DSC520: Running Random Forest on R using Stampede2

Anubhav Shankar (ashankar@umassd.edu)

Deena Kurapati (dkurapati@umassd.edu)

Dhyey Doshi (ddoshi@umassd.edu)

College of Engineering
University of Massachusetts Dartmouth
Dartmouth, United States of America

***Abstract*: Stampede2 is a powerful super-computer primarily utilized for running computationally expensive tasks, especially parallelization. The primary goal of this project is to get a basic understanding of the Random Forest algorithm and run the same on Stampede2 with scaling to assess how parallelization can help speed up similar tasks.**

Keywords*: *High-Performance Computing (HPC), Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS), Random Forest (RF)

I. MOTIVATION

This undertaking aims to achieve the following objectives:

a.) To gain a primary understanding of the Random Forest Algorithm
b.) Gain familiarity with R Programming Language
c.) Gain a more thorough understanding of scaling processes, core allocation, and running programs on Stampede2

II. INTRODUCTION – DATA SOURCE

A. Dataset

For this project, we'll use the Titanic dataset from Kaggle. The dataset is part of the competition to predict the passengers' survival status on the Titanic post its capsizing. The data has been 80:20 split into Training(891 records) and Testing(418 records) sets for the competition, and I'll be using the same for my analysis[1].

B. Dataset Specifications

The table below gives the variable definition and some additional notes/keys regarding the same:

Variable	Definition	Key
survival	Survival Status	0 = Died, 1 = Survived
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
Sex	Gender of the passenger	
Age	Age in years	
sibsp	# of siblings/spouses aboard the Titanic	
Parch	# of parents/children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

TABLE 1: DATASET SPECIFICATIONS[1]

III. METHODOLOGY & TECH-STACK

Before we proceed, let's have a quick introduction to our tech stack and what we mean by Random Forests and High-Performance Computing for this project:

1.) **R**: is a programming language for statistical computing and graphics supported by the R Core Team and the R Foundation for Statistical Computing. R is used among data miners, bioinformaticians, and statisticians for data analysis and developing statistical software. Users have created packages to augment the functions of the R language.

2.) **Stampede2**: TACC's Stampede2 is the flagship supercomputer of the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS), a single virtual system that scientists can use to share computing resources, data, and expertise interactively.

Stampede2 has Over 56,000 computer nodes with Intel's Knights Landing and Skylake chips. Each node has 48 - 68 compute cores (nearly 400,000 cores in total). Basically, 100,000 desktops connectors that have 96GB - 198 GB of memory per node cost: over $30,000.

Following are the steps to run R on Stampede2 –

- Request an account on Stamped2 : https://allocations.access-ci.org/login.
- Login with the username. To connect, put the command on the terminal: ssh -A username@stampede2.tacc.utexas.edu
- Install R in the working directory using anaconda
- Run the R scripts either using standalone commands or through a shell-script

High-Performance Computing: Clusters of powerful processors working parallel to process massive datasets and solve complex problems at high speeds. Thus, helping uncover critical new insights that advance human knowledge and create significant competitive advantage. Examples include weather forecasting, automated stock-market training, etc.

Random Forest Algorithm: The random forest is a classification algorithm consisting of many decision trees. It uses bagging and feature randomness when building each tree to try to create an uncorrelated forest of trees whose prediction by committee is more accurate than that of any individual tree.

A. Abbreviations and Acronyms

The following abbreviations will be used extensively throughout the document for brevity:

i.) RF -> Random Forest

ii.) HPC -> High-Performance Computing

iii.) TACC -> Texas Advanced Computing Center

iv.) CRAN -> Comprehensive R Archive Network

B. Problem Statement

Predict the survivability of a passenger onboard the Titanic with a Random Forest algorithm. Additionally, run the R script on Stampede2 along with a strong scaling parallelization to observe relevant speedup and scaleup.

For the project, we used the following packages:

Package	Purpose
Foreach	It is true horsepower of parallel processing in R. Uses *%dopar%* to parallelize tasks and returns it at a list of results vectors.
doParallel	Provides parallel backend for *%dopar%* function.
randomForest	Builds a number of decision trees. Uses *foreach* and the *combine* function to get parallelization.
Caret	Provides framework to find optimal model by trying multiple models with resampling.

Let's briefly look at the package "randomForest" and a few of its key arguments. The package randomForest implements Breiman's random forest algorithm (based on Breiman and Cutler's original Fortran code) for classification and regression. It can also be used unsupervised to assess data points' proximities.

Here's a screenshot of the CRAN documentation[9] for the package:

Usage

```
## S3 method for class 'formula'
randomForest(formula, data=NULL, ..., subset, na.action=na.fail)
## Default S3 method:
randomForest(x, y=NULL,  xtest=NULL, ytest=NULL, ntree=500,
             mtry=if (!is.null(y) && !is.factor(y))
             max(floor(ncol(x)/3), 1) else floor(sqrt(ncol(x))),
             weights=NULL,
             replace=TRUE, classwt=NULL, cutoff, strata,
             sampsize = if (replace) nrow(x) else ceiling(.632*nrow(x)),
             nodesize = if (!is.null(y) && !is.factor(y)) 5 else 1,
             maxnodes = NULL,
             importance=FALSE, localImp=FALSE, nPerm=1,
             proximity, oob.prox=proximity,
             norm.votes=TRUE, do.trace=FALSE,
             keep.forest=!is.null(y) && is.null(xtest), corr.bias=FALSE,
             keep.inbag=FALSE, ...)
## S3 method for class 'randomForest'
print(x, ...)
```

For our purpose, two arguments, "mtry" and "ntree," hold the most relevance. Let's have a look at them briefly.

a.) **Mtry**: Specifies the number of variables randomly sampled as candidates at each split.
b.) **Ntree**: Number of trees to grow. This should be set to a manageable number to ensure that every input row gets predicted at least a few times.

C. Implementation

We implemented our code's Serial and Parallel versions locally and on Stampede2. Let us now look at each version separately to understand them.

a.) **Serial**: Typically, a problem is divided into instructions, and each instruction is executed sequentially on a single processor. Only one instruction can be executed at a particular time.

In serial execution of random forest, several decision trees are generated sequentially and aggregated. The class with the maximum votes of predictions from all decision trees is the predicted class.

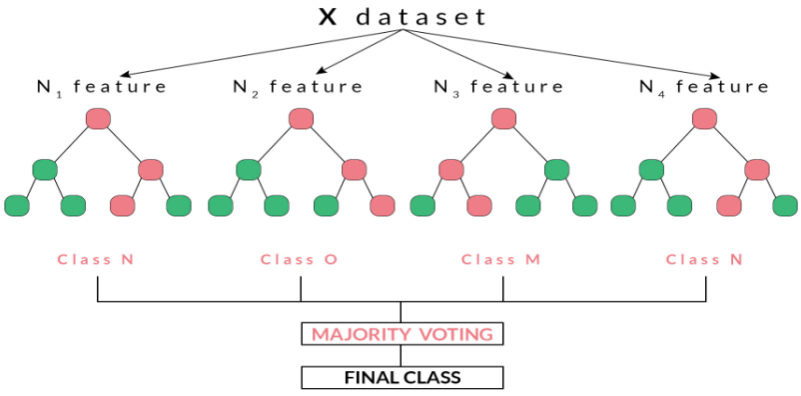


Figure 1: Serial Execution of Random Forest [7]

Additionally, we have implemented cross-validation on the serial version. We start with a kitchen-sink approach and steadily go on making our RF model parsimonious basis the variable importance obtained. Concurrently, cross-validation also helps increase our overall accuracy, given that the dataset is small, and RFs tend to underperform with small datasets.

The runtime was captured using the "tictoc" package with the timer beginning with the tic() and ending with the toc() command, respectively.

b.) **Parallel**: In this approach, the problem is broken into discrete parts that can be solved simultaneously. Each part is further broken down into a series of instructions. Instructions from each part execute concurrently on different processors, for which an overall control and coordination mechanism is employed. There are different types of parallel computing, like multi-threading and multi-processing.

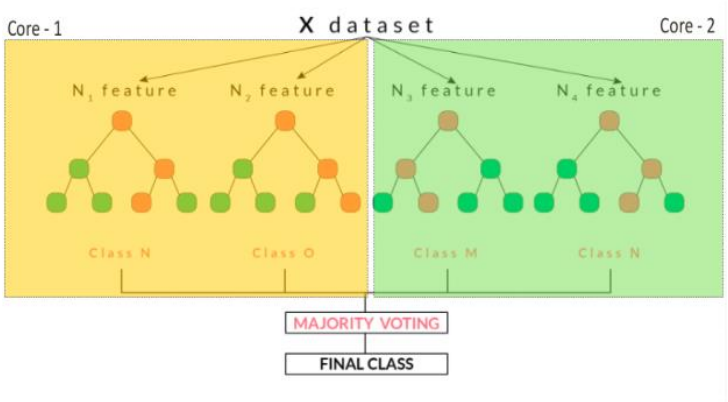


Figure 2: Parallel Implementation of Random Forests [7]

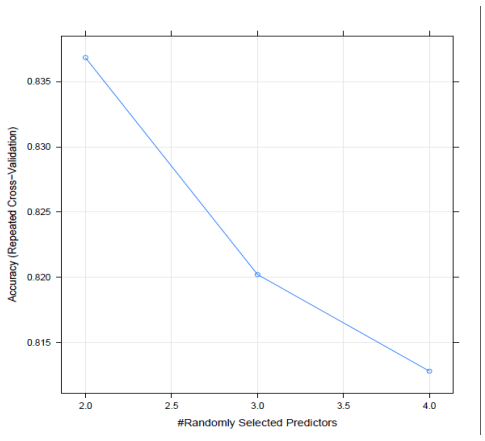
Unlike the Serial approach, we stuck to the kitchen-sink approach here. However, we did notice a significant decline in the run-time, which was even more pronounced than the run-time of the parsimonious model in our serial execution.

Given that our input size would not change, a Strong Scaling was the only plausibility. Hence, we continuously increased the number of cores with the help of the "registerDoParallel(cores = i)" and used the "getDoParWorkers()" command to run the code multiple times.

The execution time was captured using the Sys.Time() command.

IV. OBSERVATIONS & EVALUATION

The error rate of the Serial approach is shown below:

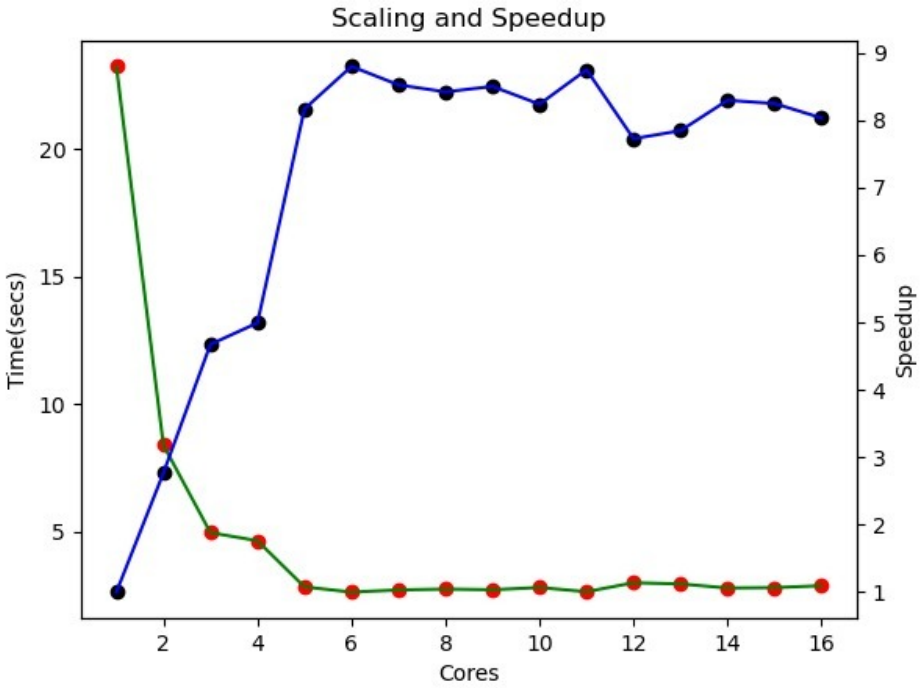


From the above graph, it is clear that a random sample of two variables yields the highest accuracy on the Train set with an accuracy of ~83%. Concurrently, the Test set accuracy for the parsimonious model was also ~81%.

The time taken to run the serial approach ranged from 29.1 – 33.7 seconds on our local systems and 23.3 – 26.5 seconds on Stampede2 across the three models created.

In the parallel approach, we were more concerned with the time decrement across multiple core allocations to see whether our scaling expectations pan out. Consequently, we noticed a significant time decrement with the Parallel run even though, especially on Stampede2, we had taken a kitchen-sink approach where some feature-engineered variables were also included.

The dual-axis graph below shows the scale-up and speed-up of the parallel process:



As expected, the time decreases significantly as the number of core allocations increases. Concurrently, the speed-up factor increases as the number of cores allocated increases, as expected from the formula ->

$$T_1/T_p$$

Where T_1 = time required by one core; for our purposes, we've taken 23.35s, the time needed to run our parsimonious serial model.

T_p = Time required by successive 'p' allocated cores.

V. CHALLENGES & FUTURE SCOPE

While things aligned with our expectations, the following are some of the challenges that we faced:

1.) Stampede2 proved tricky for us initially as we faced some unexpected hurdles in the form of broken packages, which prevented us from running our scripts successfully. To mitigate the same, we tried to run it in an Anaconda environment; however, owing to space constraints, we dropped this and ran it

through a miniconda, but the performance was not up to par. Finally, we switched to the "/work" directory on Stampede2 and did the needful by creating a functional Conda environment.

2.) As mentioned in Section III-C, Random Forests tend to underperform with small datasets compared to other "simpler" methods like Logistic Regression. Test set accuracy for our Parallel approach was only ~62%, whereas Logistic Regression easily obtained an accuracy of ~76% [8].

3.) This is a modest attempt to understand Random Forests and parallelization. The model can easily be made more robust by cross-validation across multiple cores.

VI. References

[1] **Kaggle** -> https://www.kaggle.com/competitions/titanic

[2] https://computing.llnl.gov/tutorials/parallel_comp/

[3] https://math.unm.edu/~motamed/Teaching/OLD/Fall20/HPSC/parallel.html

[4] https://docs.anaconda.com/anaconda/user-guide/tasks/using-r-language/

[5] https://www.r-bloggers.com/2021/04/random-forest-in-r/

[6] https://www.systranbox.com/how-to-get-started-with-r-in-linux/

[7] https://github.com/AnushreeChopde/teaching-examples/tree/master/R/random_forest

[8] MTH522 Project2: Predicting Passenger Survivability using Supervised Machine Learning, Anubhav Shankar, College of Engineering, University of Massachusetts, Dartmouth

[9] https://cran.r-project.org/web/packages/randomForest/index.html