PARALLEL COMPUTING IN STATISTICAL METHODS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ORÇUN OLTULU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
STATISTICS

AUGUST 2022

Approval of the thesis:

PARALLEL COMPUTING IN STATISTICAL METHODS

submitted by **ORÇUN OLTULU** in partial fulfillment of the requirements for the degree of **Master of Science in Statistics Department, Middle East Technical University** by,

Prof. Dr. Halil KALIPÇILAR
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Özlem İlk Dağ
Head of Department, **Statistics** _____

Assist. Prof. Dr. Fulya Gökalp Yavuz
Supervisor, **Statistics, METU** _____

Examining Committee Members:

Prof. Dr. Olçay Arslan
Statistics, Ankara University _____

Assist. Prof. Dr. Fulya Gökalp Yavuz
Statistics, METU _____

Prof. Dr. Ceylan Talu Yozgatlıgil
Statistics, METU _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Orçun Oltulu

Signature :

ABSTRACT

PARALLEL COMPUTING IN STATISTICAL METHODS

Oltulu, Orçun
M.S., Department of Statistics
Supervisor: Assist. Prof. Dr. Fulya Gökalp Yavuz

August 2022, 56 pages

Cost-efficient data collection and storage methods enable scientists, companies, and even regular computer users to reach high-dimensional data sets faster and cheaper. Even though personal computers are getting more powerful and efficient, some algorithms, tasks, and problems still require too much computational power and time to run on a personal computer. For a few decades, parallelization in statistical computing had an increasing trend, and researchers put significant effort into converting or adjusting known statistical methods and algorithms in parallel. The main reasons for the transition to parallel methods are the rapid growth in the size and the volume of data and the accelerated hardware developments. In this study, we applied the parallelization technique to statistical algorithms such as Linear Regression models, Non-parametric Regression models, and the measurement error kernel regression operator (MEKRO) algorithm for variable selection in Non-parametric Regression models. Simulation studies are conducted for each algorithm and recorded their accuracy measures and elapsed times to compare and see whether parallelization methods offer significant efficiency while maintaining the accuracy level as high as their sequential versions. The overall simulation results show that parallelization of the offers

a great potential of time efficiency with negligible or no changes in accuracy values.

Keywords: Parallel Computing, Multi-Core Systems, Linear Regression, Non-Parametric Regression, Variable Selection

ÖZ

İSTATİSTİKSEL YÖNTEMLERDE PARALEL HESAPLAMALAR

Oltulu, Orçun
Yüksek Lisans, İstatistik Bölümü
Tez Yöneticisi: Dr. Öğr. Üyesi. Fulya Gökalp Yavuz

Ağustos 2022 , 56 sayfa

Düşük maliyetli veri toplama ve depolama yöntemleri, bilim insanlarının, şirketlerin ve hatta günlük bilgisayar kullanıcılarının yüksek boyutlu veri setlerine daha hızlı ve daha ucuza ulaşmasını sağlamaktadır. Kişisel bilgisayarlar daha güçlü ve verimli hale gelmelerine rağmen bazı algoritmalar, görevler ve problemler kişisel bir bilgisayarda çalışmak için hala çok fazla hesaplama gücü ve zaman gerektirmektedir. Son yıllarda, istatistiksel hesaplamada paralel hesaplama yöntemlerinin cazibesi artmıştır ve araştırmacılar, bilinen istatistiksel yöntemleri ve algoritmaları paralel olarak çalıştırmak veya ayarlamak için önemli çaba sarf etmektedir. Paralel yöntemlere geçişin temel nedenleri, veri boyutundaki ve hacmindeki hızlı büyüme ve hızlanan donanım gelişmeleridir. Bu çalışmada, Doğrusal Regresyon modelleri, Parametrik Olmayan Regresyon modelleri ve Parametrik Olmayan Regresyon modellerinde değişken seçimi için kullanılan Ölçüm Hatası Çekirdek Regresyon Operatörü (MEKRO) algoritması gibi istatistiksel algoritmalar için parallel hesaplama tekniği uygulanmıştır. Her bir algoritma için simülasyon çalışmaları tasarlanmış, uygulanmış ve doğruluk ölçüleri ve geçen süreler kaydedilmiştir. Kullanılan paralelleştirme yöntemlerinin bu

algoritmalar için ardışık versiyonlar kadar yüksek doğruluk seviyesini korurken önemli verimlilik sağlayıp sağlamadığı tartışılmıştır. Genel simülasyon sonuçları, paralelleştirmenin, doğruluk değerlerinde ihmal edilebilir veya hiç değişiklik olmadan büyük bir zaman verimliliği potansiyeli sunduğunu göstermektedir.

Anahtar Kelimeler: Paralel Hesaplama, Çok Çekirdekli Sistemler, Doğrusal Regresyon, Parametrik Olmayan Regresyon, Değişken Seçimi

To my family and my love Ekin.

In memory of M.Hulki UZ.

ACKNOWLEDGMENTS

First of all, I would like to thank my thesis advisor Assist. Prof. Dr. Fulya Gökalp Yavuz for her limitless patience, quality support and motivation in this period. Without her effort these pages would not have become a thesis. There is not enough word to express my gratitude for her. I also would like thank my examining committee members, Prof. Dr. Olçay Arslan and Prof. Dr. Ceylan Talu Yozgatlıgil for their interest on my study and precious time to review my thesis.

I am thankful that I was a student and then worked as a research assistant at METU. I must present my gratitude to all the members of the Department of Statistics, METU; especially to my roommate Serenay Çakar and other fellow RA friends.

I wish to express my gratitude to my parents who have put limitless effort and never stopped supporting me. Also, I would like to thank my sweet sister Defne for making my life joyful and happier. I believe that without my family, I would not be the person I am today.

My love Ekin, I have no words to acknowledge your support, encouragement, and energy to overcome every obstacle and achieve all my goals. I feel super lucky to have you in my life.

I owe special thanks to my closest friend İbrahim Hakkı Erduran for being always there for me every time I needed. Also, special thanks to Muhammed Ali Aşgit, Berkay Türkan and Berker Fidancı for their wonderful friendship.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

2D	2 Dimensional
3D	3 Dimensional
AIC	Akaike Information Criterion
CDF	Cumulatives Distribution Function
CPU	Central Processing Unit
DnR	Division and Recombination
EM	Expectation-Maximization
EDF	Empirical Distribution Function
EP	Embarrassingly Parallel
GPU	Graphics Processing Unit
KDE	Kernel Density Estimation
LMS	Linear Multi-splitting
MCMC	Markov-Chain Monte Carlo
MEKRO	Measurement Error Kernel Regression Operator
MEM	Measurement Error Models
MLR	Multiple Linear Regression
MSE	Mean Squared Error
OLS	Ordinary Least Squares
OS	Operating System
RMSE	Root Mean Squared Error
SLLN	Strong Law of Large Numbers
SSE	Sum of Squared Errors
SWW	Stefanski, Wu, and White Method

CHAPTER 1

INTRODUCTION

The traditional way to write programs or software is called serial computing. In serial computing, the problem is broken into discrete series of instructions to be run on single CPU. Then, the instructions are executed one by one and only one instruction can be executed at any time point. Figure 1.1 shows an example of a schematic representation of serial (or sequential) computing.

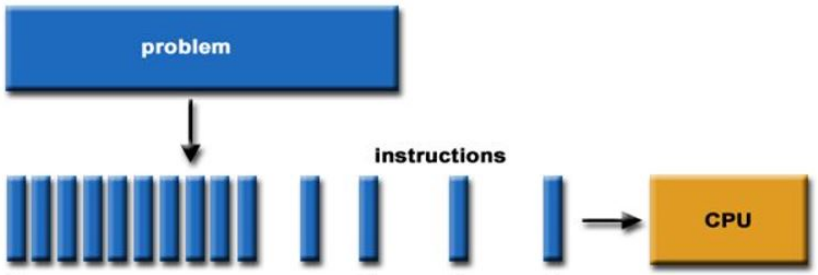

Figure 1.1: Serial Computing Scheme

On the other hand, parallel computing is a form of computation in which many calculations are carried out simultaneously. In its simplest form, it is the simultaneous use of multiple compute resources to solve a computational problem. An illustrative example for parallel computing scheme is provided in Figure 1.2.

Parallel computing can be briefly defined as solving a simple or a complex task using many processing elements. The idea is that the task is divided into many sub-tasks and they are solved at the same time. Therefore, parallel computing is a time and cost efficient process. After the invention of multi-core processors on daily-use computers, personal computers has become significantly more pow-

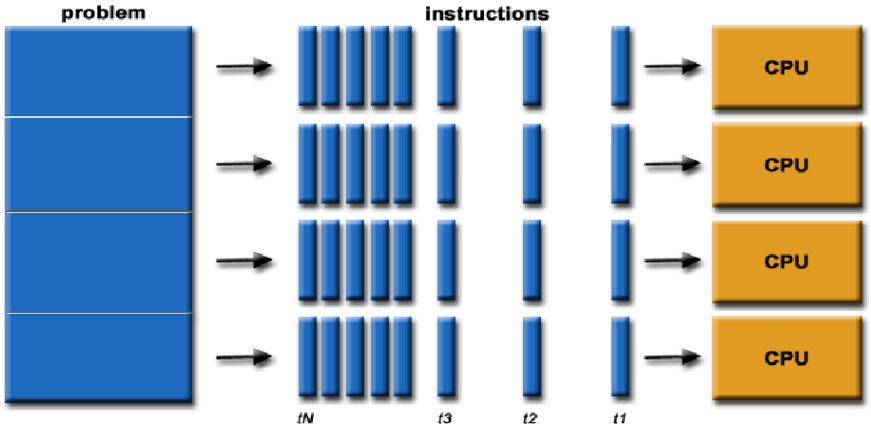

Figure 1.2: Parallel Computing Scheme

erful. However, serial computing, the basic way of coding, often avoids using the potential computation power of a machine. For example, while running a serial algorithm, only the first core of a personal computer's CPU is working, the rest of them are in 'sleep' mode.

There are two main parallelization methods for statistical computation. The first one is domain decomposition. The main idea behind domain decomposition, also known as data decomposition or data parallelism, is that the computational domain is distributed across computing processors. For each processor, different subsets of the domain, or data, are assigned and in each of them the same algorithm is applied. Using multiprocessors provides higher efficiency compared to single domain approaches. Moreover, domain decomposition can be easily applied in shared memory machines (1). The second method is functional decomposition. The main difference between functional decomposition, also known as task decomposition or task parallelism, and domain decomposition is that for functional decomposition, each processor has different functions. For this strategy, each processor is responsible for different portions of a cooperative task. This strategy can be run on different computers asynchronously (1).

Some problems, set of problem, or complete jobs can be easily parallelized, and this is even considered 'embarrassing' for a regular programmer; therefore, such algorithms are called Embarrassingly Parallel (EP) Algorithms. For an EP algorithm there is always a potential speed-up according to the number

of threads used since subtasks run completely independent from each other. Monte-Carlo simulations, Boostrap and cross-validation methods can be examples of EP Algorithms (1). Unfortunately, not all algorithms or problems can be considered as EP, there exist non-EP algorithms, especially in statistics, like expectation-maximization (EM) algorithm, and Markov-Chain Monte Carlo (MCMC) method. However, Matloff (2) suggests that some of the non-EP algorithms can be converted into EP, while maintaining their statistical properties.

Parallel statistical computing is one of the most popular techniques which was driven by the ability to store high dimension data sets and decrease in the cost of using the high performance computers. Parallelization of a statistical algorithm can even be done on modern personal computers.

In this study, we provide a simulation study to observe the potential of parallelization of least squares problem for multiple linear regression models using linear multi-splitting method described in (1; 3; 4; 5). In short, the parallel algorithm divides the global least squares problem into local LS problems and then recombines the local solutions with proper weighting and then estimate $\hat{\beta}$ with an iterative algorithm.

Divide and recombine (DnR) is one of the parallelization methods that allows the existing data or method to be implemented by dividing it into smaller pieces. It is possible to use the DnR method in most of the regression methods used to reveal the relationship between the data. Although several libraries created in existing programming languages for many of the regression methods, such an approach is not yet used for the non-parametric regression. However, it should be kept in mind that the non-parametric regression calculation method takes relatively long time. For this reason, parallelization would be a handy strategy to decrease the calculation time in non-parametric regression. In this study, we aim to demonstrate how time efficiency is achieved using DnR methods for non-parametric regression with the help of several parallelization strategies in R. The results indicate that the computation time is possible to be reduced proportionally with a trade off between time and accuracy.

We conduct a simulation study to compare parallel and sequential methods for

KDE and non-parametric regression models. The main purpose of this simulation study is to see if parallelization of those kernel methods allow time efficiency and provide accurate results as good as sequential (regular) versions and to provide open source code for prospective researchers. The comparison is based on run times (in seconds) and root mean squares of the models. For this aim, we use DnR method by running the process independently on each core. The process is started by dividing the data in random into several chunks and they are sent to each unique core to make the required calculations. Then, the results are collected from each core to present the final predictions. For the simulations, R programming language is used to fit non-parametric regression models with 'npreg' function from 'np' package (6).

Non-parametric approaches for variable selection in modeling maintains its popularity with increasing momentum for statistical and machine learning methods. The increasing size of the data and the number of variables make it necessary to develop the variable selection methods to work more effectively and fast. However, due to the nature of non-parametric methods, when the variable selection step is also added, the calculations get cumbersome. For example, due to its consecutive subtraction algorithm, kernel regression is a slow algorithm, even for small sample sizes. As the dimension of the training data increases, the cost of fitting the kernel regression model increases exponentially. The data has grown a lot, not only based on observations (rows) but also based on variables (columns). Therefore, reducing the data size with variable selection is an efficient way to reduce the training data dimension and workload. This study works on accelerating the variable selection algorithm in non-parametric (kernel) regression with parallelization. The algorithm combines two steps in non-parametric regression: bandwidth selection for the Nadaraya-Watson estimator and variable selection. Also, it consists of independent sequential calculations that iterate over each observation point. Due to its iterative calculations, this algorithm creates a high time cost as data sets' dimensions become more prominent. This computational load makes it impossible to use the methods proposed in this field for large data sets. To eliminate this deficiency, we apply a parallelization technique to the independent sequential analyses to reduce the processing time while keeping the

same accuracy level. We construct a simulation design to compare results for different dimensions of the artificial data and the different number of cores used in the parallelization. In the simulation, while the calculation results show a significant gain in the computation time of the parallelization methods used in R programming, it also gives the exact accuracy measurements.

CHAPTER 2

LITERATURE REVIEW

2.1 Parallel Linear Regression Models

Regression models have been used for modeling the relationship amongst the variables for so many years by statisticians. The basis of many machine learning and statistical methods, linear regression performs sufficiently accurate results in wide range of areas.

As we discuss in the earlier chapter, the accessibility of the vast amount of data is much easier than it was before. Large-scale structures bring along different problems such as timing for any calculations on the data. For example, LS algorithm requires high dimensional matrix multiplications and taking the inverse of high dimensional matrix.

To shorten the time required for matrix operations; in 1985, O'Leary and White (4) introduced a parallel multi-splitting strategy for linear systems. Building a linear regression model requires to solve OLS problem. In 1998, Renaut (3) introduced a solution to LS problem using parallel multi-splitting method. The idea behind this technique is that the minimization of the global $||Ax - b||^2$ is partitioned to local $||Ax - b||_r^2$. Then, the local 'r' OLS problems can be solved separately at the same time in a distributed scheme. After all the required 'r' problems solved, the outputs are combined using appropriate weights. The weighting is another object in this research area, there has been a lot of study to generalize this weighting concept in parallel multi-splitting method (7; 8). The algorithm that is discussed in (3), connects several articles and theorems, will be discussed in Chapter 3.

2.2 Parallel Kernel Density Estimation

Kernel method, which is the main tool of non-parametric methods, is proposed by Rosenblatt (9). Since then, it serves to wide range of subjects such as image processing (10), economics (11), geostatistics (12) and bioinformatics (13). Racine (14), pioneer parallel kernel density estimation (KDE) method, claims that parallelization can be done by splitting the data points into r chunks, and then on each core, separate KDEs are estimated for separate data chunks, then all the estimations from each core are combined. After the data splitting is done, approximately $\frac{n}{r}$ data points (where n denotes the number of data points and r denotes the number of CPU cores) are sent to each core. Therefore, the amount of time needed to estimate the kernel density is expected to be significantly shortened.

Several studies survey the parallelization of the kernel estimation (1; 14). Guo (1) states that the parallel implication of KDE can be done by decomposing the kernel function, K(.). Briefly, each part of the splitted data is sent to each processor to obtain density function estimations. Then, the estimated densities are combined in the master node. With this splitting and combining method, he claims that parallel method is faster than the sequential one and even linear speed-up can be achieved in some cases.

Instead of functional decomposition, Racine (14) suggests that KDE can be run parallel by data division. In this method, the sample is divided into chunks, and each distinct chunk is sent to distinct cores (processors). In each core, KDE method is applied on subsamples and finally the estimated values are combined in master node. This study is limited to KDE only, it is not extended to any regression model.

Michailidis and Margaritis (15) investigate the parallelization of the KDE technique in multi-core architecture. In the article, simulation study is conducted for univariate and multivariate KDE and it is concluded that near linear speed-up can be achieved for both cases by data partitioning method from the simulation studies. The article also provides a quantitative comparison of six parallel

programming frameworks covering Pthread, OpenMP and others. Additionally, Lopez-Novao (16) conduct simulation study for KDE on multi-core architecture, too. They conclude that even with a modest CPUs they are able to achieve a significant decrease in the elapsed run time.

2.3 Parallel Non-Parametric Regression Models

Modern applied data analysts take the advantage of non-parametric methods which do not have any distributional assumptions. However, non-parametric approaches are more demanding in terms of computational time than parametric ones, for several reasons such as requiring iterative steps. However, the pace of the enhancements of the hardware and software technologies allow us to shorten the time required to apply non-parametric methods. Also, we aware of the fact that the methods used to collect and store the data are being improved. Analyzing bigger data sets requires more time and computational effort. This requirement leads the reveal of parallel methods (3; 17) to shorten the time needed to solve a problem or fit a model.

Although there was a package called 'npRmpi' (18) which had created to solve kernel regression problems in distributed architectures by combining 'np' package (6) and Rmpi package (19) in R. Currently it is no longer available either directly from CRAN or from github. This package was constructed on an Open MPI base. Ho et al. (20) states that this package led Linux and Windows users to take several steps before using the package due to its Open MPI requirement; however, it was not a big issue for Mac OS users since Mac OS provides fully functioning Open MPI to the users.

2.4 Variable Selection in Non-Parametric Regression Models

In the literature, there are several well-known variable selection algorithms that can be applied to the kernel regression model. Those algorithms can be gathered under two main classes. The first class is the algorithms that down weight the

variables with small or no impact on the response, and the second class is the algorithms that implement subset selection. The effectiveness and usability of variable selection algorithms in those two classes have been discussed in several articles; (21; 22; 23; 24).

White et al. (25) introduced a variable selection method in kernel regression on the basis of their proposed generalized variable selection method, SWW, for measurement error models (MEM). In many cases, explanatory variables can be observed with some error; this leads to a violation of the fundamental assumption of statistical modeling. In such cases, the observed data is contaminated by measurement error. On MEM solution framework, SWW technique provides similar solution paths to LASSO for linear models (26; 27). However, applying the SWW technique on nonparametric regression provides the Nadaraya-Watson estimator (28). The article states that the measurement error kernel regression operator (MEKRO) combines two critical steps in kernel regression: bandwidth selection for Nadaraya-Watson estimator and variable selection.

In this study, we provide a time and cost-efficient algorithm based on the MEKRO method for variable selection in kernel regression for the first time in the literature. The algorithm is introduced in the third chapter. A simulation study, similar to the simulation in (26), is conducted and results are shown in the forth chapter.

CHAPTER 3

METHODOLOGY

3.1 Parallel Linear Regression Models

Under this section linear multi-splitting approach is discussed.

Define the OLS linear regression system where Y denotes the response variable and X denotes the covariate matrix (or design matrix). Then, we can write the linear regression model in matrix format as follows

$$.y = X\beta + \varepsilon \tag{3.1}$$

For the parameter estimation, the least square problem is solved by minimizing the SSE assuming that X^TX is non-singular. Then, the parameter estimation defined as follows

$$\hat{\beta} = (X^TX)^{-1}X^Ty. \tag{3.2}$$

The design matrix, denoted by X, can be too large and taking inverse of X^TX can create stress on the CPU in some cases. Thus, in 1998, Renaut (3) introduced a parallel multi-splitting strategy in which the matrix X^TX is decomposed by columns into disjoint blocks. Therefore, the burden on the CPU while taking the inverse of larger X^TX, is decreased significantly.

A multi-splitting of X^TX is a collection of matrices $M_i, N_i, E_i \in \mathbb{R}^{pxp}, i = 1, \ldots, r$ that satisfies:

- A = M - N

- $X^T X = M_i - N_i, i = 1, \ldots, r$
- $M_i \geq 0, \quad N_i \geq 0$
- E_i are non-negative diagonal matrices; $\sum_{i=1}^{r} E_i = I$

The LMS method is defined by the iteration:

$$\beta^{k+1} = E^{(i)} {M^{(i)}}^{-1} (N^{(i)} \beta^k + y), \quad i = 1, \ldots, r; k = 1, ... \tag{3.3}$$

With this algorithm, for every iteration, the workload is distributed to 'r' cores while solving r independent problems of the kind

$$M^{(i)} y_i^k = N^{(i)} x^k + y, \quad i = 1, ..., r \tag{3.4}$$

where y_j^k represents the solution of local least squares problem. LS problems now can be solved independently and then the only communication is required while updating β^{k+1} in Equation 3.3.

O'leary and White (4) suggests a decomposition technique in $A = M - N$;

$$A = \sum_{k=1}^{K} A_k,$$

where each matrix A_k has smaller rank than A itself. Moreover, by this method they introduced an example algorithm which depends on the fact that the whichever the initial β vector is chosen, after the iteration β vector converges to $\hat{\beta}_{OLS}$ by Berman Plemmons theorem (29; 5).

However, to ease of coding we propose an alternative solution. Instead of decomposing the matrix A with respect to a rule, we suggest to first create matrix N_i with randomly generated values from standard normal distribution (assuming that the data is standardized). By definition N_i has to be non-negative, then we take the absolute values of the randomly generated values. After N_i is being created, M_i matrix is created by $M_i = X^T X + N_i$.

Also, we need to initialize (β^0). According to Berman Plemmons theorem (29; 5) we can use any initial (β^0), therefore we start generating random values from

standard normal distribution. Then, $\hat{\beta}$ can be obtained when β^k converges to β^{k+1}.

For the further researches, initial (β^0) and decomposing matrix A can be optimized to achieve fewer iterations to save more time.

The Algorithm 1 introduce the parallel non-parametric regression process given below:

Algorithm 1 Parallel Linear Regression

1: $X \leftarrow Model\ Matrix, X_{(n,p)}$
2: $Y \leftarrow Dependent\ Variable, Y_{(n,1)}$
3: Standardize the data
4: $A \leftarrow X^T X$
5: $N^{(i)} \leftarrow rnorm(p)$
6: $M^{(i)} \leftarrow A + N^{(i)}$
7: $Define:\ D^{(i)}\ as\ \sum_{i=1}^{r} D^{(i)} = I_{(n,n)}$
8: Initialize $\beta_{p,1} \leftarrow rnorm(p)$
9:
10: $H = \sum_{i=1}^{r} D^{(i)} \% * \% solve(M^{(i)}) \% * \% N^{(i)}$
11: $G = \sum_{i=1}^{r} D^{(i)} \% * \% solve(M^{(i)})$
12:
13: $While\ (convergence\ condition)$,
14: $\quad beta.update < -H \% * \% beta + G \% * \% b$

Application of the Algorithm 1 can be found in Chapter 4; with simulation study and the results.

3.2 Parallel Kernel Density Estimation

Statistical inference often includes probability density functions and in some cases distributional properties like variance, skewness, kurtosis or family of the distribution are not provided or known with the data. If that is the situation, we may simply apply a non-parametric method called KDE on an observed data set, which is a sample represents the target distribution of the population.

For a given random variable, kernel, provides a probability density. The main idea to use Kernel in Density estimation is to extend the probability calculation to the range of the random variable with a Kernel function. A Kernel function is a real-valued, continuous and bounded function with the following property: (K: $\mathbb{R} \to \mathbb{R}$)

$$\int_x K(x)dx = 1. \tag{3.5}$$

By definition, sum of the probabilities in the domain of the random variable must be equal to one. Now, assume that we have a random variable X with cumulative distribution function, F(x), and the random sample $X_1, \ldots, X_n$ has the empirical distribution function, S(x), can be defined as

$$S(x) = \frac{\text{number of } X_i \leq x}{n} = \begin{cases} 0, & x < X_{(1)} \\ \frac{i}{n}, & X_{(i)} \leq x \leq X_{(i+1)}, \quad i = 1, 2, \ldots, n. \\ 1, & x \geq X_{(n)} \end{cases} \tag{3.6}$$

Then,

$$S(x) = \frac{\text{number of } X_i \leq x}{n} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[X_i \leq x]. \tag{3.7}$$

Moreover,

$$\hat{f}(x) = \lim_{h \to 0} \frac{S(x+h) - S(x-h)}{2h} = \frac{d}{dx}\hat{F}(x). \tag{3.8}$$

So, for small $h > 0$,

$$\begin{aligned} \hat{f}(x;h) &= \frac{1}{2nh}\sum_{i}^{n} \mathbb{1}[x-h \leq X_i \leq x+h] \\ &= \frac{1}{2nh}\sum_{i}^{n} \mathbb{1}\left[\frac{|X_i - x|}{h} \leq 1\right] \\ &= \frac{1}{nh}\sum_{i}^{n} K\left(\frac{X_i - x}{h}\right). \end{aligned} \tag{3.9}$$

Note that $\hat{f}(x)$ has the following properties:

- $\hat{f}(x)$ is a consistent estimator of f(x),

- $\hat{f}(x)$ is asymptotically unbiased for f(x).

There are several studies to show how parallelization of the kernel density can be estimated and the time efficiency that the parallelization provides to the users (14; 15; 30).

For the parallelized version of KDE; assume data set (X) is divided into 'r' chunks randomly, and from the above equation we define the estimation function in parallel as the following form:

$$\hat{f}^{(r)}(x;h) = \frac{1}{n^{(r)}h^{(r)}} \sum_{i=1}^{n^{(r)}} K\left(\frac{X_i^{(r)} - x^{(r)}}{h^{(r)}}\right), \tag{3.10}$$

where $\hat{f}^{(r)}(x)$ denotes the estimated kernel density function in r^{th} chunk and $X^{(r)}, n^{(r)}, h^{(r)}$ denote the r^{th} data chunk, the number of observation in r^{th} data chunk, and selected band width for the r^{th} data chunk, respectively.

After estimating r kernel density functions from r data chunks, they are combined in master node as

$$\hat{f}(x;h) = \bigcup_{r} \hat{f}^{(r)}(x). \tag{3.11}$$

Simulation study is conducted for Parallel Univariate KDE and the simulation results are provided in the Chapter 4.

3.3 Parallel Non-Parametric Regression Models

In statistics, linear models are very popular due to the fact that they are providing great accuracy, easy to apply and interpret. However, they depend on several assumptions, and in most of the cases it is hard to satisfy all the assumptions at the same time. Unlike linear models, kernel regression models do not require any distributional assumptions and that is why it is a good alternative for some of the modern data analysts.

In Kernel Regression, the main objective is to estimate the regression function g(x): $\mathbb{R}^p \to \mathbb{R}$ as follows:

$$\begin{aligned} g(x) &= \mathbb{E}[Y|X=x] \\ &= \int y f_{Y|X=x}(y)dy \\ &= \frac{1}{f_X(x)} \int y f_{x,y}(x,y)dy. \end{aligned} \tag{3.12}$$

For a fixed X, we have

$$\int f_{x,y}(x,y)dy = f_X(x). \tag{3.13}$$

Therefore;

$$\int y f_{x,y}(x,y)dy = \sum_i y_i K\left(\frac{x_i - x}{h}\right). \tag{3.14}$$

Finally the estimated regression function g(x) can be written as

$$\begin{aligned} \hat{g}(x) &= \mathbb{E}[Y|X=x] \\ &= \sum_{i=1}^{n} \frac{K(\frac{X_i - x}{h})}{\sum_{i=1}^{n} K(\frac{X_i - x}{h})} y_i \\ &= \sum_{i=1}^{n} W_i(x) y_i, \end{aligned} \tag{3.15}$$

where

$$W_i(x) = \frac{K(\frac{X_i - x}{h})}{\sum_{i=1}^{n} K(\frac{X_i - x}{h})}.$$

The resulting estimator is called Nadaraya–Watson estimator of the regression function g(x) (28).

It is possible to run KDE algorithm in parallel with data division for Kernel Regression. Several packages and functions available in CRAN is possible to use for this aim. *Foreach* and *parLapply* are two of them and they provide quite simple syntax with easy to read code blocks. For the first function, *foreach*, managing the algorithm to run as a loop, if possible, is the key point for this package. In this package there are two main operators, %do% and %dopar%. The first operator, %do%, is used to evaluate the algorithm sequentially; whereas

%dopar% operator is used to run the algorithm in parallel. With the second function, *parLapply*, syntax gets much easier. The usage of the function is quite similar to its sequential version, *lapply* which is one of the widely used function in apply family. The apply family is used to manipulate or apply different types of functions to partitions of data frames, matrices, lists etc. without creating a loop based scheme.

In the implementation of this study, after we divide the data set into several chunks, we use *foreach* and *parLapply* function to send our algorithm, including model estimation, testing and combining prediction values, to each core.

From the Nadaraya-Watson estimator for non-parametric regression we have the following equation:

$$\begin{aligned}\hat{g}(x) &= \mathbb{E}[Y|X=x] \\ &= \sum_{i=1}^{n} \frac{K(\frac{X_i - x}{h})}{\sum_{i=1}^{n} K(\frac{X_i - x}{h})} y_i. \end{aligned} \tag{3.16}$$

Now, after dividing training sample into 'r' chunks the equation will become as following:

$$\hat{g}^{(r)}(x) = \sum_{i=1}^{n^{(r)}} \frac{K(\frac{X_i^{(r)} - x^{(r)}}{h^{(r)}})}{\sum_{i=1}^{n^{(r)}} K(\frac{X_i^{(r)} - x^{(r)}}{h^{(r)}})} y_i^{(r)}, \tag{3.17}$$

where $\hat{g}^{(r)}(x)$ represents the fitted kernel regression model in the r^{th} chunk and $n^{(r)}$ represents the number of observations in the r^{th} chunk.

Then, to predict new values, each estimated models, $\hat{g}^{(r)}(x)$, are applied on new observations, X_{Test}.

$$\hat{g}^{(r)}(X_{Test}) = \hat{Y}_{Test}^{(r)} \tag{3.18}$$

Then, for final predictions,

$$\hat{Y}_{Test} = \frac{1}{r} \sum_{r} \hat{Y}_{Test}^{(r)} \tag{3.19}$$

Here, $\hat{Y}_{Test}$ can be a single value or a vector.

Figure 3.1 depicts the non-parametric regression in parallel. On this diagram, 'n' represents the number of observation in the training data set, 'r' represents the number of cores used, 'k' represents the number of observations sent to the cores as chunks, in most cases the data sets cannot be split with equal number of observations. However, small changes in the number of observations in the chunks are not likely to lead major differences. Moreover, 't' represents the number of observation in the 'new' or test data set. The same test set is sent to each chunk to predict response variable with 'r' different models. After 'r' models are predicted, each prediction for each individual, $\hat{y}_{rt}$ are combined and then mean value of each model for each individual are calculated as final prediction.

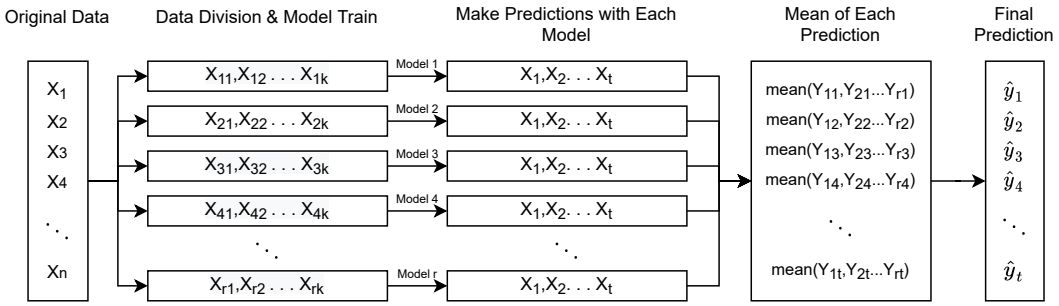

Figure 3.1: Illustration of Parallel Non-Parametric Regression.

The Algorithm 2 introduce the parallel non-parametric regression process given below:

Application of the Algorithm 2 can be found in Chapter 4; with simulation study and the results.

We discuss and adapt two main packages/functions into non-parametric regression models. The first function called *foreach* from 'foreach' package (31), counted among explicit parallelism methods in CRAN, is used. This package allows users to execute their operations and run their algorithms in parallel on a multi-core architecture. By the foreach package, user is provided to execute the algorithm in a loop based scheme, in other words the algorithm can be run repeatedly on a multi-core architecture. Weston and Steve (32) claims that it

Algorithm 2 Parallel Non-Parametric Regression

1: $x \leftarrow$ *Independent Variable(s)*
2: $y \leftarrow$ *Dependent Variable*
3: $h \leftarrow$ *Predefined bandwidth*
4: $K \leftarrow$ *Predefined Kernel Function*
5:
6: *Division*:
7: *Split x and y into r chunks*
8:
9: *Inside Function, f, that returns estimated response,* $\hat{y}$,
10: *Obtain* $K(x)$
11: *Obtain* W_i
12: *Estimate* $\hat{y} \leftarrow \sum_i W_i y_i$
13:
14: *In processor r, apply function f on r-th chunk,* $1 \leq r \leq R$,
15: *Recombination*:
16: $\hat{y} \leftarrow$ *Collections of outputs from each processor*

can be possible to complete a task in minutes where it measured in hours when it runs sequentially by using 'foreach' package and decent hardware. Additionally, unlike some of the packages in CRAN that are used for parallel computing, 'foreach' package is available for Windows OS users, too. In short, this package creates loop systems so that the divided data chunks can be sent to each slave node and on each node a non-parametric regression model can be fitted with a data chunk. The second function is called *parLapply* from 'parallel' package (33). This function is basically the parallel version of *lapply* function. In several cases, creating a loop system results in higher computational burden compared to an apply-like system. There are two main motivations to impose the aforementioned functions in this study. The first one is to show that both functions can be used by calling the required package in R without any pre-setup. Secondly, to see if apply family provides faster solutions compared to loop systems in parallel level. As far as we know, it is the first attempt in literature to give an insight into the several parallelization strategies in kernel regression.

3.4 Parallel Variable Selection Method in Non-Parametric Regression Models

In this part, the mathematical expression of the MEKRO method for variable selection in kernel regression is represented, and an example algorithm is provided. Then, parallelization of the algorithm is proposed.

3.4.1 MEKRO method for Variable Selection in Kernel Regression

Assume the observed data is $\{X_{ip}, Y_i\}_{i=1}^n$, where $X_{.p}$ denotes independent variables, and Y_i denotes dependent variable. White et al. (25) states that the application of the SWW to kernel regression results in measurement error kernel regression operator (MEKRO), where the MEM selection likelihood is

$$\hat{L}_{SEL}(\lambda) = -\frac{1}{n}\sum_{i=1}^{n}\{Y_i - \hat{g}(X_i, \lambda)\}^2, \tag{3.20}$$

where

$$\hat{g}(X_i, \lambda) = \frac{\sum_{k=1}^{n} Y_k \prod_{j=1}^{p} \exp\{-\lambda^2 (X_{i,j} - X_{k,j})^2/2\}}{\sum_{k=1}^{n} \prod_{j=1}^{p} \exp\{-\lambda^2 (X_{i,j} - X_{k,j})^2/2\}}, \tag{3.21}$$

where λ is smoothing parameter. Notice that the likelihood function defined in Equation 3.20 is the negative mean squared error. This is the objective function to be maximized. Additionally, for the observed data $\{X_{ip}, Y_i\}_{i=1}^n$, $\hat{g}(X_i, \lambda)$ defined in Equation 3.21 is the Nadaraya-Watson estimator of g(.), computed using Gaussian product kernel and diagonal bandwidth matrix.

Here, instead of using traditional smoothing bandwidths, h_j, inverse bandwidths $\lambda_j = 1/h_j$ are used. In this design, as inverse bandwidths get smaller and approach zero, bandwidths h_j get larger and larger and result in infinitely smoothed covariates, X_j, and thus they are selected out.

While maximizing MEM selection likelihood, in other words, negative mean squared error in equation Equation 3.20, a constraint is set as follows;

$$\lambda_j \geq 0, \quad j = 1, \ldots, p; \quad \sum_{j=1}^{p} \lambda_j = \tau, \quad \text{for fixed } \tau > 0, \tag{3.22}$$

where τ is a hyper-parameter that tunes the roughness of the estimated g(.) function. For smaller τ values, $||\lambda||$ to be small, implying bigger bandwidths and substantial smoothing. However, as the hyper-parameter gets larger, increase the roughness in estimated g(.), $\hat{g}(.)$, since it causes smaller individual bandwidths.

To avoid constrained optimization, White et al. (25) introduce $\gamma \in \mathbb{R}^p$ and let

$$\lambda_j(\gamma_j) = \frac{\tau\gamma_j^2}{\sum_{k=1}^{p}\gamma_j^2}, \quad j = 1, \ldots, p, \quad \text{for a fixed } \tau.$$

We then maximize $\hat{L}_{SEL}(\lambda(\gamma))$ with respect to γ. This guarantees that the constraints in Equation 3.22 on λ are satisfied for any γ, at the cost of one additional parameter.

Let $\pi_{ik} = \Pi_{j=i}^{p} \exp\{-\lambda^2(X_{i,j} - X_{k,j})^2/2\}$ and $\Gamma = \sum_{j=1}^{p}\gamma_j^2$, then

$$\hat{g}(X_i, \lambda) = \sum_{k=1}^{n} Y_k\pi_{ik} / \sum_{k=1}^{n}\pi_{ik}, \tag{3.23}$$

and the required gradients are

$$\frac{\partial\hat{g}(X_i,\lambda)}{\partial\lambda_t} = \left[\sum_{k=1}^{n} Y_k\pi_{ik}\{-\lambda_t(X_{i,t} - X_{k,t})^2\}\sum_{k=1}^{n}\pi_{ik} - \sum_{k=1}^{n}\pi_{ik}\{-\lambda_t(X_{i,t} - X_{k,t})^2\}\sum_{k=1}^{n} Y_k\pi_{ik}\right]\left(\sum_{k=1}^{n}\pi_{ik}\right)^{-2} \tag{3.24}$$

Also, $\partial\lambda_t/\partial\gamma_j = -2\tau\gamma_t^2\gamma_j\Gamma^{-2}$ when $t \neq j$ and $\partial\lambda_t/\partial\gamma_j = 2\tau\gamma_j(\Gamma - \gamma_t^2)\Gamma^{-2}$ when $t = j$. Finally,

$$\frac{\partial\hat{L}_{SEL}}{\partial\gamma_j} = -\frac{4\tau\gamma_j}{n\Gamma^2}\sum_{t=1}^{p}\left[\sum_{i=1}^{n}\{Y_i - \hat{g}(X_i,\lambda)\}\left(\frac{\partial\hat{g}(X_i,\lambda)}{\partial\lambda_t}\right)\right](\gamma_t^2 - \Gamma\mathbb{1}_{t=j}), \tag{3.25}$$

where $\mathbb{1}_{(.)}$ is the indicator function (25).

Then, the Akaike information criterion (AIC) for each value of the tuning parameter, τ, must be compared and finalized the selection algorithm with the one providing the minimum AIC value. To calculate AIC values that τ minimizes,

$$AIC_c(\tau) = \ln\{-\hat{L}_{SEL}(\hat{\lambda}_\tau)\} + \frac{n + tr(S_\tau)}{n - tr(S_\tau) - 2}, \tag{3.26}$$

where

$$S_\tau = \frac{\Pi_{j=i}^{p} \exp\{-\lambda_j^2 (X_{s,j} - X_{r,j})^2/2\}}{\sum_{k=1}^{n} \Pi_{j=i}^{p} \exp\{-\lambda_j^2 (X_{r,j} - X_{k,j})^2/2\}}. \tag{3.27}$$

3.4.2 MEKRO Algorithm

In this part, an explanation of the variable selection algorithm in kernel regression using MEKRO is given. A pseudo code for the MEKRO algorithm in (25) is provided in Algorithm 3.

The algorithm first initialize Y_i, X_i and hyper-parameter τ and set 4 main functions. The resulting output must optimize λ_j values. In the Algorithm 3, 'res' in 10^{th} and 19^{th} lines represent the i^{th} item in a vector of size 'n'. Therefore, the functions 'gx' and 'ggx' returns a vector of size 'n'. In the 9^{th} and 17^{th} lines of the Algorithm 3 while obtaining π_i, vectorization technique is suggested for the subtraction. By vectorization, we avoid an extra loop for every $\lambda_j, (j = 1, \ldots, p)$ which would slow down the algorithm. Also, while obtaining the gradient of g(.) function for every $\lambda_t; (t = 1, \ldots, p)$, instead of adding a loop scheme that goes over each t, it is suggested to call the 'ggx' function with 'sapply' function, in 'gL_Sel' function. This also makes a significant impact on the elapsed time for data sets consisting several independent variables; in other words, when 'p' is large. In the 29^{th} line of the Algorithm 3, 'gL_t' is a vector of size 'p'. Therefore, in the 30^{th} line colSums() function returns sum of $\sum_{i=1}^{n}(Y_i - \hat{g}(X_i, \lambda)) * \partial\hat{g}(X_i, \lambda)/\partial\lambda_t$ for $t = 1, \ldots, p$. Let's denote the sums of each column as $S_1, S_2, \ldots, S_p$
Now, assume that we have three explanatory variables, p=3; then we have the following for gg_x function;

$$\text{For } j = 1, \quad \frac{-4\tau}{n\Gamma^2}\gamma_1\Big[S_1(\gamma_1^2 - \Gamma) + S_2\gamma_2^2 + S_3\gamma_3^2\Big]$$

$$\text{For } j = 2, \quad \frac{-4\tau}{n\Gamma^2}\gamma_1\Big[S_1\gamma_1^2 + S_2(\gamma_2^2 - \Gamma) + S_3\gamma_3^2\Big]$$

$$\text{For } j = 3, \quad \frac{-4\tau}{n\Gamma^2}\gamma_1\Big[S_1\gamma_1^2 + S_2\gamma_2^2 + S_3(\gamma_3^2 - \Gamma)\Big]$$

Algorithm 3 Variable Selection in Kernel Regression

Input: X_i, Y_i, τ

Output: λ_j

Write a function, gx, to obtain $\hat{g}(X_i, \lambda)$:

gx:

  for i in 1:n

    for k in 1:n

      $\pi_i = prod(\exp(-\lambda_j^2(X_{i,j} - X_{k,j})^2\ )\ )$

    $res_i = \sum_{k=1}^{n} Y_k \pi_i / \sum_{k=1}^{n} \pi_i$

  return(res)

Write a function, ggx, to obtain $\frac{\hat{g}(X_i,\lambda)}{\partial \lambda_t}$ for every λ_t:

ggx:

  for i in 1:n

    for i in 1:k

      $\pi_i = prod(\exp(-\lambda_j^2(X_{i,j} - X_{k,j})^2\ )\ )$

      $L_t = -\lambda_t(X_{i,j} - X_{k,j})^2$

    $res_i = \Big[\sum_{k=1}^{n} Y_k \pi_i L_t * \sum_{k=1}^{n} \pi_i - \sum_{k=1}^{n} \pi_i * L_t * \sum_{k=1}^{n} Y_k \pi_i\Big] / (\sum_{i=1}^{k} \pi_i)^2$

  return(res)

Write a function, L_Sel, to obtain $\hat{L}_{SEL}(\lambda)$:

L_Sel:

  return$(-\frac{1}{n}\sum_{i=1}^{n}\{Y_i - \hat{g}(X_i, \lambda)\}^2)$

Write a function, gL_Sel, to obtain $\frac{\partial \hat{L}_{SEL}}{\partial \gamma_j}$:

gL_Sel:

  C $= (-4\tau)/(n\Gamma^2)$

  gL_t = Apply ggx() function for each λ_t, t=1,..,p

  res = C * γ_j * colSums$(\sum_{i=1}^{n}(Y_i - gx()) * gL_t)$ %*% $(\gamma_j^2 - diag(\Gamma, p))$

  return(res)

Maximize L_Sel function using it's gradient gL_Sel.

and this is converted into R language as vectorization and matrix multiplication as follows

$$\frac{-4\tau}{n\Gamma^2}\left[\gamma_1 S_1 \gamma_2 S_2 \gamma_3 S_3\right]\begin{bmatrix}\gamma_1^2-\Gamma & \gamma_2^2 & \gamma_3^2\\ \gamma_1^2 & \gamma_2^2-\Gamma & \gamma_3^2\\ \gamma_1^2 & \gamma_2^2 & \gamma_3^2-\Gamma\end{bmatrix}$$

Therefore, Equation 3.24 's indicator function can be easily implemented in the R language.

Notice that in Algorithm 3, there are two π_i calculations in the 9^{th} and 17^{th} lines. Those could be easily combined in a single function; however, we choose to write the algorithm with one more step for the sake of readability.

3.4.3 MEKRO with Categorical Variables

Based on the approach described in (34), the extended MEKRO for variable selection in kernel regression where categorical variable(s) are also included in the model is introduced in this chapter.

Let two domains C and D denotes the variable space for continuous and categorical respectively.

$$\hat{g}(X_i,\lambda)=\frac{\sum_{k=1}^{n} Y_k \prod_{j\in C} e^{\{-\lambda^2(X_{i,j}-X_{k,j})^2/2\}} \prod_{j\in D} e^{\{(-\lambda_j w_j \mathbb{1}_{X_{kj}\neq X_{ij}})/2\}}}{\sum_{k=1}^{n} \prod_{j\in C} e^{\{-\lambda^2(X_{i,j}-X_{k,j})^2/2\}} \prod_{j\in D} e^{\{(-\lambda_j w_j \mathbb{1}_{X_{kj}\neq X_{ij}})/2\}}} \quad (3.28)$$

where

$$w_j=\frac{2}{1-[\sum_{t=1}^{D_j}\{\hat{P}(X_{kj}=t)\}^2]}, \quad (3.29)$$

where

$$\hat{P}(X_{kj}=t)=\frac{1}{n}\sum_{k=1}^{n}\mathbb{1}_{X_{kj}=t}. \quad (3.30)$$

Algorithm 4 illustrates the MEKRO algorithm runs with categorical predictors.

Algorithm 4 MEKRO with Categorical Predictors

1:
2: *Write a function, gx, to obtain* $\hat{g}(X_i, \lambda)$:
3: gx:
4: Calculate w_j;
5: for i in 1:n
6: for k in 1:n
7: $\pi_{C_i} = prod(\exp(-\lambda_j^2(X_{i,j} - X_{k,j})^2\ )\ )$
8: $\pi_{D_i} = prod(\exp(-\lambda_j w_j/2))$
9: $res_i = \sum_{k=1}^{n} Y_k \pi_{C_i} \pi_{D_i} / \sum_{k=1}^{n} \pi_{C_i} \pi_{D_i}$
10: return(res)

3.4.4 Parallelization of Variable Selection Method in Kernel Regression

As it can be easily seen in the Algorithm 3, estimation of $\hat{g}(X_i, \lambda)$ is done by taking one observation, X_i, and make 'n' separate calculations for π_{ik}, where $\pi_{ik} = \Pi_{j=i}^{p} \exp\{-\lambda^2(X_{i,j} - X_{k,j})^2/2\}$, and repeats for every 'n' observations consecutively. For every X_i, the calculations are done sequentially and creates a burden on the processors, therefore takes much time as the dimension of the data is getting larger. This calculation also takes place in obtaining gradient of $\hat{g}(X_i, \lambda)$, which is $\frac{\partial \hat{g}(X_i, \lambda)}{\partial \lambda_t}$.

At this point since every $\pi_{ik};\ i = 1, \ldots, n$, the calculation is completely independent of each other; this process can be considered as an embarrassingly parallel process. Therefore, it can be run in parallel, with a potential speed up even for small sample sizes.

Elapsed time and AIC comparison will be reported and discussed in the following chapters. After parallelization of the process, the objective function to be maximized now becomes as follows

$$\hat{L}_{SEL}(\lambda) = -\frac{1}{n} \sum_{i=1}^{n} \{Y_i - \cup_{c=1}^{r} \hat{g}(X_i^c, \lambda)\}^2, \tag{3.31}$$

where 'c' denotes the c^{th} chunk whereas 'r' denotes the number of chunks the

parallelized process has; in other words, the number of cores is used in the parallel process.
Also, the gradient of the objective function then become as following;

$$\frac{\partial \hat{L}_{SEL}}{\partial \gamma_j} = -\frac{4\tau\gamma_j}{n\Gamma^2} \sum_{t=1}^{p} \left[\sum_{i=1}^{n} \{Y_i - \cup_{c=1}^{r} \hat{g}(X_i^c, \lambda)\} (\frac{\partial \hat{g}(X_i, \lambda)}{\partial \lambda_t}) \right] (\gamma_t^2 - \Gamma \mathbb{1}_{t=j}). \quad (3.32)$$

In order to run the algorithm in parallel, *foreach* package in R is used. The outer loop in the Algorithm 3 for '*gx*' and '*ggx*' are converted to a parallel loop scheme with *foreach* package. Additional to the parallelization of π_{ik} calculation, in gL_Sel function in Algorithm 2, we apply ggx() function for each λ_t, t=1,..,p. This can also be done in parallel using *parSapply* function, which allows us to run the classical 'sapply' function in parallel. This parallelization does not provide shorter elapsed time values for small 'p' values, but it may lead to a better elapsed time for larger dimensions.

Algorithm 5 illustrates parallel subset selection algorithm with only continuous variables. The MEKRO algorithm can run with categorical domains too. However, parallelization of the categorical domain, denoted with (D) in Section 3.4.3, does not provide any time efficiency. The reason is that the categorical part of the algorithm depends on the proportions of levels of categorical variables, thus does not require a high performance computing. At this point, if the data set contains both categorical and continuous variables, it is suggested that while estimating $\hat{g}(X_i, \lambda)$ in Section 3.4.3, the calculations are made for continuous variables run on parallel but for categorical variables they can run sequentially.

If the data set includes categorical variables too, then the 8^{th} line in Algorithm 4 can be added to Algorithm 5, and multiplied with the values come from the parallel part for continuous variables of the algorithm.

Algorithm 5 Parallel Variable Selection in Kernel Regression

Input: X_i, Y_i, τ

Output: λ_j

Write a function, gx_par, to obtain $\hat{g}(X_i, \lambda)$:

gx_par:

  foreach parallel i in 1:n

    foreach sequential k in 1:n

      $\pi_i = prod(\exp(-\lambda_j^2(X_{i,j} - X_{k,j})^2\ )\ )$

    $res_i = \sum_{k=1}^{n} Y_k \pi_i / \sum_{k=1}^{n} \pi_i$

  return(res)

Write a function, ggx_par, to obtain $\frac{\hat{g}(X_i,\lambda)}{\partial \lambda_t}$ for every λ_t:

ggx_par:

Update parameters; λ_j, γ_j

  foreach parallel i in 1:n

    foreach sequential k in 1:n

      $\pi_i = prod(\exp(-\lambda_j^2(X_{i,j} - X_{k,j})^2\ )\ )$

      $L_t = -\lambda_t(X_{i,j} - X_{k,j})^2$

      $\text{Nom} = \left[\sum_{k=1}^{n} Y_k \pi_i L_t * \sum_{k=1}^{n} \pi_i - \sum_{k=1}^{n} \pi_i * L_t * \sum_{k=1}^{n} Y_k \pi_i\right]$

      $\text{Denom} = (\sum_{i=1}^{k} \pi_i)^2$

    res_i = Nom / Denom

  return(res)

Write a function, L_Sel_par, to obtain $\hat{L}_{SEL}(\lambda)$:

L_Sel_par:

  return$(-\frac{1}{n}\sum_{i=1}^{n}\{Y_i - \hat{g}(X_i, \lambda)\}^2)$

Write a function, gL_Sel_par, to obtain $\frac{\partial \hat{L}_{SEL}}{\partial \gamma_j}$:

gL_Sel_par:

  C = $(-4\tau)/(n\Gamma^2)$

  gL_t = **parSapply** ggx_par() function for each λ_t, t=1,..,p

  S = colSums$(\sum_{i=1}^{n}(Y_i - gx_par()) * gL_t)$

  res = C * γ_j * S %*% $(\gamma_j^2 - diag(\Gamma, p))$

  return(res)

Maximize L_Sel function using it's gradient gL_Sel_par.

CHAPTER 4

SIMULATION STUDIES

In this chapter both simulation designs and corresponding simulation results are provided.

4.1 Simulation Design

Each subsection in this part defines corresponding simulation designs. For some of the simulation studies a personal computer is used; however, for those which require significantly higher computation time are run on an external server.

4.1.1 Simulation Design for Parallel Linear Regression Models

In this section the simulation design for application of parallel linear multi-splitting algorithm on linear regression models is introduced. In this design there are 3 different sample sizes; $10^4, 10^5, 10^6$ for training set and respectively 3 different sample sizes, $10^2, 10^3, 10^4$ for testing. Also, 4 different number of explanatory variables, $5, 10, 15, 20$ are used. The data sets are generated randomly and the process repeats 30 times for each sample sizes. Therefore in this simulation study we have created $3x3x30 = 2700$ distinct training and test sets. In this study, sequential and parallel runs with 2-core, 4-core and 8-core are compared in terms of elapsed time (in seconds) and MSE.

The simulation is run with R programming language and using *foreach* function. The hardware we use for this simulation is an Intel Core i7-10875H CPU having 8 logical cores running with 64-bit Windows OS.

4.1.2 Simulation Design for Parallel Kernel Density Estimation

The simulation study for KDE is designed to compare the median elapsed time (in seconds) values while estimating the kernel density for a random variable. Three different sample sizes, n=$10^5, 10^6, 10^7$ and record elapsed time for 30 different runs for each sample sizes are used in the simulation design. Sequential and parallel runs with 2-core, 4-core and 8-core are compared. Data generation for each simulation run is carried out randomly, and the samples are generated from normal distribution with mean zero and standard deviation one.

The simulation is run with R programming language and using *foreach* and *parLapply* functions. The hardware we use for this simulation is exactly the same as we use for the simulation study for parallel linear regression models.

4.1.3 Simulation Design for Parallel Non-Parametric Regression

The simulation design of this part is based on 3 different sample sizes for training set which are $10^2, 10^3$ and 10^4 and the test sets consist of 25% of each training set. This is the upper limit of the hardware used for this study, wider range of sample sizes and different number of cores may be investigated for further studies.

Briefly, data sets are split into r chunks randomly. After splitting the train set, each chunk are being sent to separate cores to fit a kernel regression model and then the test set is sent to separate cores as a whole to make predictions on different models. Finally, all the predictions from different models on different cores are collected and the mean value of all predictions are taken from different cores for every single test observation. The pseudo code of this procedure is located in Algorithm 2 for details.

Algorithm 6 Simulation: Parallel Non-Parametric Regression

1: *Data Generation (Train & Test Sets)*:
2: $f \leftarrow$ *Define a function*
3: $x \leftarrow$ *Generate Independent Variable(s)*
4: $\varepsilon_i \leftarrow$ *Generate an Error Vector*
5: $y \leftarrow$ *Generate Dependent Variable,* $f(x) + \varepsilon_i$
6:
7: *Division*:
8: *Split Train set into r chunks*
9: *Fit Kernel Regression Model on each chunk,* $Model_r$
10: *Make Prediction for Test set with fitted models on each chunk*
11: $\hat{y_r} \leftarrow Model_r(X_{test})$
12:
13: *Recombination*:
14: *Collect prediction (*$\hat{y_r}$*) vectors from each chunk*
15: $\hat{y_i} \leftarrow \frac{1}{r} \sum_r \hat{y}_{ri}$

For the simulations, R programming language is used to fit kernel regression models with 'npreg' function from 'np' package (6). All simulations for the kernel regression were run on a server called TRUBA - levrekv2, provides 8-core Intel E5-2680v3 processor, with overall 24-cores, supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK). The system has 256 GB memory and runs with Centos 7.3 Linux OS.

4.1.3.1 Univariate Case

The first simulation design is constructed for one dependent (Y) and one independent (X) variable. The univariate function used in the simulation is defined as below:

$$f(x) = x^2 cos(x).$$

Figure 4.1 shows the shape of the function.

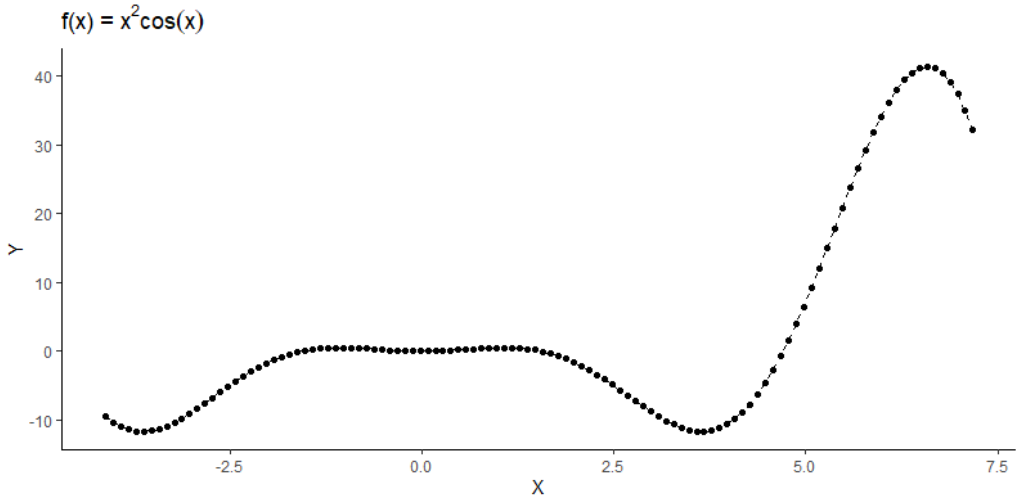

Figure 4.1: Shape of the univariate function.

Data sets are generated with the following steps:

- Generate X from a standard normal distribution.
- Generate ϵ from a standard normal distribution.
- Generate Y by applying a non-linear function (f) on X, then add ϵ, generated from standard normal distribution.

$$Y = f(X) + \epsilon.$$

When a random data set is generated, it is used in each scenario; in other words, a non-parametric model is fit with the same data but on different number of cores. Then, with the same design there is another random sample which is 25% of training data for the testing purpose. Each scenario is run 100 times and their run time (in seconds) and RMSE values are stored for further investigation.

4.1.3.2 Bivariate Case

The second simulation design is created for one dependent (Y) and 2 independent variables ($X_i, i = 1, 2$). Three different sample sizes are used, n = 10^2, 10^3 and 10^4 similar to the previous part. The following bivariate function is used in the

simulation:

$$f(x) = x_1^2 \cos(x_1) + \sin(x_2)^3.$$

Data sets are generated with the following steps:

- Generate $X_{ij}, j = 1, 2$ from standard normal distribution.
- Generate ϵ from standard normal distribution.
- Generate Y by applying a non-linear function (f) on X, then add ϵ, generated from standard normal distribution.

$$Y = f(X) + \epsilon$$

4.1.3.3 Multivariate Case

The third simulation design is created for one dependent (Y) and 5 independent variables ($X_i, i = 1, ..., 5$). The same sample sizes with the previous designs are used. The artificial data created for this simulation is gathered from the study of Friedman (35). The following function is used in the simulation:

$$f(x) = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5.$$

Data sets are generated with the following steps:

- Generate $X_{ij}, j = 1, 2, 3, 4, 5$ from uniform distribution.
- Generate ϵ from standard normal distribution.
- Generate Y by applying a non-linear function (f) on X, then add ϵ.

$$Y = f(X) + \epsilon$$

4.1.4 Simulation Design for Parallel MEKRO Algorithm

To compare the accuracy and elapsed time for this algorithm, we generate i.i.d. samples from the same function that is suggested in (25), as follows;

$$Y = sin(2\pi X_1) + sin(\pi X_2) + 0.5\varepsilon, \tag{4.1}$$

where $X_1, X_2, X_3 \sim U(0,1)$ and $\varepsilon \sim N(0,1)$. The simulation function (10) is demonstrated in the Figure 4.2 with a small sample, generated randomly.

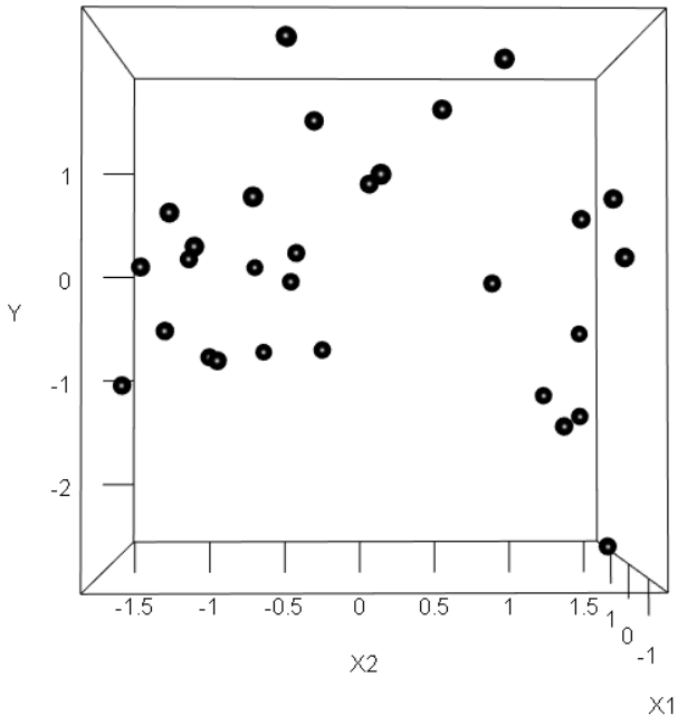

Figure 4.2: Shape of the simulation function

Here we generated artificial data set with three independent variables, but the function is made of 2 of them, X_3 is irrelevant. The reason why X_3 is also added to the data set is that since this is a variable selection method, we expect to see that the corresponding λ_3 value is estimated 0, and this variable is selected out. After the generation of the artificial data, standardization of the variables is done as suggested in (25). The X_1 variable has a higher correlation with Y than the X_2 variable has. While explaining the response variable Y variation, the X_1 variable is more important than the X_2 variable. Moreover, we would expect to see higher λ values obtained for X_1 variable than X_2 variable for every pre-defined τ value.

Simulations were run on a server consisting 8-core Intel E5-2680v3 processors with overall 24-cores. The system has 256 GB memory and runs with Centos 7.3 Linux OS.

4.2 Results and Findings

4.2.1 Simulation Results for Parallel Linear Regression

Results of the simulation study for linear multi-splitting in linear regression models are provided in Table 4.1. Those are the median elapsed time values (in seconds) for 30 distinct simulation scenario. In each sub-table, p represents the number of explanatory variable used and n represents the number of observations in the design. On the rows, 'Seq' means sequential runs and 2, 4 and 8 cores stand for the number of cores used in the design.

According to Table 4.1 for considerably small sample sizes, $n = 10^4$, and for every design $p = 5, 10, 15, 20$, parallelization slows down the algorithm due to the communications amongst to the cores in each iteration. Therefore, using parallel algorithm for small dimensions does not provide any time efficiency. On the other hand, for $n = 10^5$, we are able to see that the parallelization technique starts to shorten the median elapsed time for linear regression algorithm. We expected to achieve faster solutions as we increase the number of cores used in the algorithm however increasing the number of cores does not provide any considerable time efficiency. The reason is the communication between the cores in each step of the iterative solution requires remarkable time. Thus, using higher number of cores does not provide faster solutions. Finally, when the sample size is increased to $n = 10^6$ we can achieve more than 6 times faster solutions for $p = 20$.

Table 4.2 shows the median MSE values for each simulation. Theoretically, MSE values for parallel multi-splitting does not depend on how many cores are used and they are the same for $ncores = 2, 4, 8$ since we use the identity weights in the simulation design. Therefore in the Table 4.2 there is only one line of results for parallel process for each number of parameter.

Table 4.1: Elapsed Time (in seconds), Linear Regression

p=5			
	n=10^4	n=10^5	n=10^6
Seq	0.02094102	0.068717	0.3352962
2 cores	0.03958106	0.051929	0.148092
4 cores	0.05302596	0.06161714	0.2177
8 cores	0.06209302	0.08273697	0.209908
p=10			
	n=10^4	n=10^5	n=10^6
Seq	0.0257411	0.115129	0.8991439
2 cores	0.03916502	0.0546031	0.166157
4 cores	0.05194712	0.06696916	0.238462
8 cores	0.070755	0.0402334	0.2357371
p=15			
	n=10^4	n=10^5	n=10^6
Seq	0.02765393	0.1936979	1.74864
2 cores	0.03973389	0.0660069	0.2341781
4 cores	0.03770029	0.07937002	0.2967241
8 cores	0.07218885	0.0742436	0.412267
p=20			
	n=10^4	n=10^5	n=10^6
Seq	0.03456402	0.1983008	2.0244518
2 cores	0.03986216	0.06616592	0.3562949
4 cores	0.05080605	0.09563994	0.4198401
8 cores	0.0734098	0.135219	0.3306359

Table 4.2: Mean Squared Errors, Linear Regression

p=5			
	n=10^4	n=10^5	n=10^6
Sequential	2.672683	2.097797	2.681294
Parallel	2.672368	2.097772	2.681292
p=10			
	n=10^4	n=10^5	n=10^6
Sequential	1.693587	2.103697	2.562159
Parallel	1.693601	2.103698	2.562159
p=15			
	n=10^4	n=10^5	n=10^6
Sequential	2.264115	1.835099	2.206042
Parallel	2.263897	1.835082	2.206041
p=20			
	n=10^4	n=10^5	n=10^6
Sequential	1.733186	1.46195	2.338848
Parallel	1.733161	1.461949	2.338848

In Table 4.2 the median MSE values for both parallel and sequential this simulation design are compared. According to the results shown in Table 4.2 there is small or no difference between median MSE values of two methods for each design.

4.2.2 Simulation Results for Parallel Kernel Density Estimation

The results of the simulation studies with univariate kernel density estimation are reported. In Table 4.3, we share median elapsed times for a simulation for kernel density estimation.

Table 4.3: Elapsed Time (in seconds), Kernel Density Estimation

	Foreach			parLapply		
	n=10^5	n=10^6	n=10^7	n=10^5	n=10^6	n=10^7
Sequential	0.054896	0.791883	18.78582	0.054896	0.791883	18.78582
2 cores	0.053835	0.524596	9.473815	0.660283	0.704948	9.933074
4 cores	0.041925	0.364080	7.767375	1.241756	0.802907	8.712872
8 cores	0.082816	0.328122	6.534538	2.433246	1.362404	7.910116

Table 4.3 provides us with one main result. The parallelization of KDE using DnR method provides time efficiency for larger samples with both *foreach* and *parLapply* functions, however for relatively smaller sample sizes the parallelization does not provide time efficiency. In fact, the parallelization may increase the elapsed time for relatively small samples.

The simulation results show that the parallelization strategy is crucial in terms of time efficiency for relatively larger samples. For the sample size 10^5, the DnR parallelization does not provide any time efficiency for *foreach* function, moreover for *parLapply* function it significantly slows down the algorithm. For the sample size 10^6, the *foreach* function offers almost 35% faster solution in 2-core parallelization and respectively 55% and 60% faster solutions for 4-core

and 8-core parallelization. However, for the same sample size, although *parLapply* function provides almost 12% faster solution in 2-core parallelization, with 4-core and 8-core parallelization, time efficiency cannot be achieved. Finally, for the sample size 10^7, significant time efficiency is achieved by DnR parallelization with both functions. With 2-core parallelization, approximately 50% and 48% effective results can be achieved with *foreach* and *parLapply* functions respectively. Increasing the number of cores used for this sample size results in a decrease in the median elapsed time for both functions. It is possible to achieve almost 59% and 54% time efficiency with 4-core parallelization using *foreach* and *parLapply* functions respectively. Moreover, 8-core parallelization offers nearly 66% and 58% time efficiency while using *foreach* and *parLapply* functions respectively.

4.2.3 Simulation Results for Parallel Non-Parametric Regression

Three different simulations designs which are univariate, bivariate and multivariate cases for kernel regression are reported with the elapsed time (in seconds), relative change in elapsed time and RMSE. In this part, median values for both run time and RMSE results for a set of simulations for Non-Parametric Regression method are listed on following tables.

Table 4.4 shows the median elapsed times (in seconds) for each simulation design. There are three different sample generating functions, and three different sample sizes in our design. On the top of each sub-tables, p=1,2 and 5 represents the number of parameters used in the sample generating functions. In this section, we compare median elapsed times of 50 different runs of each design for two different function which are *foreach* and *parLapply* introduced earlier.

Table 4.4: Elapsed Time (in seconds), Non-Parametric Regression

p=1	Foreach			parLapply		
	$n=10^2$	$n=10^3$	$n=10^4$	$n=10^2$	$n=10^3$	$n=10^4$
Seq	0.274381	1.787594	152.0201	0.287057	2.130512	189.6113
3 cores	1.666053	1.908610	25.93563	1.710747	2.036216	23.97967
6 cores	1.796663	1.885139	8.323675	1.726394	1.844012	8.137343
12 cores	1.960821	2.008906	3.785587	1.837807	1.872109	3.537981
24 cores	2.558390	2.462539	2.993848	2.324590	2.376621	2.933135
p=2	Foreach			parLapply		
	$n=10^2$	$n=10^3$	$n=10^4$	$n=10^2$	$n=10^3$	$n=10^4$
Seq	0.481238	18.29995	1684.813	0.475189	18.51795	1761.300
3 cores	1.679229	4.780826	267.7928	1.673428	4.298342	219.2693
6 cores	1.868309	2.508023	80.25953	1.789368	2.489918	64.82297
12 cores	1.977318	2.127511	22.17067	1.799092	2.053200	19.82452
24 cores	2.547829	2.531299	7.455359	2.357470	2.457566	7.097829
p=5	Foreach			parLapply		
	$n=10^2$	$n=10^3$	$n=10^4$	$n=10^2$	$n=10^3$	$n=10^4$
Seq	5.647150	248.60433	24566.82	5.46708667	274.2403	25877.04
3 cores	2.785601	43.880413	2810.537	2.57744431	42.78230	2869.852
6 cores	2.176856	17.904519	757.3612	1.97839820	18.49372	724.2150
12 cores	2.128459	7.5005460	226.9664	1.99841415	7.400818	219.3958
24 cores	2.375777	4.0321351	79.12754	2.26951777	3.855904	82.57617

Although the sequential runs are completely the same for each part of the simulation, the median elapsed times of sequential runs vary a little. The reason why they differ slightly is that work load on the server that we run our simulations may change from time to time. In order to eliminate the unbalanced work load on the server, line graphs in Figure 4.3 are provided, which are illustrating the ratios of median elapsed times.

The first major results from Table 4.4 is that the parallelization offers significant time efficiency for relatively medium and large samples, $n = 10^3$ and 10^4. However, for considerably small samples, $n = 10^2$ and small number of parameters (p=1,2), the method does not provide sufficient time efficiency, in fact it requires more time to complete the kernel regression algorithm compared to sequential run. Additionally, when we increase the number of independent variables to 5, we can see a decrease in the median elapsed time even in smaller samples. Considering that the increase in data size continues, it is understood that the effort to be spent on parallelization on statistical methods is substantial.

Table 4.4 also proves that increasing the number of cores used yields a significant time efficiency for bigger sample designs for each case. Except for the first design includes only one independent parameter, the increase in the number of cores used provides time efficiency for 'medium sized' samples, too. As we discussed earlier, the parallelization of the kernel regression algorithm and parallelization with higher number of cores results in an increase in the median elapsed time for 'small sized' samples.

Figure 4.3 represents the ratio of elapsed time of parallel runs relative to the sequential run. On the left hand side, line graphs show the ratio of elapsed times for *foreach* function and on the right hand side line graphs represent the ratios of elapsed time for *parLapply* function.

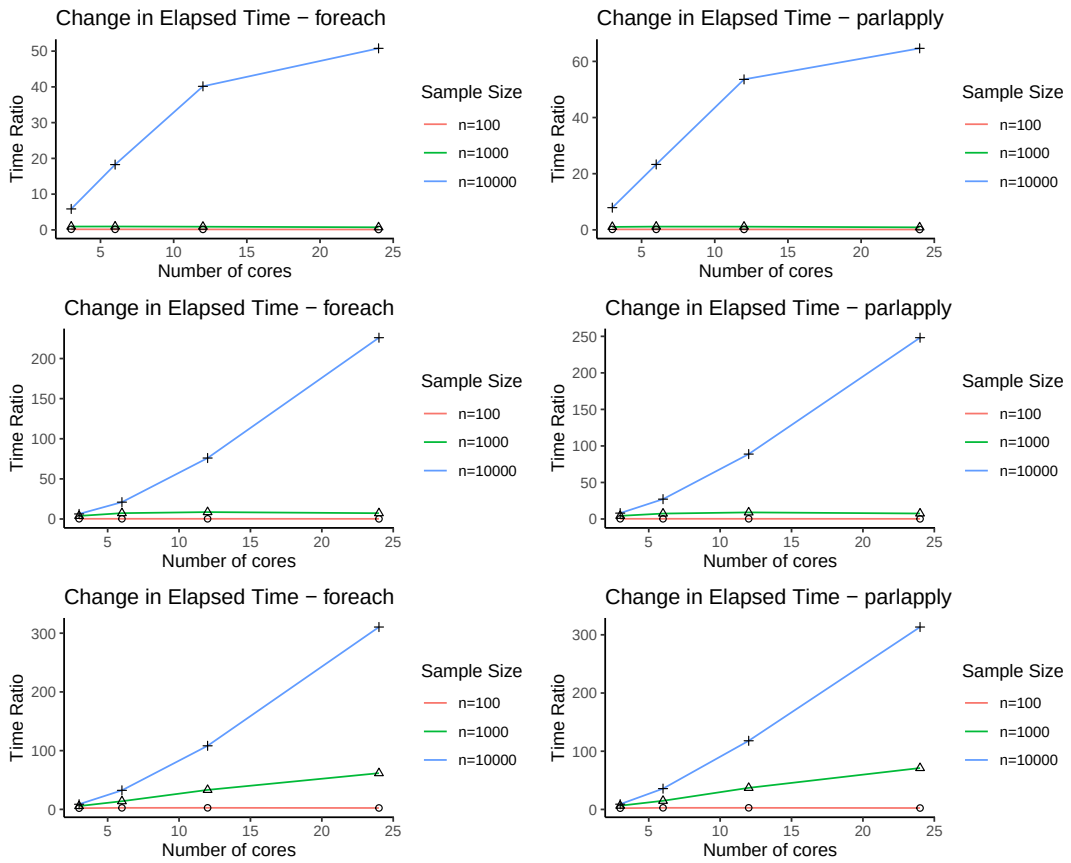

Figure 4.3: Change in Elapsed Time Relative to Sequential Process

General and similar conclusions can be drawn from these graphs. The parallelization for relatively small and medium sample sizes does not provide a significant time efficiency with 1 or 2 independent variables. However, when we increase the number of independent variables to 5, we are able to see that for n=10^3, parallelization provides an opportunity to run the algorithm remarkably fast.

Additionally, when the sample size is considerably high, 10^4, we can see a significant increment in the ratio of elapsed times. This increment seems to be correlated with the number of cores. For univariate case (p=1), we see that doubling the number of cores used after 12 does not provide a linear speed-up. Apart from that scenario, for all scenarios we are able to see that there is a potential linear speed-up for parallel kernel regression.

Finally, although ratios of elapsed times for each function (*foreach* and *parLapply*) seem to be really close to each other, there are slight differences. Apparently, *parLapply* brings slightly higher speed-up compared to *foreach* function for kernel regression method. For example, running the kernel regression algorithm with *foreach* 24-core makes it is possible to achieve 310.47 times faster solutions compared to the sequential ones for multivariate case (p=5), and for bigger sample sizes (n=10^4). However, on the same conditions, *parLapply* provides 313.37 times faster solutions compared to sequential run. This slight differences can be seen for all three different simulation designs with different number of parameters.

Table 4.5 represents the median Root Mean Squared Error for each simulation design. We have three different sample generating functions and three different sample sizes similar to Table 4.4. On the top of each sub-tables, the number of parameters used in the sample generating functions are indicated (p=1,2 and 5). Unlike the Table 4.4, we do not have two separate results for each functions because all the RMSE results are exactly the same for both functions.

Table 4.5: Root Mean Squared Errors, Non-Parametric Regression

p=1	RMSE		
	n=10^2	n=10^3	n=10^4
Sequential	1.4140619	1.5043147	1.6313355
3 cores	1.3904418	1.5056749	1.6313433
6 cores	1.3943259	1.5047113	1.6313042
12 cores	1.3982014	1.5057129	1.6314233
24 cores	1.3997703	1.5049241	1.6314469
p=2	RMSE		
	n=10^2	n=10^3	n=10^4
Sequential	1.1922569	1.3429781	1.2961050
3 cores	1.1901615	1.3437884	1.2965671
6 cores	1.2088067	1.3474531	1.2963941
12 cores	1.1873848	1.3451843	1.2979357
24 cores	1.1916416	1.3458324	1.2997028
p=5	RMSE		
	n=10^2	n=10^3	n=10^4
Sequential	2.8216695	1.8993387	1.609711
3 cores	2.8429371	1.9875962	1.638009
6 cores	3.1270144	2.1074268	1.684394
12 cores	3.2829552	2.3426828	1.762903
24 cores	4.1298523	2.5496637	1.885775

From the Table 4.5 table, we are able to see that parallelization of the kernel regression algorithm provides slight decrease in the median RMSE value of small sample (n=10^2), for univariate case (p=1). For this case, there is no significant difference in terms of RMSE between parallel runs and sequential runs for n=10^3 and 10^4. For bivariate case where we have two independent variables for kernel regression, for medium and bigger sized samples (n=10^3, 10^4), there is no significant difference between median RMSE values of parallel and sequential runs. However, in bivariate simulation design, we see that for small sample size, the median RMSE values fluctuate when we increase the number of cores used. Finally, for multivariate case, the increase in the median RMSE is significant. Although the difference between 3-core parallel run and sequential run may be considered negligible for multivariate case, up to 4.5%, increasing the number of cores used results in an increase in the median RMSE value. For small sample size, we can see up to 46%, 34% and 17% increase in the RMSE value respectively for each different sample size respectively.

4.2.4 Simulation Results for Parallel MEKRO Algorithm

In this section, simulation results for the MEKRO algorithm is provided.

Figure 4.4 shows the solution path for some values of the hyperparameter τ. Recall that the first and the second variable is the ones we generated the response variable, described in Section 4. The third one is the 'irrelevant' variable. $\lambda_1, \lambda_2, \lambda_3$ are the MEKRO parameters for X_1, X_2, X_3. From Figure 3.1, for sample size 50, λ_3 is estimated 0 for $\tau = 1, 2, 3$ and then it stars to increase. For sample sizes 100 and 200 the value for λ_3 is estimated 0 for one more step of increase in τ, $\tau = 1, 2, 3, 4$. However, for the sample size 400, λ_3 value is shrunk to 0 for $\tau = 1, 2, 3, 4, 5$. Therefore, this is an indicator that as the sample size increases the algorithm can detect the irrelevant variables and shrunk their effects better.

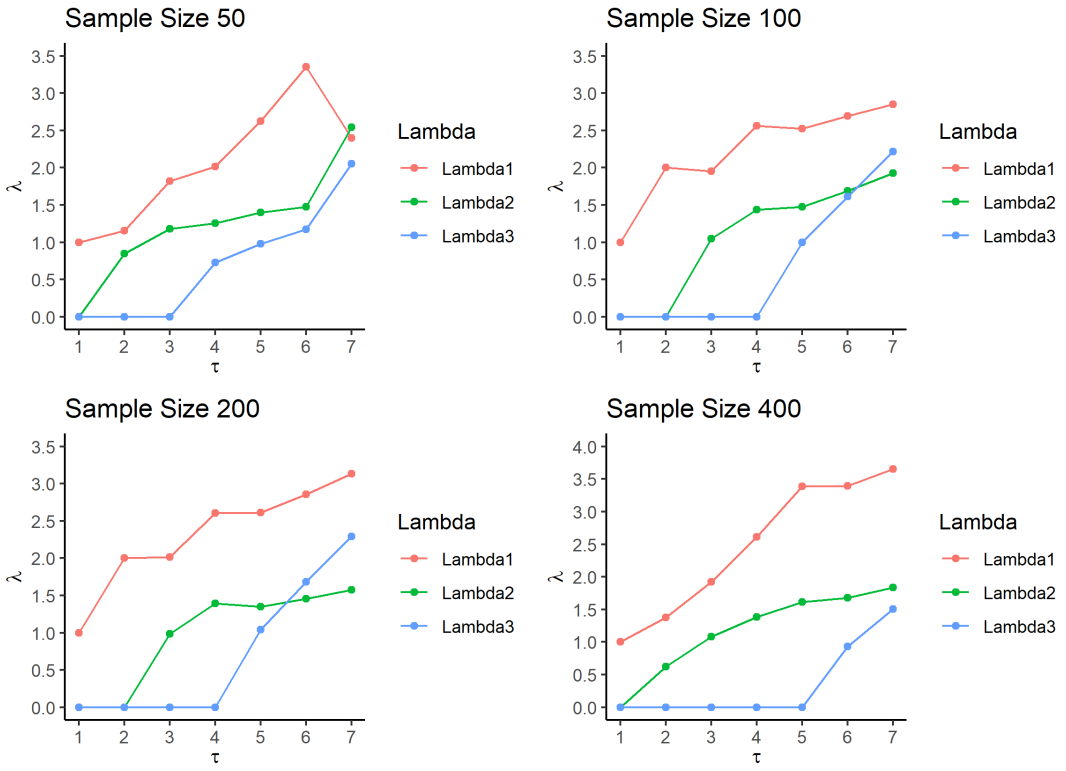

Figure 4.4: Change in λ over τ

Figure 4.5 represents the median relative elapsed time for MEKRO algorithm running with different number of cores. Relative Time, is calculated as following;

$$\text{Relative Time} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

In Figure 4.5, each line represents the relative elapsed time for each sample size in the simulation design. As it is expected, the parallelization technique provides significantly faster solutions. For sample size 50, we are able to achieve almost 5 times faster running algorithm when 12 cores are used. However, increasing the number of cores used 12 to 24 does not provide any considerable time efficiency for sample size 50. When the sample size is increased to 100, up to 12 cores we are able to achieve almost linear speed-up, but after 12 cores the speed-up potential does not follow a linear pattern. However, for sample size 200 and 400 we are able to achieve linear speed-up for this algorithm. With the parallelization method, this MEKRO algorithm can be run up to 16 times faster for a data set with 3 variables and 400 observations.

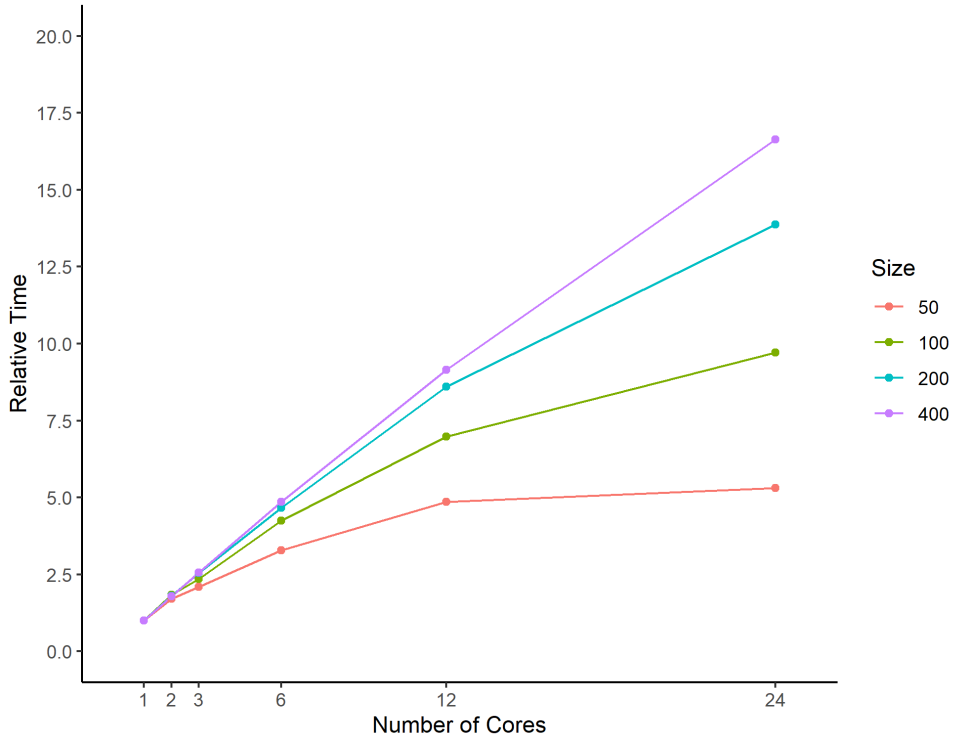

Figure 4.5: Parallel Elapsed Time Relative to Sequential Elapsed Time

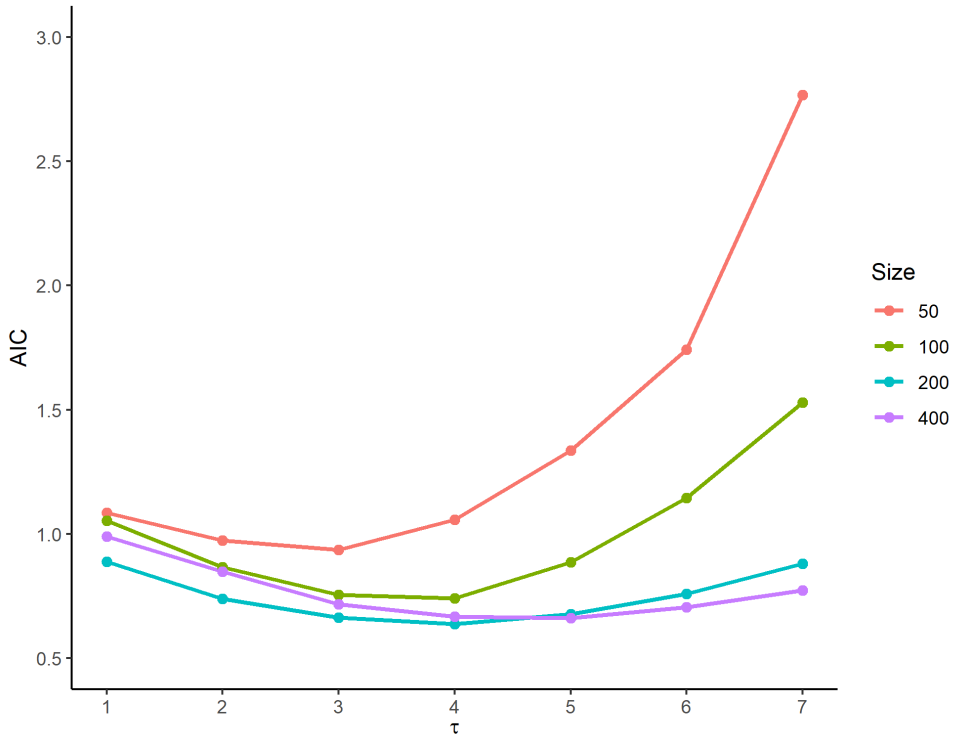

Figure 4.6: Change in AIC, over tuning parameter τ

Table 4.6: Elapsed Time (in seconds), MEKRO Algorithm

n=50	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 5$	$\tau = 6$	$\tau = 7$
Seq	521.9719	416.1632	485.2282	428.4259	371.2660	394.0977	314.0572
2 cores	302.1288	242.9140	282.4885	249.6838	216.0665	229.9969	184.6151
3 cores	244.8393	199.0052	230.1185	203.6477	176.4736	187.1028	149.6270
6 cores	158.0615	126.9947	147.9697	130.4784	113.3307	120.3874	96.08578
12 cores	107.6001	85.68309	99.47257	88.10784	76.95138	80.94674	64.59627
24 cores	96.22435	78.44308	91.35289	80.27536	69.13914	73.45763	59.08070
n=100	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 5$	$\tau = 6$	$\tau = 7$
Seq	1680.522	1489.918	1695.172	1687.472	1371.502	1294.625	1171.9943
2 cores	917.3809	808.1143	918.7912	921.9849	744.3063	692.9727	633.1621
3 cores	718.7200	633.5475	719.4204	724.1402	581.6202	551.8655	497.7785
6 cores	399.7617	350.6442	399.9367	402.2911	325.3113	305.3619	276.6470
12 cores	242.2070	213.4682	243.6693	243.3268	196.5807	185.2893	167.8158
24 cores	174.4567	153.3412	176.3156	175.1280	142.6010	134.0309	121.6185
n=200	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 5$	$\tau = 6$	$\tau = 7$
Seq	6452.619	6467.589	5559.839	6257.362	4994.518	5125.853	4979.012
2 cores	3618.718	3598.480	3086.759	3305.213	2638.192	2701.537	2592.538
3 cores	2551.116	2551.184	2190.019	2549.617	2032.880	2095.583	2053.382
6 cores	1384.625	1388.745	1191.854	1313.7294	1054.119	1086.111	1053.500
12 cores	757.1868	755.7779	645.8416	719.1007	572.2419	591.9123	572.5263
24 cores	470.3340	470.9377	400.9010	432.9642	348.4030	355.4760	346.2039
n=400	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 5$	$\tau = 6$	$\tau = 7$
Seq	24065.62	24643.37	23612.42	21860.69	21146.30	19301.47	17715.21
2 cores	13431.95	13927.86	13183.43	12180.44	11962.69	10718.70	9942.21
3 cores	9379.325	9597.791	9228.918	8529.242	8247.296	7532.441	6918.769
6 cores	5017.795	5130.878	4903.443	4501.471	4381.930	3995.138	3739.085
12 cores	2628.232	2706.231	2578.861	2388.798	2317.873	2106.292	1968.726
24 cores	1457.633	1503.371	1422.440	1313.570	1273.339	1165.053	1073.262

Elapsed time results is given in the Table 4.6. As Table 4.6 suggests that the parallelization of the algorithm offers a great potential of decrease in the elapsed time for almost all cases. Parallelization of the process with only 2 cores, results in almost 2 times faster solutions. However, for small sample sizes increasing the number of cores used, or in other words number of chunks that the algorithm is split, is not always producing linear speed up. However, when the sample size is 400, there is a great linear speed up potential that is offered by parallelization of the algorithm.

The subtraction part in the algorithm slows the calculations down, and it is easily seen by looking at the elapsed time values for sequential runs given in Table 4.6. For example, doubling the sample size 50 to 100 for $\tau = 1$, the algorithm runs 3 times slower. Again, doubling the sample size one more step, 100 to 200, the algorithm runs 4 times slower.

Additionally, Table 4.6 suggests that for higher τ values the algorithm converges shortly compared to the smaller τ values. Faster solutions are not always the better solutions. Calculated AIC values shown in Figure 4.6, should be another indicator while selecting the best model. In our simulation design, it is appeared to be that setting the hyper parameter $\tau = 5$ is giving the optimal results, although $\tau = 7$ provides generally faster solutions.

CHAPTER 5

CONCLUSION

The aim of this study is to apply parallelization techniques to some statistical algorithms, and raise awareness of the efficiency that parallelization can offer. In this study, 4 different algorithms, linear multi-splitting method for MLR, KDE, non-parametric regression and MEKRO algorithm for variable selection in non-parametric regression, are introduced and their simulation results are provided in order to compare efficiency that is offered by parallelization. Our main goal is to achieve time efficiency while maintaining the accuracy of the estimations and predictions.

Firstly, we introduced the OLS problem for linear regression models. With iterative multi-splitting technique, OLS problem can be distributed on several local OLS problems and properly combining their results in an iterative method. For the ease of coding, we suggested a method in the initial part of the algorithm and the simulation study is conducted on our suggested method. According to the results of the simulation study in Chapter 4, parallelization of the LS algorithm using linear multisplitting method does not affect the accuracy of the models as shown in the Table 4.2. Also, it provides time efficiency for high dimensions. However, it does not provide time efficiency for smaller data sets due to the fact that its iterative solution path requires communication between threads and that slows down the algorithm.

Secondly, we discuss the kernel density and kernel regression in parallel with the DnR method. Two main conclusions can be drawn from the simulation results. Firstly, while estimating kernel densities, DnR parallelization provides time efficiency for relatively medium and large sample sizes. However, for rel-

atively smaller sample sizes achieving time efficiency seems very unlikely while using DnR parallelization. Additionally, *foreach* function provides better results compared to *parLapply* function in KDE. The second main result is that the parallelization of kernel regression algorithm offers significant time efficiency for data sets with higher dimensions. We see that parallelization offers more than 300 times faster results for bigger samples. However, it appears that we can not achieve a time efficiency while running the algorithm in parallel except for multivariate case for small samples.

Additionally, parallelization of the process brings a trade-off with itself. In some designs or some sub-cases of simulation designs, varying amount of increase are observed in the median RMSE values. This increase in most cases can be considered negligible, however there is a significant increase in the median RMSE for small sample sizes in multivariate case compared to sequential run. For that type of modeling design, we can suggest users not to divide the process to higher cores, instead running the algorithm with 3 or 6 cores results in better RMSE values and still they are running significantly faster compared to sequential ones.

Finally, we propose a parallel version of MEKRO algorithm. Thus, the proposed version performs faster solutions compared to the sequential version of the same algorithm. The traditional version of the algorithm includes a iterative subtraction operation which causes computation burden on CPU. However, since the subtraction operation in the algorithm can be split into several independent chunks and provides a potential speed-up. Table 4.6 shows that the traditional algorithm, even for considerably small sample sizes, needs significantly high computational time and power. However, we managed to decrease the required time to complete the algorithm.

All in all, in this study we wished to introduce concept of parallel computing and its applications in statistics. Based on our simulation studies, parallel computation methods offer a remarkable time efficiency. Thanks to the new technologies, the data that we can reach is growing exponentially and scientists have been developing new methods day-by-day and yet still there is so much to discover.

Bibliography

[1] G. Guo, "Parallel statistical computing for statistical inference," *Journal of Statistical Theory and Practice*, vol. 6, no. 3, pp. 536–565, 2012.

[2] N. Matloff, "Software alchemy: Turning complex statistical computations into embarrassingly-parallel ones," *Journal of Statistical Software*, vol. 71, no. 4, p. 1–15, 2016.

[3] R. A. Renaut, "A parallel multisplitting solution of the least squares problem," *Numerical Linear Algebra with Applications*, vol. 5, no. 1, pp. 11–31, 1998.

[4] D. P. O'leary and R. E. White, "Multi-splittings of matrices and parallel solution of linear systems," *SIAM Journal on Algebraic Discrete Methods*, vol. 6, no. 4, pp. 630–640, 1985.

[5] J.-J. Climent and C. Perea, "Iterative methods for least-square problems based on proper splittings," *Journal of Computational and Applied Mathematics*, vol. 158, no. 1, pp. 43–48, 2003.

[6] T. Hayfield and J. S. Racine, "Nonparametric econometrics: The np package," *Journal of Statistical Software*, vol. 27, no. 5, 2008.

[7] Z.-Z. Bai, "On the convergence of additive and multiplicative splitting iterations for systems of linear equations," *Journal of Computational and Applied Mathematics*, vol. 154, no. 1, pp. 195–214, 2003.

[8] D. B. Szyld and M. T. Jones, "Two-stage and multisplitting methods for the parallel solution of linear systems," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 2, pp. 671–679, 1992.

[9] M. Rosenblatt, "Remarks on Some Nonparametric Estimates of a Density Function," *The Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 832 – 837, 1956.

[10] H. Takeda, S. Farsiu, and P. Milanfar, "Kernel regression for image processing and reconstruction," *IEEE Transactions on Image Processing*, vol. 16, no. 2, pp. 349–366, 2007.

[11] A. Yatchew, "Nonparametric regression techniques in economics," *Journal of Economic Literature*, vol. 36, no. 2, pp. 669–721, 1998.

[12] P. J. Diggle and E. Giorgi, *Model-based geostatistics for global public health: methods and applications*. CRC Press, 2019.

[13] T. Wen, F. Yang, J. Gu, S. Chen, L. Wang, and Y. Xie, "An adaptive kernel regression method for 3d ultrasound reconstruction using speckle prior and parallel gpu implementation," *Neurocomputing*, vol. 275, pp. 208–223, 2018.

[14] J. Racine, "Parallel distributed kernel estimation," *Computational Statistics and Data Analysis*, vol. 40, no. 2, pp. 293–302, 2002.

[15] P. D. Michailidis and K. G. Margaritis, "Parallel computing of kernel density estimation with different multi-core programming models," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 77–85, IEEE, 2013.

[16] U. Lopez-Novoa, J. Sáenz, A. Mendiburu, and J. Miguel-Alonso, "An efficient implementation of kernel density estimation for multi-core and many-core architectures," *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, pp. 331–347, 2015.

[17] H. Adeli and P. Vishnubhotla, "Parallel processing," *Computer-Aided Civil and Infrastructure Engineering*, vol. 2, no. 3, pp. 257–269, 1987.

[18] T. Hayfield, J. S. Racine, and M. J. S. Racine, "Package 'nprmpi'," 2013.

[19] H. Yu, "Rmpi: Parallel statistical computing in r," *R News*, vol. 2, no. 2, pp. 10–14, 2002.

[20] A. T. Ho, K. P. Huynh, and D. T. Jacho-Chavez, "nprmpi: A package for parallel distributed kernel estimation in r," 2011.

[21] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[22] E. Kartal Koc and H. Bozdogan, "Model selection in multivariate adaptive regression splines (mars) using information complexity as the fitness function," *Machine Learning*, vol. 101, pp. 35–58, Oct 2015.

[23] Y. Lin and H. H. Zhang, "Component selection and smoothing in multivariate nonparametric regression," *The Annals of Statistics*, vol. 34, Oct 2006.

[24] J. Lafferty and L. Wasserman, "Rodeo: Sparse, greedy nonparametric regression," *The Annals of Statistics*, vol. 36, no. 1, pp. 28 – 63, 2008.

[25] K. White, L. Stefanski, and Y. Wu, "Variable selection in kernel regression using measurement error selection likelihoods," *Journal of the American Statistical Association*, vol. 112, 08 2016.

[26] L. Stefanski, Y. Wu, and K. White, "Variable selection in nonparametric classification via measurement error model selection likelihoods," *Journal of the American Statistical Association*, vol. 109, no. 506, pp. 574–589, 2014.

[27] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[28] E. A. Nadaraya, "On non-parametric estimates of density functions and regression curves," *Theory of Probability & Its Applications*, vol. 10, no. 1, pp. 186–190, 1965.

[29] A. Berman and R. J. Plemmons, "Cones and iterative methods for best least squares solutions of linear systems," *SIAM Journal on Numerical Analysis*, vol. 11, no. 1, pp. 145–154, 1974.

[30] S. Łukasik, "Parallel computing of kernel density estimates with mpi," in *International Conference on Computational Science*, pp. 726–733, Springer, 2007.

[31] R. Calaway, S. Weston, and M. R. Calaway, "Package 'foreach'," *R package*, pp. 1–10, 2015.

[32] S. Weston, "Using the foreach package," 2019.

[33] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021.

[34] J. Racine and Q. Li, "Nonparametric estimation of regression functions with both categorical and continuous data," *Journal of Econometrics*, vol. 119, no. 1, pp. 99–130, 2004.

[35] J. H. Friedman, "Multivariate adaptive regression splines," *The Annals of Statistics*, pp. 1–67, 1991.