CHAPTER 6

Parallel Algorithm for Random Forest Classifier

Random Forest classification algorithm can be easily parallelized due to its inherent parallel nature. Being an ensemble, the parallel implementation of Random Forest classification algorithm yields better time efficiency than their sequential implementations. This has led to various parallel implementations of Random Forest classifiers. Based on our survey, we found that each such implementation is specific to some platform and uses a different programming language. Hence, we strongly felt the need for presenting a generalized parallel algorithm for Random Forest classifier. We also found that all these implementations follow the task parallel approach. They generate the trees of Random Forest in parallel. We came up with a new approach for implementing Random Forest in parallel. In our approach, along with generating the trees in parallel, the individual decision trees are also parallelized. Hence, we present below our work related to parallel Random Forest in two parts. In the first part, we present a generalized parallel algorithm for Random Forest classifier. We also analyze the time complexity and present the speedup. In the second part, we present a new approach for implementing parallel Random Forest in which along with parallelization of base trees, individual trees are also parallelized.

Parallel algorithms are run on parallel machines. The most important measure of performance of a parallel algorithm is how fast a given problem can be solved using available number of processors. Since a parallel algorithm uses several processors, the communication performance among the processors is also important. Speedup [60] is an important performance measure for a parallel algorithm, when mapped onto a given architecture. Speedup is defined as the ratio of the worst case running time of the best known sequential algorithm and the worst case running time of the parallel algorithm. Greater is the value of speedup, better is the parallel algorithm.

6.1 Analysis of Parallel Algorithms

A sequential algorithm is normally evaluated in terms of time complexity and space complexity. Similarly, the criteria normally used for analyzing a parallel algorithm are: Running time, Number of processors, and cost.

Running time is defined as the time taken by the algorithm to execute and produce the desired result on a parallel computer. Running time is also a function of the size of the input. Moreover, Running time of a parallel algorithm involves time spent in computation and communication. During computation, a processor performs local arithmetic or logic operations, while during communication, processors exchange data using shared memory or message passing mechanisms. Thus, the running time of a parallel algorithm includes the time spent in both computation and communication. In our analysis, we have considered the time taken by computation time only. Communication time depends on the underlying architecture and the mode of communication.

Cost of a parallel algorithm is defined as the product of its Running time and the number of processors used. It is indicative of the number of instructions executed collectively by all the processors in solving the problem in the worst case. If the cost of the parallel algorithm matches the lower bound of the best known sequential algorithm to within a constant multiplicative factor, then the algorithm is said to be cost-optimal.

Efficiency of a parallel algorithm is defined as the ratio of the worst case running time of the fastest sequential algorithm and the cost of the parallel algorithm. Usually, efficiency is less than or equal to one.

6.2 Existing parallel implementations of Random Forest classifiers

1. **PARF** – It is a task parallel implementation of Random Forest classifier in Fortran 90. PARF [81] was developed in the “Centre for Informatics and Computing” and “Division of Electronics” of “Rudjer Boskovic Institute”, with

the financial support of Ministry of Science, Technology and Sports of Croatia, i-Project 2004-111. In PARF, only the training / learning phase of Random Forest is parallelized. This implementation is cluster based and uses MPI (Message Passing Interface) library.

2. **FastRF** – It is a task parallel implementation of Random Forest classifier in Weka [82]. Fast Random Forest is a re-implementation of the Random Forest classifier (RF) for the Weka environment that improves speed and memory usage as compared to the original Weka RF. This implementation is based on multithreading concept. The average speedup factor achieved for FastRF is 2.3.
3. **CudaRF** – It is task parallel implementation of Random Forest classifier using Nvidia GPU and CUDA (Compute Unified Device Architecture) framework [30]. In CudaRF, both learning and classification phases of Random Forest are parallelized. With this, one CUDA thread is used to build one tree in the forest. The speedup achieved for CudaRF is in the range 7.7 – 9.2 is achieved for 128 trees in the forest.
4. **PLANET** – PLANET stands for Parallel Learner for Assembling Numerous Ensemble Trees. It is a Hadoop [86] based Map-Reduce implementation of tree ensembles.
5. **SPRINT** – SPRINT stands for Simple Parallel R Interface [83]. It is written in C with MPI for parallelization. It is a task parallel implementation of Random Forest classifier for use in microarray analysis.
6. **Randomjungle** – Randomjungle is a task parallel implementation of Random Forest in C++. It is used for microarray data. It is used to analyze big Genome Wide Association (GWA) data [85].

6.3 Parallel Algorithm for Random Forest Classifier

Random Forest classifier can learn in parallel as well as classification phase can also be parallelized. Based on our literature survey, we found that there is need of presenting a generalized parallel algorithm for Random Forest classifier.

In Random Forest classifier, both training and testing phases can be parallelized. In Section 6.3.1, we present an algorithm for parallel training / learning of Random Forest classifier. In Section 6.3.2, we present an algorithm for parallel testing of Random Forest classifier. We propose a data structure to be used with parallel testing algorithm, which is explained in Section 6.3.3.

6.3.1 Parallel Training / Learning Algorithm

The algorithm for parallel training / learning is presented below:

Algorithm 6.1 Parallel_Tr_RF

```
Input:
        p: Number of nodes / Number of cores available in parallel setup
        n: Total number of trees to be generated in Random  Forest
        $D_{tr}$: Training dataset with size tr
        M: Total number of attributes / features of dataset $D_{tr}$
Output:
        A Random Forest R

43. if (n>p) then
44. k = n/p                                  // k trees to be generated by each node
45. else  p = 1                              // n <= p
46. for i = 1 to k                           //Perform at each node of parallel setup
47. generate bootstrap dataset $d_i$ of size tr by performing
48. random sampling with replacement  from $D_{tr}$
49. $OOB_i = D_{tr} - d_i$
50. $m = \sqrt{M}$
51. select m  attributes from M randomly to form attribute set   $A_i = \{a_1, a_2, \ldots, a_m\}$
52. $T_i$ = Build_Decision_Tree ($d_i$ , $A_i$ )
53. end for
54. end Algorithm
```

6.3.2 Parallel Testing Algorithm

The algorithm for parallel testing is presented below:

Algorithm 6.2 Parallel_Ts_RF

```
Input:
        p: Number of nodes / Number of cores used to train RF
        n: Total number of trees to be generated in Random  Forest
        k:  Number of trees generated at each node (decided in Parallel_Tr_RF)
        $D_{ts}$: Testing dataset with size s
        C: Total number of classes of dataset $D_{ts}$
Output:
        A Random Forest R along with accuracy estimate

    1.  perform at each node of parallel setup
    2.   for j = 1 to s
    3.  for i = 1 to k
    4.  traverse tree $T_i$ for each record j
    5.  record classification for j as one of the {$c_1$, $c_2$, ..., $c_c$} in local copy of data
        structure                      //data structure is explained in next section 6.3.3
    6.  end for
    7.  end for
    8.  Update master copy of data structure for combining results of individual nodes
    9.  Calculate Accuracy by comparing results with original dataset
    10. end Algorithm
```

6.3.3 Data Structure for Parallel Testing Algorithm

We propose a data structure for combining the results of testing phase in the parallel Random Forest algorithm. Figure 6.1 presents this data structure. It is a 2 dimensional array, with one dimension for the number of classes in the dataset and the other dimension for the number of instances in the dataset. At each compute node of the parallel setup, there is a local copy of this array. There is one master copy at the master node. At the time of testing, every instance from the test dataset is run against *k* trees at each compute node. Recollect from the algorithm that total *n* trees are divided on *p* processors with each processor generating $k = n / p$ trees. If there are total c classes in the dataset varying from $c_1$to c_c, then for each instance it's classification by the k trees on one compute node is added in this data structure. That is, on compute node 1, if two trees out of the k are classifying instance t_1 as of class c_2; then in the local array the entry for t_1 and c_2 will be 2. If no tree classifies t_1 as of class c_3 then the entry for t_1 and c_3 will be 0, and so on. These local copies of data structures are combined into the master copy by performing addition of each entry $[t_i, c_j]$, $1 <= i <= s$ and $1 <= j <= c$.

The accuracy of the classifier is predicted by using the updated master copy of the data structure. For an instance t_i, the i^{th} row of the structure is traversed to find the maximum number. If j is the column to which this number belongs, then classification of the instance t_i is class c_j, $1 <= j <= c$

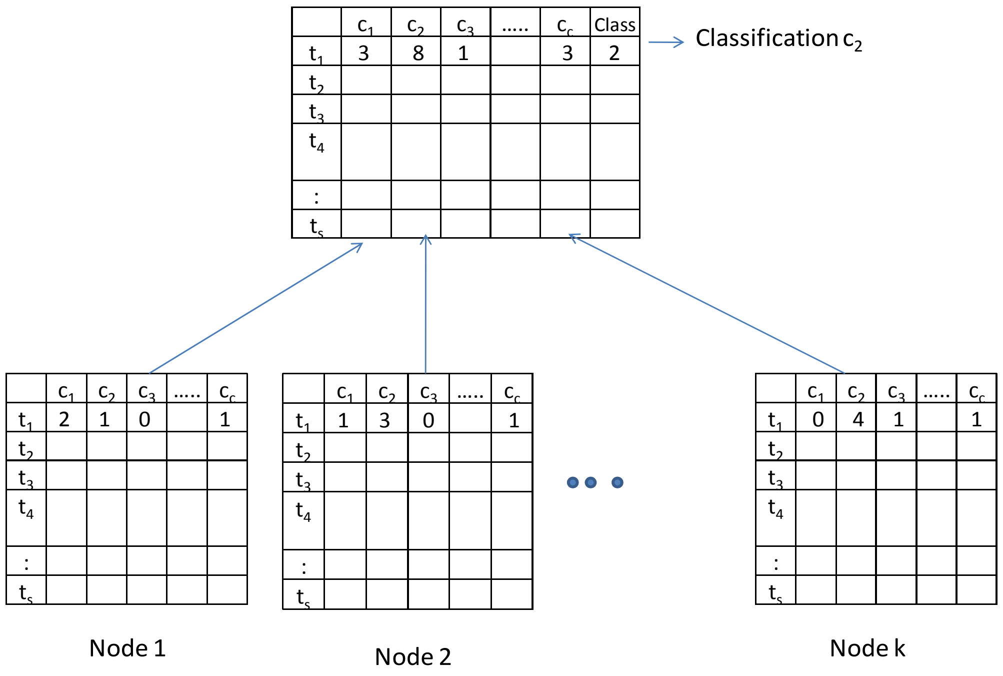


Figure 6.1: Proposed data structure for Parallel Random Forest classifier

6.4 Time Complexity Analysis of Random Forest classifier

Time Complexity of Random Forest classifier is given as,

$$= \underset{\text{(part I)}}{\textit{Time for Training}} \quad + \quad \underset{\text{(Part II)}}{\textit{Time for Testing}}$$

$$= O(n.D) + O\,(t.m.n \log n) \quad + \quad O\,(t.t_s.\log n)$$

<u>Part I: Time for Training:</u>

Let $t \rightarrow$ *Number of trees in Random Forest classifier*

$n \rightarrow$ *Size of training dataset*

$M \rightarrow$ *Total number of attributes of dataset*

$m \rightarrow$ *Attributes selected at each node (i.e. $m= \sqrt{M}$)*

Step 1: Time for bootstrap sample creation = $O\,(n.t)$

Step 2: Time for each (single) tree generation = $O\,(m.n \log n)$

Step 3: Time taken for Random forest of t trees = $O(n.t) + O\,(t.m.n \log n)$

<u>Part II: Time for Testing:</u>

Let $t \rightarrow$ *Number of trees in Random Forest classifier*

$t_s \rightarrow$ *Size of testing dataset*

Step 1: Time for testing t_s samples with a single tree = $O\,(t_s.\log n)$

Step 2: Time taken for Random forest of t trees = $O\,(t.t_s.\log n)$

6.5 Performance Measure of Parallel Random Forest Algorithm

SIMD (Single Instruction Multiple Data) [60] model for parallel processing, i.e. same instruction (program) is applied to different data on different processors / cores. This model is very well suited for task parallelism of Random Forest classifier.

Property of SIMD - All active processors (some may remain idle) perform the same operation (possibly on different data) during any parallel step. The common operation is referred to as parallel basic operation.

Worst case complexity w(n) of a parallel algorithm using *p(n)* processors is defined to be the maximum number of parallel basic operations performed by the algorithm over all inputs of size *n*.

Speedup of a Parallel Algorithm:

Let *W(n)* be the worst case complexity of a parallel algorithm for solving a given problem, and $W^*(n)$ be the smallest worst case complexity over all known sequential algorithms for solving the same problem, then

Speedup of the parallel algorithm is,

$$S(n) = \frac{W^*(n)}{W(n)}$$

Cost of a parallel algorithm is,

$$C(n) = p(n) \; X \; W(n)$$

Here, *p(n)* is the number of processors used to solve the problem.

For judging the quality of a parallel algorithm, its cost is compared with that of sequential algorithm ($W^*(n)$ X 1). A parallel algorithm is cost optimal if $C(n) = W^*(n)$.

With this background about the parallel algorithm, we now derive speedup and cost optimality for parallel Random Forest algorithm presented above.

<u>Outline of Sequential Random Forest algorithm:</u>

```
for i= 1 to n          // n is the number of trees
create n bootstrap samples i.e. in-bags
create n OOB sets
end for
```

```
for i = 1 to n        // generate n decision trees
select $m = \sqrt{M}$ attributes
create decision tree with in-bag$_i$ and m
end for
```

Hence, worst case sequential time is some function of n
i.e. $W^*(n) = f(n)$ -- I

Outline of Parallel Random Forest algorithm:

Let *p*- Number of processors

n - Number of trees to be generated

Hence, $k = \frac{n}{p}$ trees on each processor

Now on each processor / core/ thread, *k* trees are generated.

Hence, outline of Parallel Random Forest algorithm is,

```
for i= 1 to k        // k is the number of trees
  create k bootstrap samples i.e. in-bags
  create k OOB sets
end for
for i = 1 to k
  select $m = \sqrt{M}$ attributes
  create decision tree with in-bag$_i$ and m
end for
```

In Parallel algorithm, *k* loops are needed.

As $k = \frac{n}{p}$, if n = p then k =1

if n < p then k = 1 (max)

if n > p then k > 1

Hence, worst case time of Parallel algorithm is a function of *k.*

i.e. $W(n) = f(k)$ --II

From I and II,

Speedup = $W^*(n) / W(n)$

$= \frac{n}{k}$

Thus Speed up is always greater than 1

Cost of parallel algorithm:

$C(n) = p(n) \times W(n)$

$= p \times k$

$= n$

$= W^*(n)$

Hence, Parallel algorithm is cost optimal.

6.6 The Proposed Approach –Parallel Random Forest with Parallel Decision Tree (PDTPRF)

We proposed a new approach for parallel implementation of Random Forest classifier, in which along with generation of base trees in parallel, the induction process of individual trees is also parallelized. We named this approach as PDTPRF (Parallel Random Forest with Parallel Decision Tree). We tested our approach on Nvidia GPU using CUDA framework. Figures 6.2a and 6.2b present architecture of this approach. In this Figure, $D_1, D_2, ..., D_n$ are bootstrap samples used to generate trees $T_1, T_2, ..., T_n$ respectively. Figure 6.2a presents architecture for parallel learning where individual trees as well as forest learn in parallel. Figure 6.2b presents architecture in which testing and classification are done in parallel.

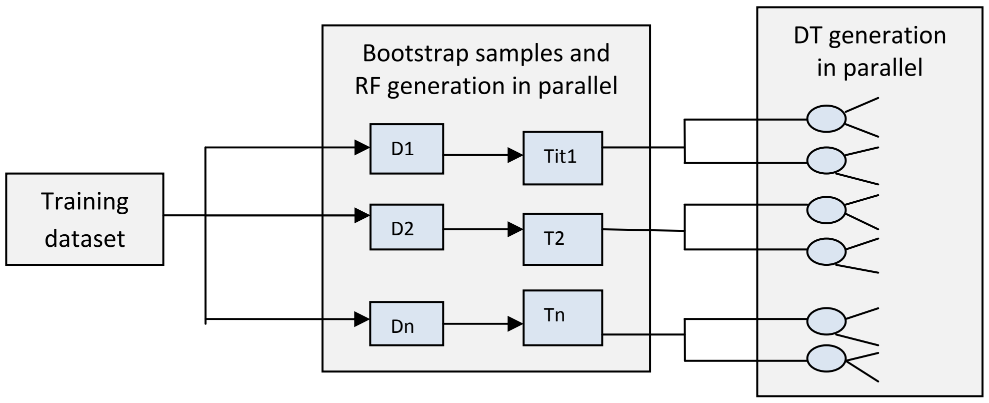

Training dataset
Bootstrap samples and RF generation in parallel
D1
D2
Dn
Tit1
T2
Tn
DT generation in parallel

Figure 6.2a: System architecture for PDTPRF approach

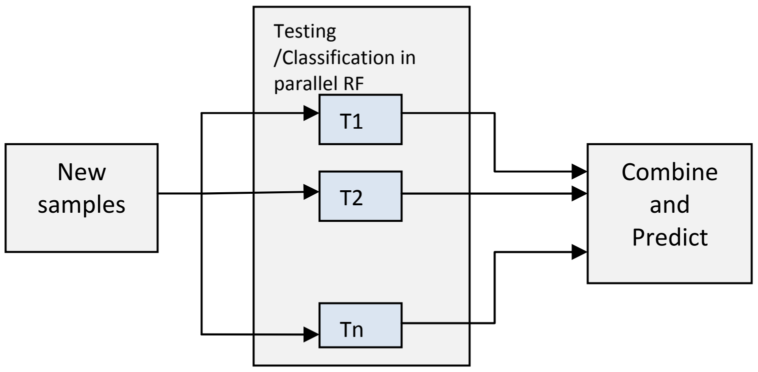


Figure 6.2b: System architecture for PDTPRF approach

The algorithm for generation of individual decision trees in parallel, in our PDTPRF approach, is as given below:

Algorithm 6.3 Build_RF_DT (Inbag[i], A)

```
// Inbag[i] is the set of instances to build tree $T_i$
// A is the set of m attributes selected randomly from original M attributes
        i.e  $A = \{a_1, a_2, ...., a_m\}$

    1.  if (size(Inbag[i] <= nodesize) then stop
    2.    else
    3.    {
    4.  select bset split out of m attributes at root node
    5.  Let $A_i$ be the attribute
    6.  Let v be distinct values for $A_i$  // v branches to root node
    7.  for j = 1 to v
        // generate v threads/ select v cores / select v processors
    8.  Build_ RF_DT (Inbag[$V_j$] , A)
    9.  end for
    10.  }
    11. end Algorithm
```

We tested the new approach on Nvidia GPU using CUDA framework. The hardware specifications are: NVIDIA GeForce GT 525M graphics card with 96 CUDA cores, 1.2GHz speed, 1GB memory and memory bandwidth of maximum 28.8 GB/sec.

The implementation performs the following tasks:

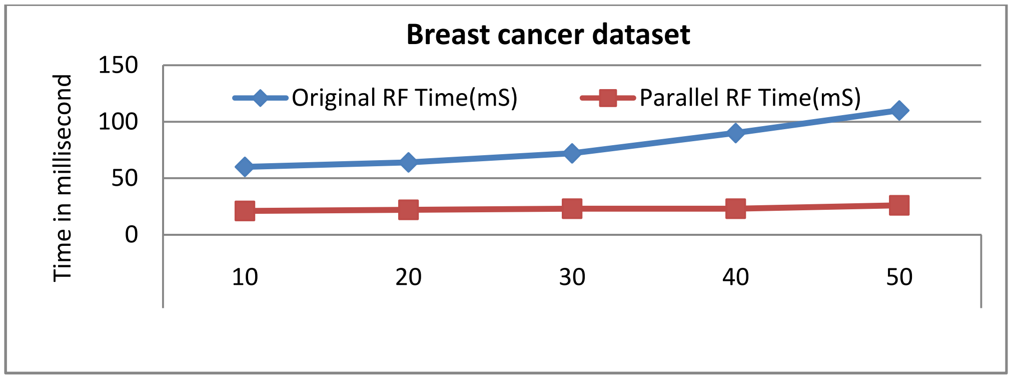
1. Open the dataset file and loads it in main memory. It is a standard service implemented by the operating system. Thus, optimizing the time for this process is not in the scope of the program.
2. Transfer the data from host memory (System RAM) to GPU memory. This process transfers all instances of the dataset to GPU memory. The bandwidth of the system bus and the latency are the only constraints.
3. A CUDA kernel with one thread per sample is launched, where each thread performs random sampling of instances, yielding bootstrapped samples used for training. Thus, data partitioning is done in parallel.

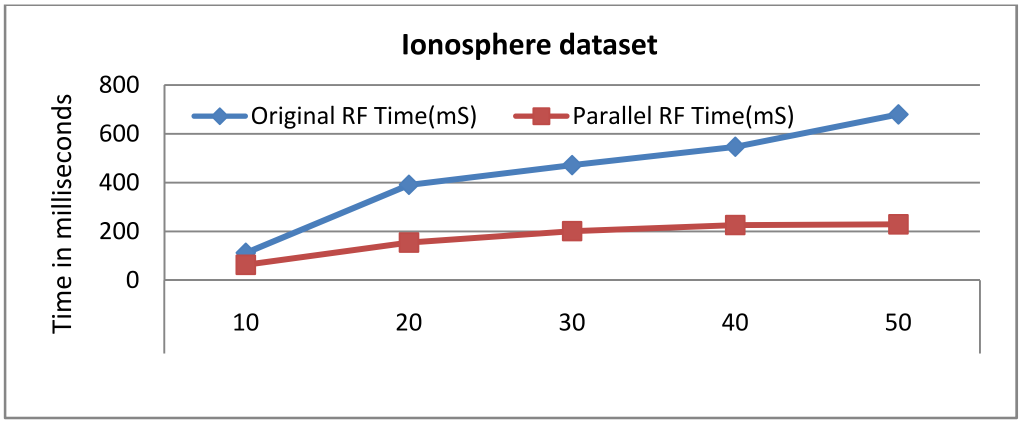
4. A synchronized CUDA kernel thread is launched per tree, which appears in active state as soon as the sampling is over. Each thread builds one tree.
5. Within the same thread, each tree performs a classification on its OOB instances in parallel, and then returns it to the host for calculating the OOB error rate.
6. At the time of prediction, CUDA kernel with a thread per tree is launched to perform voting. The kernel then calculates the majority voting and does the prediction.

We took the readings for each dataset by varying the number of trees from 10 to 50. We recorded the time required to build the forest. Table 6.1 presents the speedup achieved with our approach. Figure 6.3a and 6.3b show the graphs of speedup achieved for various datasets under experimentation.

Table 6.1: Speedup achieved with PDTPRF approach

Datasets	Number of trees				
	10	20	30	40	50
Breast-cancer	2.8571	2.9091	3.1304	3.913	4.2308
Diebetes	1.7742	2.549	2.398	2.4424	3.0942
Hepatitis	2.8571	3.5714	3.4211	3.913	4.2308
Musk1	1.7582	1.8421	2.1788	2.1918	2.2364
Sonar	1.6757	2.4286	2.5294	2.6962	2.7079
Vote	1.7419	2.1053	2.8085	3.3208	3.5082
Car	2.7692	2.8889	3.25	3.8621	4.4
Ionosphere	1.77778	2.53896	2.34826	2.42035	2.96943
Soybean	2.10989	2.15504	2.44693	2.63927	2.73802
Vehicle	2.5301	2.7486	2.8732	3.1429	3.6413
Vowel	1.7582	1.8421	2.1788	2.1918	2.2364
Waveform-5000	1.4396	1.4802	1.5122	1.5468	1.7013





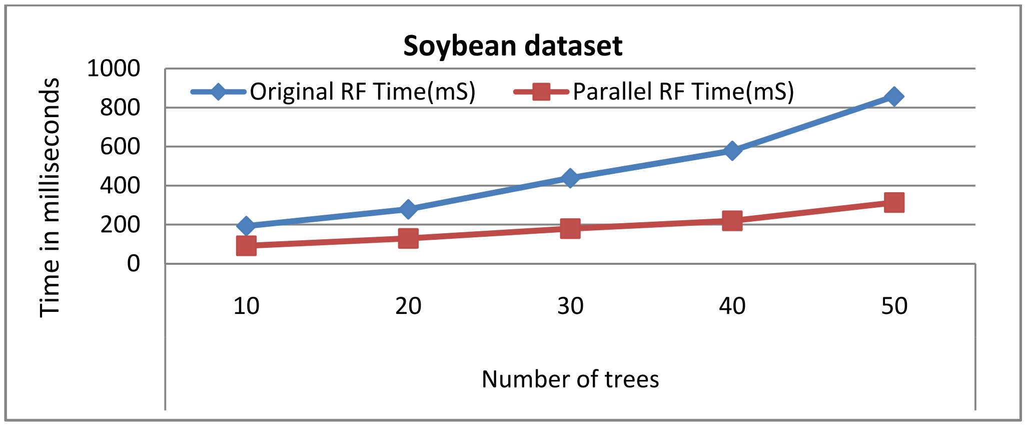
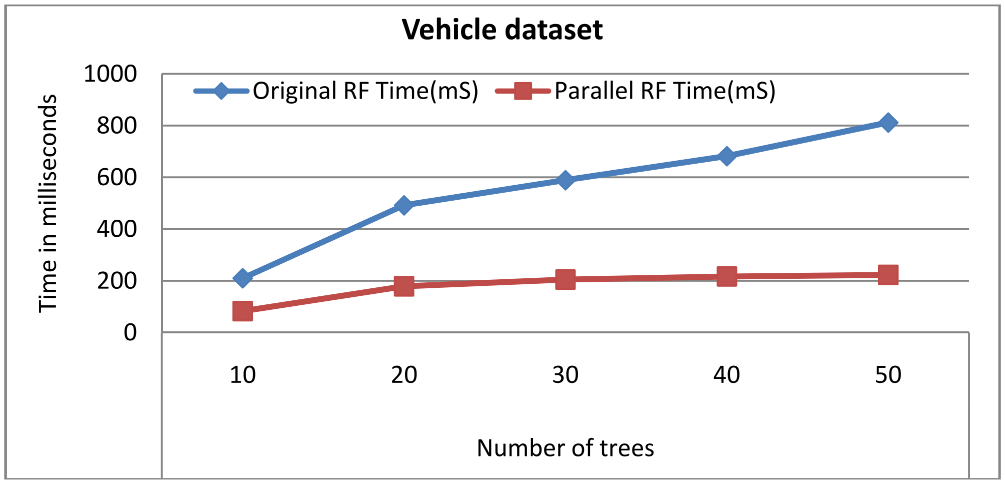
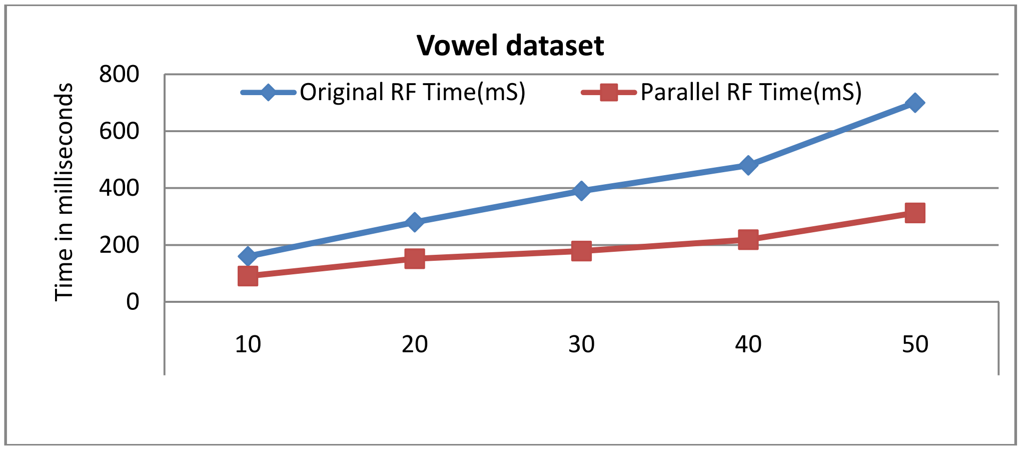


Figure 6.3a: Graphs showing speedup achieved using PDTPRF approach





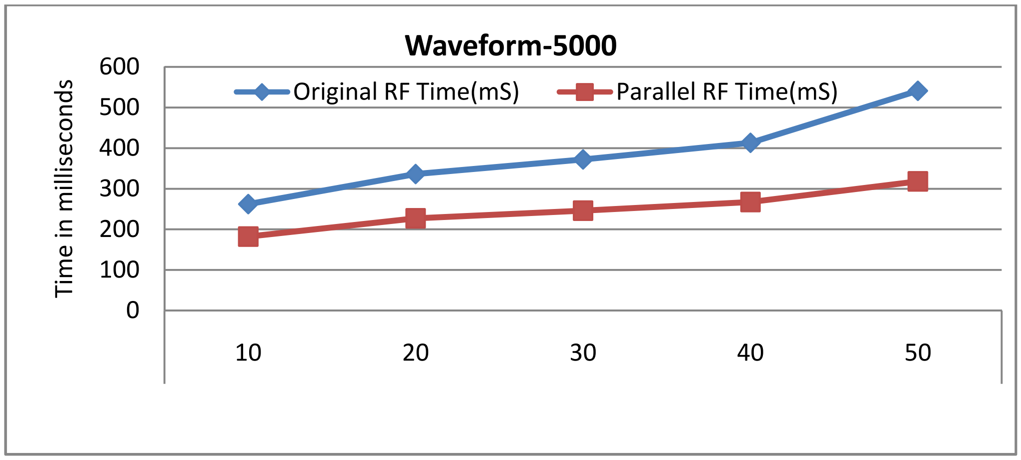


Figure 6.3b: Graphs showing speedup achieved using PDTPRF approach

6.7 Summary on Parallel Algorithm for Random Forest Classifier

We have studied existing parallel implementations of Random Forest classifier. We found that these implementations are specific to the underlying platforms and there is a need for presenting a generalized parallel algorithm for Random Forest classifier. Hence, we presented and analyzed the same, and showed that it achieves speedup, and is cost effective. We also presented a new approach named PDTPRF for parallel implementation of Random Forest classifier. In this approach, along with generation of base trees in parallel, the induction process of individual trees is also parallelized. We implemented our approach on Nvidia GPU using CUDA framework and presented results for the same. For Random Forest classifier of size 50, the speedup achieved is in the range of 1.7 to 4.4 .