Random Forest Classification in HPC
Fall 2021

Edwin Menjivar*
SFSU

ABSTRACT

The objective of this report is to explore the Random Forest Classification algorithm, and look at different ways to increase the training time using High Performance Computing, without jeopardizing the accuracy of the algorithm. This study uses different parallel programming techniques including pragmas omp, copy optimization, and compiler optimization, tested for different concurrency levels. Preliminary findings show that higher concurrency levels, higher compiler optimization level, and copy optimization increase the performance of the algorithm. A combination of all maximizes the performance.

1 INTRODUCTION

Machine Learning is evolving, and there are lots of techniques being used. Random Forest is a ML technique that uses Decision Trees, where multiple decision trees are generated during training, and each decision tree will compute their own results. The majority of the decision trees' results is used for the final answer. This algorithm as most ML algorithms are highly dependant on large data-sets, which leads to a training time of the algorithm taking long times, sometimes even weeks or months. The problem being studied in this project is ways to increase the Random Forest Classification algorithm's training performance, by using parallel programming, copy optimization or compiler optimization. The study will focus on combinations of the above. This problem has been solved before on a high level, but we will focus on the smaller part of the code.

The approach I will take for developing this project is coding the random forest classification algorithm from scratch in my program and use this as reference of a serial version. I will implement another program that uses scikit-learn's version of the algorithm, and I will also use this as reference. I will code another version of my own reference program, and I will apply different high performance computing features to my program, including synchronization features and copy optimization. Some synchronization features that I will include in my code are pragmas and its different features to parallelize and serialize different parts of my code as needed.

Some conclusions that we came up with is that increasing the concurrency level of the programs can increase performance, but depending on where the parallelization is being used, it can also decrease performance. Copy optimization increases performance, but not as much, and compiler optimization bring a lot of extra performance. Some combinations of the above can also increase or decrease accuracy.

2 RELATED WORK

Other related works exist, but our method focuses on specific improvements within the code. The other implementations use more sophisticated solutions, including external libraries and higher level HPC techniques. A Parallel Random Forest Classifier for R [5]:

The purpose of this implementation is to develop a parallel version of the random forest algorithm. The report explains how the team looked at the different aspects of parallelizing the code, and they came up a ways in which they could parallelize it. The way that they define is by splitting over the tree samples, parallelize over the samples, and combine the results. They did not implement this manually, but they decided to use an external library called "SPRINT" instead. Being ran on two supercomputers, "Cray XT4" and "Cray XT6" from the UK's national supercomputing service "HECToR," their results were amazingly good. They were able to generate a speedup of about 40.

A parallel random forest training framework based on supercomputer [6]:

The purpose of this implementation is also to develop a parallel version of the random forest algorithm. The report explains how the team looked at the different aspects of parallelizing the code, and they came up two ways in which they could parallelize it. The first way was to parallelize when splitting the tree, and the other is to parallelize over the nodes in each tree. They chose the first implementation, and with the help of MPI, they were able to receive messages between master and working nodes. Being ran on Tianhe-2 in National Supercomputer Center in Guangzhou, small datasets show no improvement, and large datasets show speedup of up to 14 times.

3 IMPLEMENTATION

To implement this study, I used multiple versions of the Random Forest Classification algotihm. The first version of the algorithm is a python based version, which uses scikit-learn's RandomForestClassifier library. The second version of the algorithm is a C++ serial version of the algorithm, which was coded using [2] as a reference. The third and fourth versions are a parallel versions developed with the serial version as a starting point. In the following subsections, we will describe more in depth the implementations for all of these versions.

3.1 Overall Code Harness

The main purpose of this implementations is to improve the performance of the Random Forest Classification algorithm, by decreasing the training runtime, without jeopardizing accuracy. And this has as the main focus to use the High Performance Computing(HPC) techniques learned during the first few weeks of the course. Such HPC techniques should be implemented in ways to demonstrate how they contribute in the increase or decrease of performance of the program.

3.2 Python Scikit-Learn Implementation

The main purpose of this implementation is to compute the Random Forest Classifier technique using a well maintained library that also has parallel capabilities. The documentation for this library is located on [3]. A snipped of the implementation is provided on Listing 1. The way this implementation works is that it uses the sk-learn library with a combination of tunable Random Forest Classification parameter, like the number of estimators, which is the number of tress, the number of jobs that will be ran in parallel, and the max depth of the three.

*email:emen15@mail.sfsu.edu

```
// n_estimators is the number of trees
// max_depth is the max depth of each tree
// n_jobs is the number of parallel jobs to be used
    model = RandomForestClassifier(n_estimators = nestimators, max_depth = depth, n_jobs = njobs)
    #Starting timer
    start_time = time.process_time()
    model.fit(X_train, y_train)
    #Ending timer
    elapsed_time = time.process_time() - start_time
```

Listing 1: Scikit-learn's RandomForestClassifier model is created, and is trained on some of the input data.

3.3 C++ Serial Implementation

Similar to the python scikit-learn's implementaion, the purpose of this implementation is also to perform the Random Forest Classification Technique. Pseudocode/code snippet of this implementation is provided on Listing 2. The purpose of this implementation is to be able to handle the same number of estimators and also the same depth as the scikit-learn implementation, in order to use both serial implementations as reference.

3.4 C++ Parallel Implementation 1 and 2 using pragmas in OpenMP

The main purpose of this implementations is to compute the Random Forest Classification algorithm without relying on one level of concurrency. This implementations use Pragmas OpenMP to parallelize different parts of the code. This implementations are built upon the previous C++ serial implementation, and the parallel techniques are added to these. Since this similar to the previous implementation, except for the parallel parts, we will only show the parts of the code that have been parallelized. Documentation on Pragmas OpenMP can be found on [4]. Implementation snippets of parallel implementation 1 can be found on Listing 3, and implementation snippets of parallel implementation 2 can be found on Listing 4.

```
for(All number of trees)
    train(Sample*sample)
    {
      int*_featureIndex=new int[_trainFeatureNumPerNode];
        Sample*nodeSample=new Sample(sample,0,sample->getSelectedSampleNum()-1);
      _cartreeArray[0]=new ClasNode();
      _cartreeArray[0]->_samples=nodeSample;
      _cartreeArray[0]->calculateParams();
      for(All nodes in each tree)
      {
        int parentId=(i-1)/2;
        if(_cartreeArray[parentId]==NULL) continue
        if(i>0&&_cartreeArray[parentId]->isLeaf()) continue;}
        if(i*2+1>=_nodeNum)
          _cartreeArray[i]->createLeaf(); continue
        if(_cartreeArray[i]->_samples->getSelectedSampleNum()<=_minLeafSample)
          _cartreeArray[i]->createLeaf(); continue
            _cartreeArray[i]->_samples->randomSelectFeature
            (_featureIndex,sample->getFeatureNum(),_trainFeatureNumPerNode);
        _cartreeArray[i]->calculateInfoGain(_cartreeArray,i,_minInfoGain);
        _cartreeArray[i]->_samples->releaseSampleIndex();
      }
      delete[] _featureIndex;
      _featureIndex=NULL;
        delete nodeSample;
    }
```

Listing 2: C++ serial RandomForestClassifier model is trained on some of the input data. Psudocode/code snipped shows iteration over every node in all trees.

4 Evaluation

We will go deeper into this study by presenting the computation platform and software environment used, the methodology, a study of both serial versions of the code, a study of the parallel versions of the code, and a study of the differrent compiler optimizations.

4.1 Computational platform and Software Environment

The computational environment for used for testing and development is: Innotek GmbH VirtualBox 1.2 with Processor: 11th Gen Inter® Core™ i7-11700KF, Clock rate: 3.60 GHz, 16GB DDR4 2666MHz memory, two 8 GB DIMMs (26.8 GiBs), VM uses half of resources, L1d cache: 48 KiB, L1i cache: 32 KiB, L2 cache: 512 KiB, L3 cache: 16 MiB and the compiler used is: g++ (Ubuntu 9.3.0-17ubuntu1-20.04) 9.3.0 with optimization levels 0 and 3, OpenMP-201511.

Data was also tested on Cori's KNL nodes NERSC [1]: Supercomputer: Cray Inc., Cascade XC40, Processor: Intel Xeon Phi Processor 7250 ("Knights Landing"), Clock rate: 1.40GHz, Each node has 96 GB DDR4 2400 MHz memory, six 16 GB DIMMs (102 GiB/s peak bandwidth), L1d cache: 32k, L1i cache: 32k, L2 cache: 1024K. The compiler used is: g++ (SUSE Linux) 7.5.0 with optimization levels 0 and 3, OpenMP-201511.

For the final results we first intended to run our data on Cori's KNL Nodes, but the results even after optimization were worse than the VM, so we decided to briefly present some of the KNL node results, and focus on the results from the local VM.

For the python version of the code we used python 3.10.0, and also the folowing packages: matplotlib:3.5.0, numpy:1.21.4, pandas:1.3.4, scikit-learn:1.0.1, scipy:1.7.3, threadpoolctl:3.0.0.

4.2 Methodology

In this study we are measuring the runtime of training part of the algorithms. In the C++ implementations we use the timer (chrono timer) provided by professor Bethel. We made some modifications to the timer to output a fixed precision. For the python implementation we calculated the runtime of the training part of the code using "time.process_time()" to measure the time before the training, and also the time right after the training, and the difference between these two is the runtime. In the parallel implementation, all the threads add their time together, but the program actually takes less time, and after research the only thing that we found out was that we have to take an average of each total time, depending on the number of threads, and this is how the runtime is calculated. In the C++ implementations, the accuracy of the predictions will be gathered by using the test sample from the data-set, and getting predictions by running it through or trained model. After the test set predictions are generated, we will test the results with the actual results that is also included in the set. We will get the percentage difference between the two and the convert this to the accuracy by subtracting the difference percentage from 100. In the python implementation, we will use the scikit-learn metrics accuracy_score to come up with this value. For

```
omp_set_num_threads(64);
#pragma omp parallel for default(shared)
  schedule(static)
for(All number of trees)
{
  //Same code as Listing2
}
```

Listing 3: C++ serial RandomForestClassifier model is trained on some of the input data. Psudocode/code snipped shows iteration over all trees.

```
for(All number of trees)
      omp_set_num_threads(50);
      #pragma omp parallel for default(shared)
  schedule(static)
      train(Sample*sample)
      {
        //Same code as Listing 2
      }
```

Listing 4: C++ parallel RandomForestClassifier model is trained on some of the input data. Psudocode/code snipped shows iteration over every node in all trees.

Implementation	Runtime(sec)	Accuracy
C++ Serial Opt 0	124.13	94
C++ Serial Opt 3	75.97	93.74
Python Serial	25.09	0.96.88

Table 1: Comparison of training runtime and accuracy of prediction for different implementations of Random Forest Classifier

each of the parallel implementations, the program is being tested with the following concurrency levels: [1,16,32,64,128,256,512], with 100 estimators, and no maximum depth. Besides measuring the runtime, the speedup will be calculated for each of these.

4.3 Serial implementation study

The first few optimizations that we found to increase performance by decreasing the runtime of the training, without lowering the accuracy of out techniques were to change the level of the compiler optimization. We decided to run the C++ serial implementation in Optimizations 0 and 3. These were the first techniques that we learned in the course, and it played out very well, as optimization level 3 is able to see a huge advantage in the runtime. This is shown on Table 1. We only implemented this in the C++ versions, because we had full control over the compiler. Copy optimization is used within the serial implementation in the form of memcopy.

Other optimization ways that we found to decrease the runtime of the algorithm was to play around with different combinations of tunable Random Forest algorithmic parameters. One is able to decrease the runtime in a huge way just by decreasing some of this parameters, like the number of trees, but what we are really doing is decreasing the performance of out model. As shown on Fig. 1, and Fig. 2, when we decrease the number of estimators, we are also decreasing the accuracy of the predictions. The accuracy figure shows that all serial versions of the algorithm follow the same patter, with low accuracy with low estimators and increases for higher number of estimators. Limiting the depth of the tree gave us similar results, so we have decided to not show it. There are other tunable parameters that we could change, like the minimum number of samples to split an internal node, or the minimum number of samples required to be at a leaf node, but we will not focus on this, and instead we will focus in the parallel versions of our

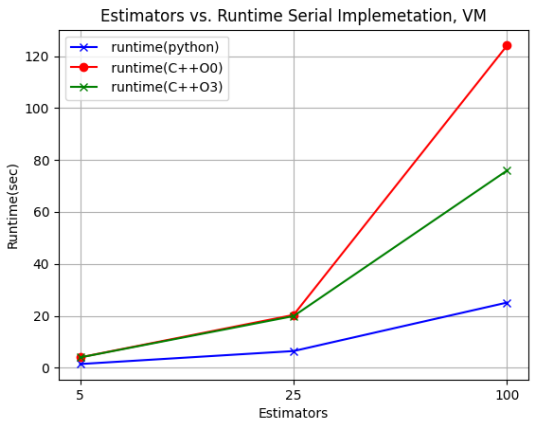

Figure 1: Number of estimators vs. runtime comparison for predictions of our three implementations. Python scikit-learn, C++ with Optimization level 0 and C++ with Optimization level 3

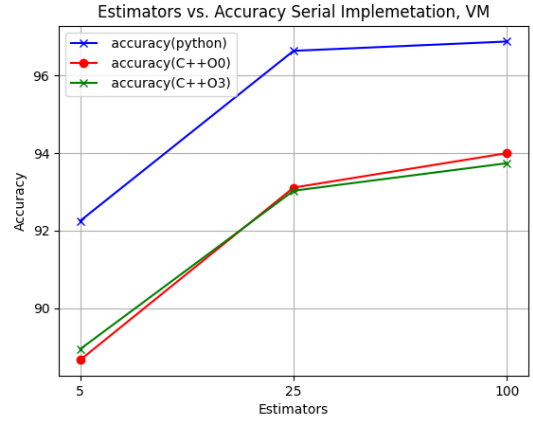

Figure 2: Number of estimators vs. accuracy percentage comparison for predictions of our three implementations. Python scikit-learn, C++ with Optimization level 0 and C++ with Optimization level 3

implementations in the following sections. Table 1, Fig. 1, and Fig. 2 also show us that compiler optimization level 3 outperforms compiler optimization level 0, and it does not affect accuracy.

4.4 Parallel implementation study

In this part of the study, the first thing that we decided to do was to run the Python's scikit-learn version of the Random Forest Classification using multiple jobs. This is one way to parallelize this implementation. The following images show the runtime on Fig. 3 and the accuracy on Fig. 4 of the parallelize python implementations. The runtime seems to decrease as the concurrency increases, and even though the accuracy figure augments the graph, if we pay close attention, we can appreciate that the accuracy stays within a very small range, and doesn't change much.

The python parallel implementations will be used as reference to the code to the other parallel implementations.

To implement the C++ parallel versions, we started out with the serial C++ version and improved the code by adding the pragmas omp to different parts of the code. In the first implementation we parallelize over the threes, and in the second implementation, we parallelize over the number of nodes. The results from these

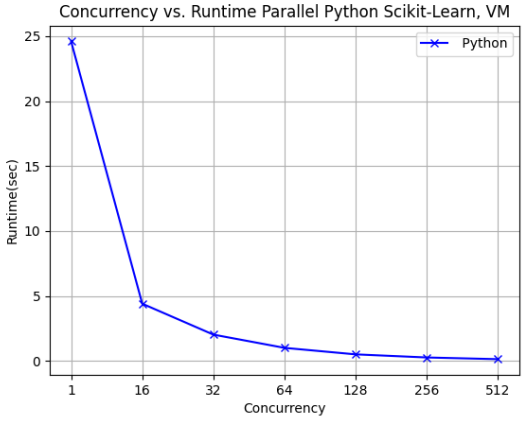

Figure 3: Concurrency level vs. runtime for Python scikit-learn parallel implementation.

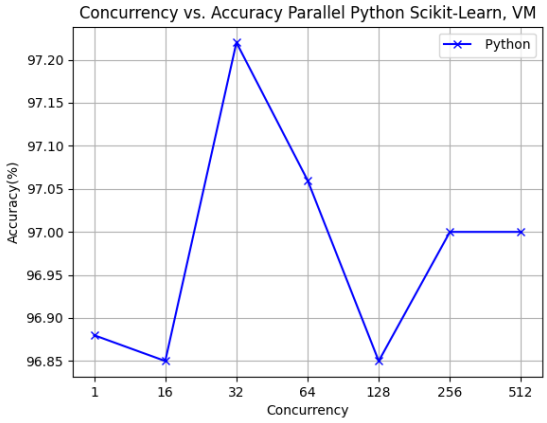

Figure 4: Concurrency level vs. accuracy percentage for Python scikit-learn parallel implementation.

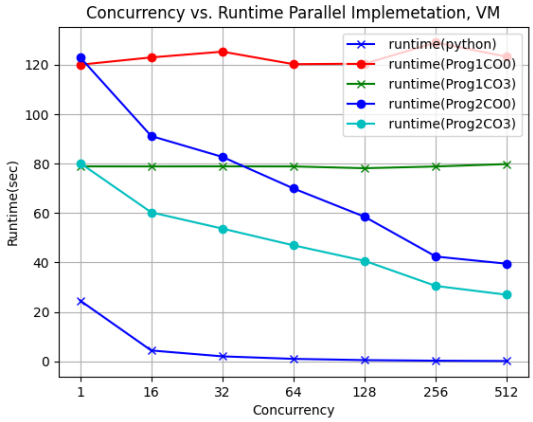

Figure 5: Concurrency level vs. runtime comparison for predictions the following implementations: Python scikit-learn, Parallel C++ Program 1 with Optimization level 0 and Parallel C++ Program 1 with Optimization level 3, Parallel C++ Program 2 with Optimization level 0 and Parallel C++ Program 2 with Optimization level 3

implementations are in the following images. The runtime is on Fig. 5 and the accuracy on Fig. 6.

From the results, we can see that our implementation of the parallelization over the trees, does not improve the training time for any of the two compiler optimizations. In fact, we can see that the runtime of these implementations stays steady. We do not see the decrease in runtime that we saw in the reference python implementation. The second implementation involves paralleling over the nodes of the tree. In this implementation, we are able to see a decrease in time as the level of concurrency increases. Again, we do not see the same results as the reference implementation. In fact, the reference implementation seems to decrease its time by a factor of 2, while the C++ implementation seems to be converging, as if it is reaching its limit. This is similar to the C++ implementations that we have seen earlier in the course, where CBLAS outperformed all of our codes. Here we can see that the optimizations of scikit learns outperform all the parallel versions of our implementation, for both compiler implementations.For the accuracy figure, we have only included the python implementation and the c++ parallel implementations for program 1, program 2, had some bugs, and it would generate segmentation errors. From the displayed accuracies, we can appreciate that as the concurrency level changes, the accuracy percentage does not change much, therefore it serves the purpose of this assignment which is to decrease rumtime without decreasing accuracy. The speedup that we see from all of our implementations is shown on Fig. 7. The speedup that we see from the C++ implementations is shown on Fig. 8

In the speedup Fig. 7 that shows all the implementations, it is very easy to see how good of a speedup the scikit-learn implementation has. From this image, it is very hard to tell the speedup of the C++ parallel implementations, because their speedup is so much lower, this is why we included Fig. 8 where it is easier to appreciate the speedup of these implementations. From this figure, we can see that program 1's speedup is of about 3 times when ran on compiler optimizations 0 and 3, while program 2 did not see a speedup at all. In program 1's implementation, we can see a constant speed up for concurrencies 16 to 128, but then we see a bigger speedup from 128 to 256, and them the speedup decreases a little bit from 256 to 512. A speedup of up to 180 is seen in the scikit learn implementation, which basically doubles as the concurrency level doubles. Fig. 5, Fig. 6 and Fig. 7 and also show us that compiler optimization level

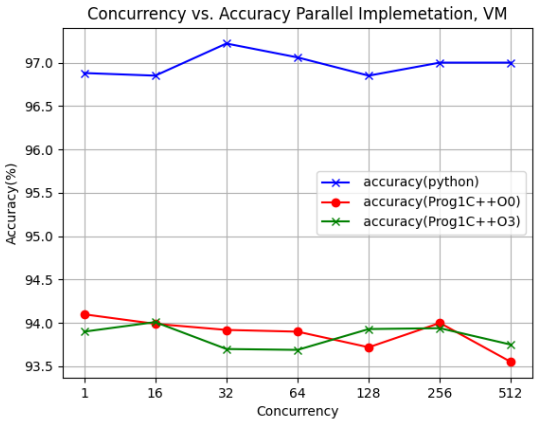

Figure 6: Concurrency level vs. accuracy percentage comparison for predictions the following implementations: Python scikit-learn, Parallel C++ Program 1 with Optimization level 0 and Parallel C++ Program 1 with Optimization level 3

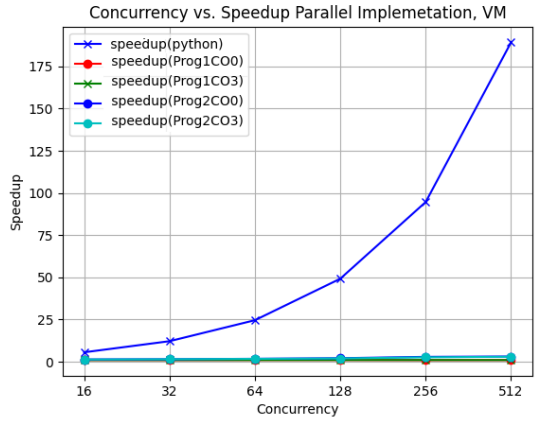

Figure 7: Concurrency level vs. speedup comparison for predictions the following implementations: Python scikit-learn, Parallel C++ Program 1 with Optimization level 0 and Parallel C++ Program 1 with Optimization level 3, Parallel C++ Program 2 with Optimization level 0 and Parallel C++ Program 2 with Optimization level 3

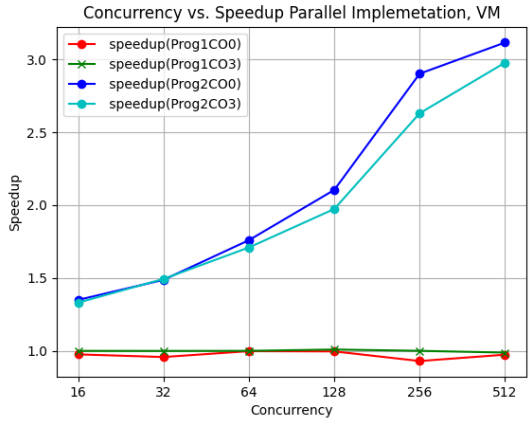

Figure 8: Concurrency level vs. speedup comparison for predictions the following implementations: Parallel C++ Program 1 with Optimization level 0 and Parallel C++ Program 1 with Optimization level 3, Parallel C++ Program 2 with Optimization level 0 and Parallel C++ Program 2 with Optimization level 3

3 outperforms compiler optimization level 0, and it does not affect accuracy.

5 Discussions and Findings

From our evaluation, we found that concurrency increases performance of the training part of the algorithm. We also found that the compiler optimization level 3 produces better results than compiler optimization level 0, and it does not jeopardize the accuracy of our model. When ran in parallel, we saw that parallelizing over the nodes as in the C++ parallel program 2, produces better results than parallelizing over the trees as in the C++ parallel program 1. The problem of cache utilization produces better results as we have seen previously, because it makes our compiler faster, but because we hard-coded this in our implementation since the beginning, we were not able to test this again. As we saw, the ration of time to train the model for serial and parallel versions varies a lot depending on the implementation, but over all, parallel versions can produce results that are many times faster that its serial version. Speedup relates to efficiency in the terms that as speedup increases, our implementations are more efficient, because the training time is lowered, and the accuracy stays about the same, so the model is more efficient.

6 Conclusions and Future Work

To conclude, the problem that we initially agreed to solve was to lower the training time of the Random Forest Classification algorithm without decreasing it's accuracy. This was solved in many ways. We have implemented parallel versions of the Random Forest Classifier that are faster than that of the serial implementations, without jeopardizing the accuracy of the model. First, we improved the classification model of scikit-learn by tuning some of it algorithmic parameters. We modified the C++ serial program to parallelize certain parts of the code. Doing so, allowed us to have to parallel C++ solutions for the Random Forest Classification algorithm, one that parallelizes over the trees which is our second solutions, and the other that parallelizes over the nodes of the trees, which is our third. Our solution is worthwhile, because it shows that people can solve problems, without the need for extreme resource, like the ones used in the previous works mentioned above. Our solution also shows that there are always small improvements that you can do to make your code perform better. Some future works that we could do to our implementation is to use more complex High Performance Comput-

ing techniques like OpenMP with device offload, or maybe create an implementation of the Random Forest Classification Algorithm that runs solely on the GPU.

REFERENCES

[1] https://docs.nersc.gov/systems/cori/.

[2] https://github.com/handspeaker/RandomForests.

[3] https://scikit learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.htmls.

[4] https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-pragmas summary.html.

[5] L. Mitchell, T. M. Sloan, M. Mewissen, P. Ghazal, T. Forster, M. Piotrowski, and A. S. Trew. A parallel random forest classifier for r. In *Proceedings of the Second International Workshop on Emerging Computational Methods for the Life Sciences*, ECMLS '11, p. 1–6. Association for Computing Machinery, New York, NY, USA, 2011. doi: 10.1145/1996023.1996024

[6] C. Wang, T. Cai, G. Suo, Y. Lu, and E. Zhou. Distforest: A parallel random forest training framework based on supercomputer. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 196–204, 2018. doi: 10.1109/HPCC/SmartCity/DSS.2018.00057