of simple linear regression), a plane (in the case of multiple linear regression), or a more complex curve (in the case of polynomial regression or other nonlinear regression techniques).

A linear regression model expresses the dependent variable Y as a linear combination of parameters multiplied by basis functions. It can be represented mathematically as:

$$\boldsymbol{Y} = \sum_{i=1}^{m} \boldsymbol{\theta} i \cdot \boldsymbol{fi}(\boldsymbol{X}) \quad + \boldsymbol{J}$$

- Y is the dependent variable (target).
- θi are the parameters (coefficients) to be estimated.
- $fi(X)$ are the basis functions of the independent variable X. These basis functions could be polynomials, trigonometric functions, or any other mathematical functions.
- J represents the error / cost term.

We aim to minimize the error term and get the best fitting model for a given dataset.
For a polynomial basis function, the number of parameters depends on the number of independent variables in X and the degree of polynomial chosen.

$$m = \frac{(n+d)!}{n!\,d!}$$

- m is number of parameters.
- n is the number of features in vector X.
- d is the degree of polynomial chosen.

For example, if X has two features and we fit a polynomial of degree two, there will be one constant term ($f0(X)=1$), two linear terms ($f1(X)=x1$ and $f2(X)=x2$), two quadratic terms ($f3(X)=x1^2$ and $f4(X)=x2^2$), and one cross-term ($f5(X)=x1 \cdot x2$).

GRADIENT DESCENT

Gradient descent is an optimization algorithm used to minimize the cost function in machine learning models. Its primary objective is to find the set of parameters that minimize the error between the predicted values and the actual values in the training data.

Gradient descent starts by initializing the parameters with some random values or zeros. At each iteration, the gradient of the cost function with respect to each parameter is computed. The gradient indicates the direction of steepest ascent, i.e., the direction in which the cost function increases the most. The parameters are updated in the opposite direction of the gradient to minimize the cost function. This update is performed using the following formula:

$$\boldsymbol{\theta i = \theta i - \alpha \cdot \left(\partial J(\theta) / \partial \theta i \right)}$$

- θi is the i-th parameter (or coefficient).
- α is the learning rate, which determines the size of the steps taken during optimization.
- $J(\theta)$ is the cost function.
- $\partial J(\theta) / \partial \theta i$ is the partial derivative of the cost function with respect to the i-th parameter.

By iteratively updating the parameters in the direction of the negative gradient, gradient descent aims to find the minimum of the cost function, which corresponds to the optimal set of parameters for the model. However, the choice of learning rate (α) is crucial, as too small a value may lead to slow convergence, while too large a value may cause the algorithm to overshoot the minimum or even diverge.

DATASET

The dataset used for this project is Auto MPG (Miles per Gallon). The Auto MPG dataset is a perfect choice for a linear regression task. It's a very popular dataset used for introductory machine learning and regression analysis. Three different instances of this dataset have been used, having 400, 1960 and 9800 datapoints respectively. This will help to capture the performance as data size increases.

The Auto MPG dataset consists of the following features:

- **Target variable:** Miles per Gallon (MPG)
- **Independent variables:**
  - Cylinders
  - Displacement (of the engine)
  - Horsepower
  - Weight
  - Acceleration
  - Model year
  - Origin

The goal is typically to use the various car features (independent variables) to predict the car's fuel efficiency (MPG) through linear regression.

To prepare the dataset to be used for regression with 2nd degree polynomial basis function, we calculate the derived features. The total number of parameters in $\boldsymbol{\theta}$ are given by m.

$$m = \frac{(n+d)!}{n!d!} = \frac{(7+2)!}{7!2!} = 36$$

SERIAL IMPLEMENTATION

The basis polynomial terms are generated. Gradient descent algorithm is implemented to train a linear regression model. The algorithm starts by initializing the model's parameters, which are essentially the weights assigned to each feature in the model. Then, it iterates for the specified number of times. In each iteration, the code calculates the model's predictions for the training data using the current parameters. It then compares these predictions to the actual target values to determine the errors.

Using the errors, we calculate a measure called the gradient. This gradient indicates the direction in which the parameters should be adjusted to minimize the overall error of the model. The learning rate controls the step size along this direction. Finally, the code updates the parameters by subtracting the product of the learning rate and the gradient from the current parameter values. After the iterations are complete, the code returns *rmse* (root mean squared error) to check for convergence.

PARALLEL APPROACH

The parallel linear regression algorithm using MPI facilitates distributed computing across multiple processes, enabling this code to train a linear regression model on large datasets efficiently. The implement can be conceptually divided into three sections:

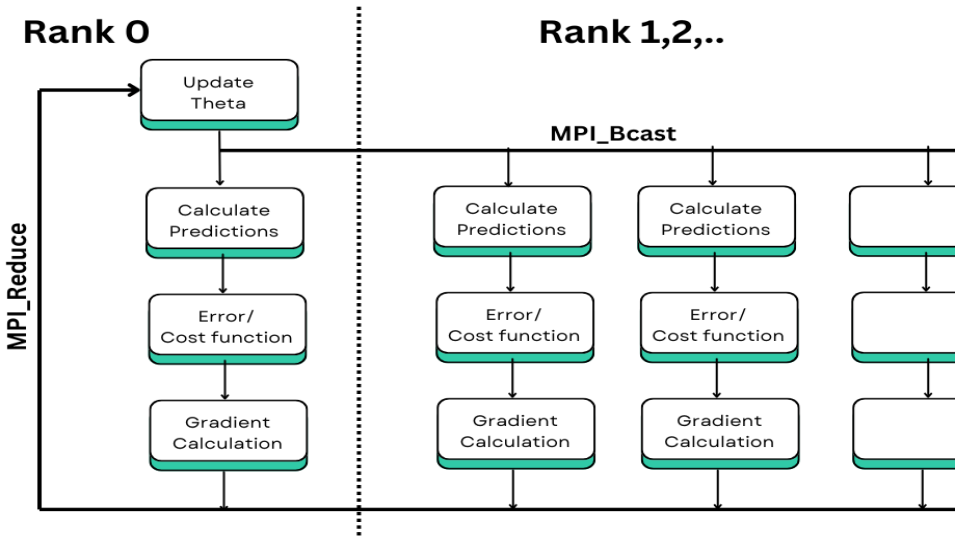
1. **Data Handling:**

Rank 0 (master process) reads the entire dataset. Rank 0 distributes data portions to other processes based on their rank using MPI. Each process receives its local data subset and constructs basis polynomial terms for its data points.

2. **Model Training:**

Each process trains a local linear regression model using gradient descent: Initializes model parameters (theta). In each iteration, calculates the gradient of the cost function with respect to the parameters. It determines the update direction for the parameters based on prediction errors.

MPI_Reduce is used to reduce (sum) the gradients calculated by all processes, resulting in the global gradient for the entire dataset.Master process then updates the model parameters using the global gradient and learning rate. The updated

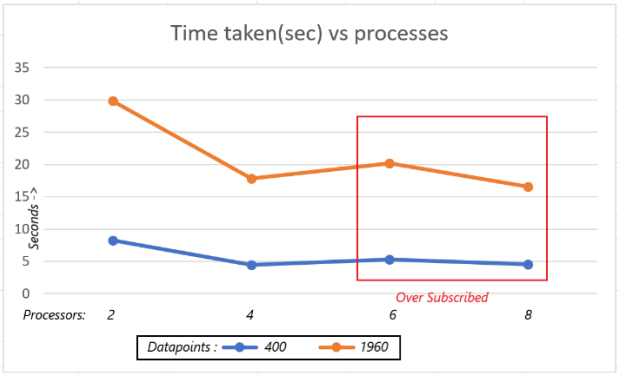


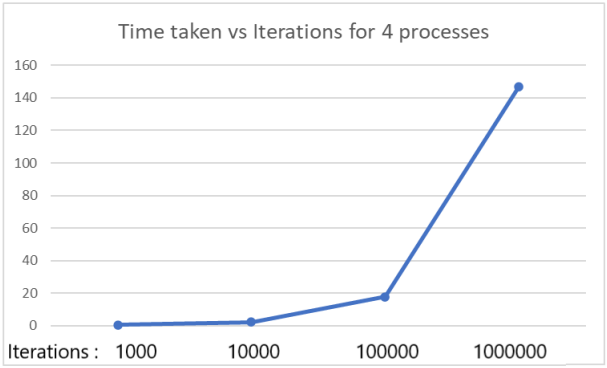
parameters are broadcasted to all processes for the next iteration, using MPI_Bcast. Figure above represents the flow of iterative loop.

3. Accuracy Calculation:

After training, each process calculates the *rmse* on its local data points. *rmse* measures the difference between model predictions and actual target values.

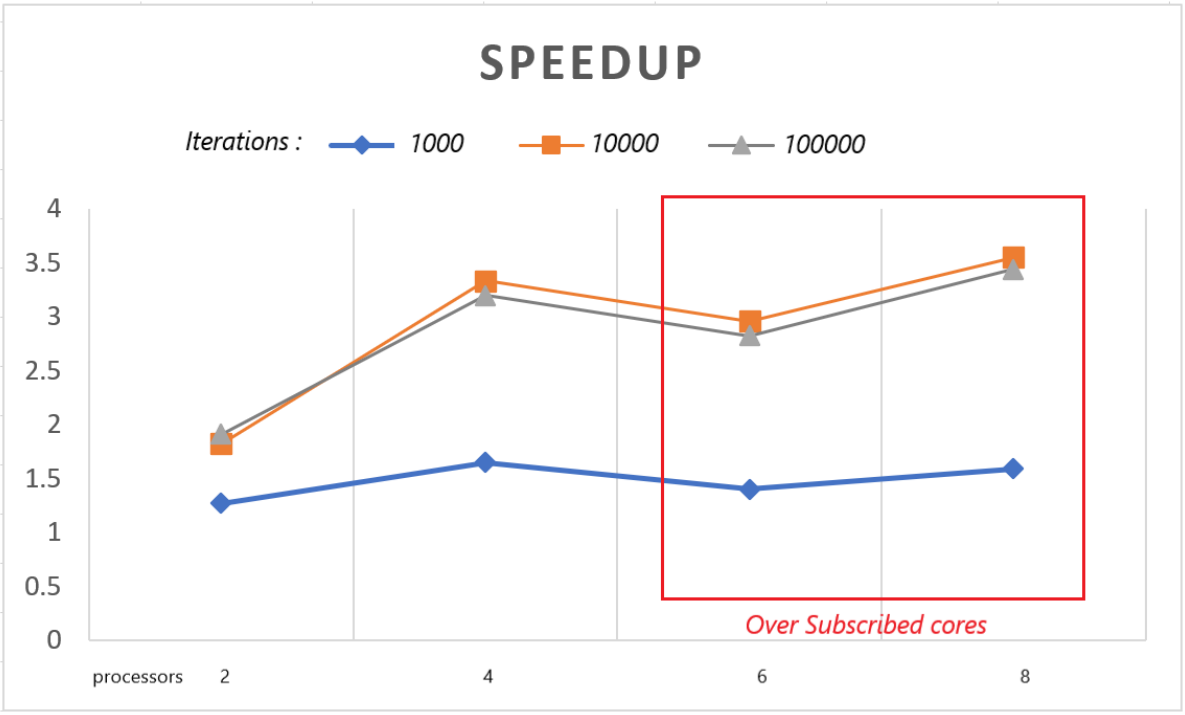
By distributing the workload of training the model across multiple processes, it allows for efficient handling of large datasets. Each process works on a subset of the data, and the gradients are combined to update the global model parameters.





RESULT AND CONCLUSION

The provided C++ code demonstrates a well-structured implementation of parallel linear regression using MPI. It effectively distributes data and computation, making it suitable for training linear regression models on large datasets.



To compile the code :

$ mpicc -o regexe MPI_Regression.cpp -lstdc++ -lm

If it does not work use

$ mpicc -o regexe MPI_Regression.cpp -lstdc++ -lm -lmpicxx

To run the code :

$ mpiexec -n 4 time ./regexe 0.01 100