An improved optimal partitioning algorithm based on multithreaded approach

Anuraag Srivastava, School of Informatics Computing and Cyber Systems
Norhern Arizona University, Flagstaff, USA
as4378@nau.edu

I. INTRODUCTION

There are several applications where we need to work with ordered data (e.g. Time-series). This includes financial data, climate data, radio signals, genome data in bioinformatics etc. Often it is of interest to find patterns or structure in this data and to detect abrupt changes in structure to model the data effectively. These abrupt changes are referred to as change points and results in data being partitioned into different segments.

There are several approaches to detect changepoints based on different statistical criteria like penalized likelihood. Also, there are class of dynamic programming algorithms to solve the optimization problem given using a penalized likelihood criterion usually referred as loss-function. These methods are referred as PELT (Pruned Exact Linear Time). The standard implementation of dynamic programming approaches to solve the problem is at least of quadratic complexity in number of data-points. Recent pruning ideas have been suggested to improve the efficiency of these algorithms further like FPOP and SNIP but often their implementation is hard to understand and to modify the solution based on different statistical criteria like a different cost function for resulting segments is troublesome.

In this project I propose a solution to improve the efficiency of the standard implementation (using PELT approach and square-error loss function) by using multithreaded approach in C++ which is easy to understand and modify to suit different loss-functions. However, same ideas can be extended to apply to pruning approach to further improve their efficiency.

II. BACKGROUND

There are several approaches to solve the optimal partitioning problem using dynamic programming approach. One such approach is described in [1] where authors have described an algorithm which scales quadratically in number of data points. The authors have described this algorithm using square-error loss function. In the same paper the authors have described pruning approaches which improve the time complexity to log-linear using approaches like segment-neighborhood search, constrained segment-neighborhood search and functional pruning. Similarly, in [2] the authors have described a quadratic time complexity algorithm for changepoint detection which finds the global optimum and useful for signal processing data.

For change-point detection problems there are techniques which are easily parallelizable including Binary Segmentation, due to Scott and Knott (1974), and its related approaches, notably the Wild Binary Segmentation (WBS) method of Fryzlewicz (2014). However, these methods are not based on PELT and are characterized as 'Embarrassingly Parallel'. It is not so straightforward to parallelize dynamic programming methods such as PELT. There have been very few attempts to implement a multi-threaded approach to solve the problem using PELT. Usually, people rely on pruning approaches for performance improvement, but pruning approaches are hard to understand and difficult to modify when building changepoint detection models for different loss functions. One such attempt to parallelize the traditional dynamic programming approach is described in [3]. The authors here have described an approach to split the data-set equally between cores in chunks of contiguous segments. Each core then computes the local change-points and returns the result to parent core. The parent core after gathering results performs its own PELT and returns the final change-points. This method can give substantial computational improvements, and, in some cases, cost decreases roughly quadratically in the number of cores. However, this method is not guaranteed to find the true minimum of the penalized cost.

The approach described in this project is based on splitting the computations equally between threads instead of data-points using a shared-memory approach and is thus guaranteed to find the true minimum of penalized cost.

III. APPROACH

In the standard dynamic programming approach, the ordered data points are denoted by:
Y = (y1, y2, …, yn)

The set of observations (or segment) from point 's' to 't' are denoted as:
Ys:t = (ys, …, yt)

The cost of a segment is denoted as:
C(Ys:t)

Where **'C'** denotes the cost function which depends on statistical criteria like square error loss function. In this project I am using standard square error loss function as cost function.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

Now, suppose we want to segment the data **Y1:t.** Let, **F(t)** denote the minimum cost of segmenting such data with **F(0) = (-1) * B** where '**B'** is a constant denoting initial penalty. The idea is to is to split the minimization over segmentations into the minimization over the position of the last changepoint, and then the minimization over the earlier changepoints.
After some mathematical calculations we arrive at the following recurrence relation:

$$F(t) = \min_{0 \le k < t} [\, F(k) + C(Y_{k+1:t}) + B \,]$$

Thus if we want to compute **F(t)** we first need to compute **F(t-1), F(t-2), …, F(0)** and then evaluate the above equation and select the minimum value as our optimal value.
For instance, **F(4)** will use results **of F(3), F(2), F(1), F(0)** and **F(3)** will use results of **F(2), F(1), F(0).** We store these values in a list so that they can be utilized when needed in computation.
The idea of multithreaded approach is to divide these computations of **F(t)** equally between '**n**' threads. In ideal case we will have '**t**' threads to compute **F(t),** but due to system limitations we might not be able to have these many threads for large values of **'t**'.

In sequential programming we start by calculating **F(0)** then we move to **F(1)** and so on till **F(t).** But will multithreaded approach we can think of each of these evaluations running in parallel. Suppose if we want to compute **F(10**) and we can create 10 threads, then **thread 1** will evaluate **F(1), thread 2** will evaluate **F(2)** etc. Since, the computations are dependent on results of previous threads we need to use condition variables, locks and shared buffer. The algorithm will work as follows for each thread.

```
//shared buffer S will have length equal to no. of data points
//for thread 'm' we need to compute F(m)
//we also have an array of condition variables C of length 't'
for (int i = 0; i < m; i++) {
        acquire lock on S
        while(S[i] == 0) {
                wait on condition variable C[i];
        }
        release lock
        Evaluate F(i) + Cost (Yi+1:m) + B
}
acquire lock on S
update S[m]
Notify all threads waiting on S[m]
release lock
```

This approach will improve efficiency as all the evaluations are running in parallel and when a particular **F(m)** is computed all the subsequent threads can utilize this value in parallel. Also, for **F(t)** we will need 't' condition variables.

To understand the parallelization more clearly, we can take an example where **t** = 4. Let's see how the computation of each of **F(i) for 1 <= i <= 4** will look like with **F(0) = (-1) * B**:

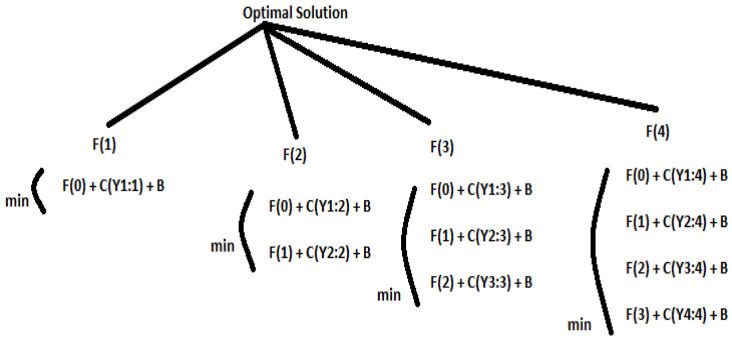

Fig 1.

For a sequential approach first **F(1)** will be computed which requires F(0). Then **F(2)** will be computed which will use values of **F(0)** and **F(1)** then **F(3)** and then **F(4)** in same fashion.

Now, if 4 threads are running in parallel for each **F(i)** then as soon as **F(0)** is available all threads from **F(1)** till **F(4)** can compute the first expression which uses **F(0)** in parallel. Similarly, when **F(1)** is done then all threads from **F(2)** till **F(4**) can evaluate the second expression which uses **F(1)** in parallel and so on.

Thus, when we were using sequential approach then for F(4) we had 1 + 2 + 3 + 4 = 10 operations. But, with multithreaded approach as described above we will have 4 effective operations as our cost function takes constant time. This number might increase when number of threads is less than '**t'** but still better than sequential approach.

For large values of '**t**' however we will not be able to create so many threads, so we will have to divide the computations of each **F(i)** equally between maximum number of threads possible (this depends on the system) using some scheduling mechanism (static or dynamic) which is one of the challenges of the project.

A. Improvements

I. No Race Conditions

If we carefully analyze the distribution of computations described in approach above (Fig 1.) we can see that the for each thread, there will be a unique value of **F(t)** it has to compute**.** For instance if there are 4 data points and 4 threads as described above then thread 1 is responsible for updating **F(1)**, thread 2 will update **F(2)** and so on. In case where there are more than 1 computations per thread (described later) we will still have unique values of **F(t)** for each thread. Thus, no two threads will compete to update a particular value of

S[m] for shared buffer **S** at index **m**. So, we can remove the locks and condition variables and update our pseudo-code as follows:

```
//shared buffer S will have length equal to no. of data points
//for thread 'm' we need to compute F(m)
    for (int i = 0; i < m; i++) {
      while(S[i] == 0) {
            //do nothing
      }
      Evaluate F(i) + Cost (Yi+1:m) + B
    }
      update S[m]
```

This will save time spent on acquiring and releasing lock and in the results section, we can see that our results are consistent with the results of FPOP on same dataset.

B. Scheduling

In the above section we have seen the case where we have small number of data points and so we can have number of threads equal to the data points. For large data sets and limited number of threads, we need to divide the computations between threads. For these there are two approaches as described below:

I. STATIC

In the static scheduling we will divide the computations sequentially between threads. To illustrate this point, let's consider a case where the number of data points are 100 and we have 10 threads. Then we will divide the computations as follows:

(i) Find the factor by dividing number of data points by number of threads
(ii) For each thread the computations will start from **(thread id -1) * factor** and end at **(thread id)*factor – 1.** These values must be passed in arguments when calling thread. I have used structures for this purpose.

Thus for 10 threads, thread 1 will get computations from (0 to 9), thread 2 from (10 to 19), thread 3 from (20 to 29) and so on. The resulting pseudo code will look as follows:

```
//shared buffer S will have length equal to no. of data /points
//for thread 'm' the computation starts from variable
//named 'start' and at 'end'
    for (int i = start; i <= end; i++){
          for (int j = 0; j < i; j++) {
                while(S[j] == 0) {
                      //do nothing
                }
          Evaluate F(j) + Cost (Yj+1:i) + B
          }
          update S[i]
    }
```

However, this approach has its disadvantages. As, we can see that for each **F(t)** the load the number of operations to compute **F(t)** are proportional to **t.** Thus, if we schedule using static scheduling then some threads which get low values of **t** will compute **F(t)** early and then will remain idle for rest of the computation. For large number of data-points the last thread will get the large values of **t** to compute **F(t)** and performance improvement will diminish. To overcome this issue and remove the load imbalance we will use dynamic scheduling as described below:

II. DYNAMIC

In dynamic scheduling, we follow an approach similar to Producer-Consumer problem where as soon as thread is finished with its current task it can be allotted remaining tasks. This makes sure that threads don't remain idle.

To implement this, we will maintain a shared variable which will always point to the index in shared buffer **'S'** which has not been assigned to any thread. This index value corresponds to the **'t'** value for **F(t).**

Each thread then checks if the current position is not greater than number of data points, it locks the shared variable pointing to current index in **S** and saves this index value in variable named **'t',** it then increments the current index by 1 and releases the lock. Then it computes **F(t)** and updates **S[t].**

The following pseudo code illustrates the idea:

```
//shared buffer S will have length equal to no. of data points
//shared variable named current points to current index in S

while(true){
      acquire lock on current
      if (current > num of data points) {
            //it means that all computations are done
            release lock on current
            break
      }
      else{
            save value of current in variable named 't'
            update the value of current by 1
            release lock on current
        }
      for (int j = 0; j < t; j++) {
            while(S[j] == 0) {
                  //do nothing
            }
      Evaluate F(j) + Cost (Yj+1:t) + B
      }
      update S[t]
}
```

In the results we can see that this approach leads to much better performance gains than static approach.

IV. RESULTS

A. *Correctness*

The following figures show that results of our C++ code without locks and condition variables (using 5 threads) are consistent with the FPOP on the same data set with penalty parameter equal to 1 when run on data set of size 10, 20 and 30 data-points respectively.

```
> Fpop(c(1:10), 1)
$`signal`
 [1]  1  2  3  4  5  6  7  8  9 10

$n
[1] 10

$lambda
[1] 1

$min
[1] 1

$max
[1] 10

$path
 [1] -10 -10   2   2   4   4   6   6   8   8

$cost
 [1]    1.0   -2.5  -10.5  -26.0  -50.0  -85.5 -133.5 -197.0 -277.0 -376.5
```

Fig 2. (FPOP on 10 data-points)

```
C:\Users\AnuraagUS\Spring2019\CS599>Parallel_St 10 5
Optimal cost values:
1 -2.5 -10.5 -26 -50 -85.5 -133.5 -197 -277 -376.5
```

Fig 3. (C++ code on 10 data-points)

```
> Fpop(c(1:20), 1)
$`signal`
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

$n
[1] 20

$lambda
[1] 1

$min
[1] 1

$max
[1] 20

$path
 [1] -10 -10   2   2   4   4   6   6   8   8  10  10  12  12  14  14  16  16  18  18

$cost
 [1]    1.0   -2.5  -10.5  -26.0  -50.0  -85.5 -133.5 -197.0 -277.0 -376.5 -496.5 -640.0
[13] -808.0 -1003.5 -1227.5 -1483.0 -1771.0 -2094.5 -2454.5 -2854.0
```

Fig 4. (FPOP on 20 data-points)

```
C:\Users\AnuraagUS\Spring2019\CS599>Parallel_St 20 5
Optimal cost values:
1 -2.5 -10.5 -26 -50 -85.5 -133.5 -197 -277 -376.5 -496.5 -640 -808 -1003.5 -1227.5 -1483
-1771 -2094.5 -2454.5 -2854
```

Fig 5. (C++ code on 20 data-points)

```
> Fpop(c(1:30), 1)
$`signal`
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

$n
[1] 30

$lambda
[1] 1

$min
[1] 1

$max
[1] 30

$path
 [1] -10 -10   2   2   4   4   6   6   8   8  10  10  12  12  14  14  16  16  18  18  20  20  22  22  24
[26]  24  26  26  28  28

$cost
 [1]    1.0   -2.5  -10.5  -26.0  -50.0  -85.5 -133.5 -197.0 -277.0 -376.5 -496.5 -640.0
[13] -808.0 -1003.5 -1227.5 -1483.0 -1771.0 -2094.5 -2454.5 -2854.0 -3294.0 -3777.5 -4305.5 -4881.0
[25] -5505.0 -6180.5 -6908.5 -7692.0 -8532.0 -9431.5
```

Fig 6. (FPOP on 30 data-points)

```
C:\Users\AnuraagUS\Spring2019\CS599>Parallel_St 30 5
Optimal cost values:
1 -2.5 -10.5 -26 -50 -85.5 -133.5 -197 -277 -376.5 -496.5 -640 -808 -1003.5 -1227.5
-1483 -1771 -2094.5 -2454.5 -2854 -3294 -3777.5 -4305.5 -4881 -5505 -6180.5 -6908.5
-7692 -8532 -9431.5
```

Fig 8. (C++ code on 30 data-points)

As we can see our results match with the cost vector returned by FPOP which gives the optimal segment costs.

The following figures shows that results of dynamic scheduling are also consistent with FPOP when run on same data sets.

```
C:\Users\AnuraagUS\Spring2019\CS599>Parallel_Dy 10 5
Optimal cost values:
1 -2.5 -10.5 -26 -50 -85.5 -133.5 -197 -277 -376.5
```

Fig 9. (Results on 10 data-points)

```
C:\Users\AnuraagUS\Spring2019\CS599>Parallel_Dy 20 5
Optimal cost values:
1 -2.5 -10.5 -26 -50 -85.5 -133.5 -197 -277 -376.5 -496.5 -640 -808 -1003.5 -1227.5
-1483 -1771 -2094.5 -2454.5 -2854
```

Fig 10. (Results on 20 data-points)

```
C:\Users\AnuraagUS\Spring2019\CS599>Parallel_Dy 30 5
Optimal cost values:
1 -2.5 -10.5 -26 -50 -85.5 -133.5 -197 -277 -376.5 -496.5 -640 -808 -1003.5 -1227.5
-1483 -1771 -2094.5 -2454.5 -2854 -3294 -3777.5 -4305.5 -4881 -5505 -6180.5 -6908.5
-7692 -8532 -9431.5
```

Fig 11. (Results on 30 data-points)

B. *Execution Times*

Following figure (Fig 12) shows how the execution time scales with number of data-points for sequential approach when run on monsoon cluster with 1 node and 1 CPU up to 120000 data points. We can see the quadratic growth of run time with data size.

Fig 12. (Sequential execution time)

Fig 13 below shows how the run-time scales when using multi-threaded approach with dynamic scheduling on 5, 10, 15, 20, 25 and 30 threads respectively when run on monsoon cluster on single node and with CPUs per task equal to number of threads, up to 120000 data points. We can also see how this run time scales in comparison to sequential approach.

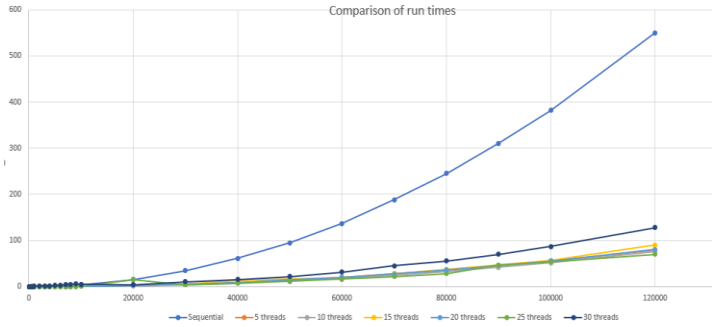

Fig 13. (Comparison of execution times of sequential approach with 5, 10, 15, 20, 25 and 30 threads)

From the above plot we can see that there is quite good improvement in execution time when implementing multithreaded approach with dynamic scheduling. However, we can see that we get almost similar results with 5, 10, 15 and 20 threads with performance slightly improved with 20 threads and for 15 threads the run time is just minutely more than others for large data-sets. With 30 threads, however the runtime is more than other threads. We should expect to see runtime decreasing with increasing threads, but, because all the threads compete to acquire a lock on shared variable a lot of time will be wasted in waiting for locks. Also, there is time taken in context switches within CPU and some cores might be busier than others. This time will increase with increasing threads. Thus, the improvement achieved by more threads is offset by the overheads discussed above.

C. Speedup

The next figure shows the speedup achieved by using different threads.

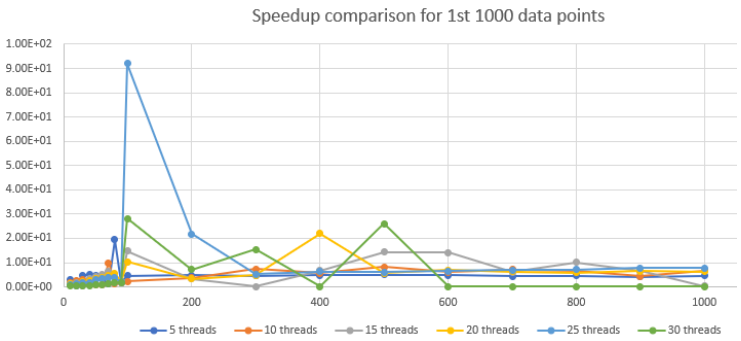

Fig 14. (Speedup for 1st 1000 datapoints)

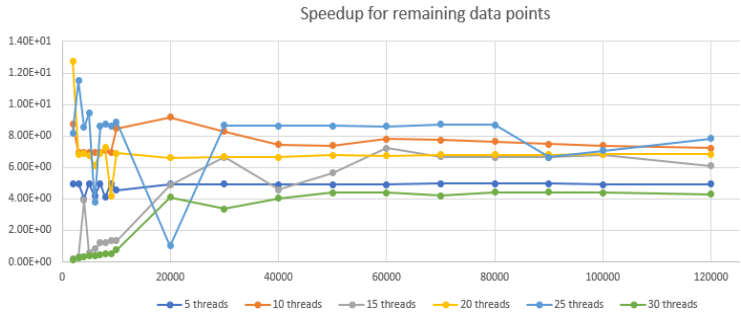

Fig 15. (Speedup for remaining datapoints)

From Fig 14 and 15 we can see that with small number of data points there are some spikes in speedup when using more threads. However, as the data increases the values saturate and we don't observe any more trends. This is due to the same reason that, when we increase threads then more time is wasted on context switches and waiting on locks which offsets the improvement in runtime.

V. DISCUSSION AND CONCLUSION

In this project we implemented a multi-threaded approach using dynamic programming to solve optimal partitioning problem which scales quadratically in number of data-points. From results we can see that the multi-threaded algorithm reduces the runtime considerably, especially for large number of data-points. However, there is not much improvement when we increase the number of threads and if we use large number of threads then sometimes the execution time increases in comparison to using lesser number of threads.

In future, we can work on methods that will improve performance of this approach when using more threads. One of the starting points could be using condition variables to signal one thread at a time with dynamic scheduling. This will make sure that multiple threads are not continuously trying to acquire lock on shared variable. It might require designing the algorithm differently.

VI. REFERENCES

[1] On optimal multiple changepoint algorithms for large data, Toby Hocking, Guillem Rigaill, Paul Fearnhead

[2] An Algorithm for Optimal Partitioning of Data on an Interva, Brad Jackson, Jeffrey D. Scargle, David Barnes, Sundararajan Arabhi, Alina Alt, PeterGioumousis, ElyusGwin, Paungkaew,Sangtrakulcharoen, Lind a Tan, Tun Tao Tsai

[3] I. S Parallelisation of a Common Changepoint Detection Method, S. O. Tickle, I. A. Eckley, P. Fearnhead, K. Haynes

[4] Wenyu Zhang; Nicholas A James.; David S.Matteson "Pruning and Nonparametric Multiple Changepoint Detection" https://arxiv.org/pdf/1709.06421.pdf

[5] Zvi Roseberg; Jean C. Walrand; P. Bertsekas Distributed Dynamic Programming

[6] Guangming Tan; Ninghui Sun; Guang R. Gao "A Parallel Dynamic Programming Algorithm on a Multi-core Architecture"

[7] M. A. Coram Nonparametric Bayesian Classification, Ph.D. thesis, Department of Statistics, Stanford University, 2002.

[8] P. Fragkou, V. Petridis, and A. Kehagias, A Dynamic Programming Algorithm for Linear Text Segmentation, to appear in Journal of Intelligent Information Systems.

[9] O. Heinonen, Optimal Multi-Paragraph Text Segmentation by Dynamic Programming, Proceedings of COLING-ACL '98, pp. 1484-1486, Montreal, Canada http://arXiv.org/abs/cs/9812005

[10] P. Hubert, Change points in meteorological time series, in Applications of Time Series Analysis in Astronomy and Meteorology, T. Subba Rao, M. Priestley, and O. Lessi, eds., 1997, Chapman and Hall.

[11] P. Hubert, The segmentation procedure as a tool for discrete modeling of hydrometeorogical regimes, Stoch. Env. Res. and Risk Ass., 14, 2000, 297-304.

[12] S. Kay, Optimal Segmentation of Time Series Based on Dynamic Programming, unpublished manuscript, 1988?