CAN Data Processing

Sequential vs Parallel implementation

List of Authors (Adam Jesse, Trey Plichta)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: ajesse@oakland.edu, treyplichta@oakland.edu

***Abstract*——For this project we will read sample CAN data from a test file and perform calculations in C++ using the data. We will compute the minimum, maximum, and average values for the signals. Acceleration will also be computed using vehicle speed at certain times. Acceleration results will be used to analyze the driving patterns and find periods of hard braking and hard acceleration. Additionally, histograms will be computed for engine speed, vehicle speed, and acceleration. We will perform these computations using a sequential and parallel approach to compare computation time. The parallel approach will be done using Intel's oneAPI Threading Building Blocks. Additionally, a MATLAB script will be created to verify the computational accuracy of the C++ programs.**

I. Introduction

In the landscape of automotive technology, the efficient analysis of Controller Area Network (CAN) data plays a pivotal role in understanding and optimizing vehicle performance. This project shows the capabilities of C++ programming in handling and interpreting CAN data. As vehicles become increasingly sophisticated, the need for robust data processing tools becomes more apparent, and this project addresses this demand by employing a parallelization strategy. The utilization of a multi-stage pipeline and parallel_for loops not only exemplifies the power of C++ programming and its potential in optimizing performance for automotive data analysis.

The findings presented in this project come from the analysis conducted on two distinct processors—a 6-core Intel i7-9750H and an Intel Atom N2600. The strategic implementation of Intel's Threading Building Blocks (TBB) approach yields reductions in computation times, highlighting the effectiveness of parallel computing strategies in enhancing overall computational efficiency. The incorporation of a MATLAB script verification process adds an extra layer of confidence in the accuracy and reliability of the C++ programs.

II. Methodology

A. Data Collection

Data was collected from the controller area network (CAN) of a 2012 Chevrolet Malibu. The standard on-board diagnostics services were used to collect certain information. ODB-II offers service 0x01 (show current data) with parameter identifiers (PID) to specify requested data. Vehicle data recorded was engine coolant temperature (PID 0x05), engine speed (PID 0x0C), vehicle speed (PID 0x0D), fuel tank level input (PID 0x2F), and distance traveled since diagnostic trouble codes (DTCs) were cleared (PID 0x31).

The method of data collected utilized a few different tools. The first being Tracetronic's ECU-TEST software connected with a PEAK Systems PCAN-USB tool. Using ECU-TEST a test case was created to first clear DTCs and request each signal with a 100-millisecond delay between each request to give the engine control module enough time to respond. This part of the loop ran during the entire length of the drive. ECU-TEST records all data on the bus while this testcase is active and places the data in a text file.

To analyze the text file and filter out unwanted data, a simple Python script was used. The Python script would then generate a new text file with the data formatted into three columns: timestamp, identifier, data value. This text file would be read into the C++ program to be further analyzed.

B. Common Set-Up

As described above, there is data from five PIDs/signals stored in an input text file. The program first creates five variables of the data type of a user defined structure called DATA1. DATA1 includes arrays fields for data, PID, timestamp, and an individual variable for datalength. The program first opens the text file, counts the number of each of the five PIDs, and stores that number in the datalength variable for each PID structure. As part of the input from the user, the program collects an input called n_multiplier. This is how many times the user would like to copy the data from the text file, to have a sufficient data size to ultimately compare computation times. This means that the structures that hold the data are dynamically allocated to hold input datalength of that PID multiplied by n_multiplier. Now that memory has been properly allocated, the program can reopen the text file and read the entirety of text file, sorting by PID. Next, the program uses the duplicateArray function to copy the original data into the empty locations previously allocated. This copy and paste process is repeated n*multiplier times. So, if the input data from the text file was {1,2,3,4} and n*multiplier was 2; the data array would look like {1,2,3,4,1,2,3,4} after this step was complete. Finally, memory was allocated for the results of the computations.

This included locations for acceleration and the three histogram results.

C. *Sequential Approach*

With the sequential approach, set-up is complete, and computations can be implemented. The first set of calculations is the maximum vehicle speed, maximum engine speed, maximum engine coolant temperature, maximum fuel level, maximum distance travelled, minimums for every signal, and averages for every signal. The maximums and minimum functions simply implements a for loop that compares every data point and keeps the max or min. The average function uses a for loop to add every data point and divides the datalength. The next set of calculations is the acceleration data. Since the DATA1 structure keeps a timestamp for every data point in addition to the normal data, the acceleration can be computed as change in vehicle speed over change in time (*3.6 to get result in m/(s^2)). The accelerations are chunked off into sets of 100, since the data is sampled every 0.1 seconds the chunks are 10 seconds long. The maximum and minimum of each chunk is computed. Finally, these numbers are compared to 2.7 m/(s^2) and -5.4 m/(s^2), respectively, to see if the 10 second segment should be categorized as a hard breaking event, hard acceleration event, or good driving event. The final stage of the sequential calculations involves finding the histogram of engine speed, vehicle speed, and acceleration. Engine speed has a range of 0 to 8000 rpms with bins every 500 rpms. Vehicle speed has a range of 0 to 160 km/hr with bins every 10 km/hr. Acceleration has a value of -10 m/(s^2) to 10 m/(s^2) with bins every 1 m/(s^2). This concludes the sequential computations.

D. *TBB Approach*

The same calculations just described in the sequential approach section are done with the TBB approach. However, this approach uses many instances of parallelization. First some extra memory must be allocated for the TBB approach. This includes a 2-D array called A that simply stacks the engine speed, vehicle speed, ECT, fuel level, and distance travelled data on top of each other. This is done so the parallel_for in part two has an easy way to pass the data to function it calls. Additionally, memory must be allocated for the partial histograms. There is a user input nt which is the number of partial histograms that the user wishes to compute in an attempt to optimize the program. The memory allocation uses the nt input to allocate the correct amount. The parallelization will be broken down into three parts: a four-stage parallel pipeline which deals with the acceleration data, the main parallel_for loop which deals with the min, max, and calculations in addition to partial histograms for the final histograms, and a parallel reduction which is used in the calculation of every final histogram.

Beginning with the parallel pipeline, let's dive into figure 1. The parallel pipeline is broken up into 4 stages. The first is a serial_in_order stage that has input parameters that include pointers to the vehicle speed data vector, vehicle speed timestamp vector, acceleration vector (currently empty), and an int for datalength. Stage 1 packages this all up into groups of 100 and passes it off to stage 2 in an object of class Vehicle Speed. Stage 2 is a parallel stage which does the computation for acceleration. With the information received from stage 1, stage 2 can compute acceleration as the change in vehicle speed over the change in time (*3.6 to get result in m/(s^2)). It also knows where to store this acceleration data and sends the pointer to stage 3. Stage 3 is a serial_in_order stage that takes the acceleration data for the package of 100 and computes the maximum and minimum value out of that 100. It then wraps them up into an object of class MIN_AND_MAX and sends to stage 4. Stage 4 compares the min and max to -5.4 and 2.7, respectively, to make classifications for hard braking and hard accelerations events. This concudes the parallel pipeline.

The second part of the parallelization is detailed in figures 2 and 3. It is a parallel_for loop which optimizes the calculation of the minimums, maximums, averages, and partial histograms. The minimums, maximums, and averages all nest a parallel_reduce; the min and max to do multiple comparison at once and the average to do many sub-sums at once. This means that not only can minimums of different signals be computed at once, but multiple comparisons can be made at once to find the smallest value. This is all shown in figure 2 as the parallel_for loop increments from 0 to 14. Finally, figure 3 shows the partial histogram calculations. We know that the user uses an input nt to request a certain number of partial histograms be made. Therefore, as parallel_for loop increments from 15 to 15+(nt-1) the partial histograms for engine speed are updated. As it increments from 15+(2nt-1) the partial histograms for vehicle speed are updated. Finally, as it increments from 15+(2nt) to 15+(3nt-1) the partial histograms for acceleration are made. This concludes the main parallel_for section.

The last part of the parallelization strategy involves using parallel_reduce to combine the partial histograms into one final histogram. This means that if nt is equal to 4, engine speed partial histogram 1 can be added to partial histogram 2 while 3 is added to 4, and the results added together. This would save one set of the addition operation across the partial histogram vector as opposed to doing it sequentially and only increases in savings as nt is increased. This partial histogram reduction is done for engine speed, vehicle speed, and acceleration. The three sets of reductions are optimized within but execute sequentially when looking across the bunch. One potential improvement area could be implementing a parallel_for loop that allows this three reductions to be executed in parallel.

E. *Common Clean_up*

To clean up the program, the results computed are printed to the terminal. The number of instances of hard acceleration, hard breaking and cruising are printed using the pipeline_result vector indices 0, 1, and 2 respectively. Next the minimum and maximum indexes are assigned to the respective signal index variable using the max and min arrays. Using the indices, the proper array can be indexed to print out the corresponding minimum and maximum value.

The average values are displayed using the avg array that contains the values themselves as opposed to the indices. The 3 histograms that are outputted as .bofs are printed using the bin size variables, max value variables, and histogram arrays for the corresponding signal. After the histograms are displayed, the vectors are written to .bofs using the fwrite C++ function.

The start and end times are printed as well as the elapsed time to compare against both approaches. The start time is taken before the actual computations directly after the end of the common setup. The end time is taken right before the start of the common clean up.

The last step is to free the dynamically allocated memory by using the free C++ function.

III. Experimental Setup

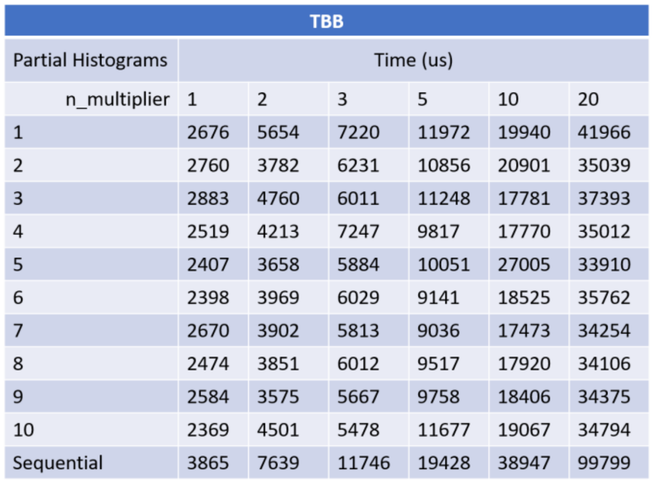
A MATLAB program is used to verify results of the C++ program. The MATLAB program reads the same data text file that is used in the C++ program. Using the table2array function and indexing the correct column of the table, three arrays are created for time, identifier, and data value. The data value array is split into five arrays corresponding to the PID. Using these arrays, the minimum, maximum, and average values are found using the built in min, max, and mean MATLAB functions and printed to the terminal.

The acceleration is found using the same algorithm of point slope formula used in the C++ program. This is done by finding the rows of the data table where the PID is equal to 13 and storing the time stamps in these rows into a new time vector labeled time_speed. To create a vector of the time difference between each point, the diff MATLAB function was used on both the time_speed vector and the speed_values vector and stored in dt and dv respectively. Using dt and dt, acceleration can easily be computed using the element-wise division ./ of dv over dt with dt multiplied by 3.6 to covert to meters per second. Next to verify the driving flags given by the C++ program, the instances of hard acceleration and deceleration are found. This can be done by setting a group_size of 100 which is equivalent to the length of the arrays that are fed into the parallel pipeline. The number of groups is found by taking the floor of the acceleration vector length divided by the group_size. Using the number of groups, two zero arrays are created for minimum acceleration and maximum acceleration. Next a loop iterates from 1 to num_groups setting start_idx equal to the current iteration minus 1 multiplied by the group size plus 1. An end index is created as the current iteration multiplied by group. This allows the acceleration vector to be broken into segments of 100 values. The maximum and minimum values of each segment are stored in the max_acceleration and min_acceleration variables. The maximum and maximum values are narrowed down further to minimum values less than -5.4 and maximum values greater than 2.7. The lengths of the filtered minimum and maximum vectors indicate the number of hard acceleration and hard deceleration.

The last verification calculations needed are the generations of histograms. 6 histograms are made for data values of RPM, speed, ECT, distance traveled, fuel percentage, and acceleration. The histogram MATLAB function with parameters of the corresponding data vectors and bins is used to create the histograms. The last 3 histograms are generated by reading in the binary output files (.bof) from the C++ program. The fopen and fread MATLAB functions are used to read in the files and extract the data. By comparing all the histograms, the accuracy of the C++ program can be verified.

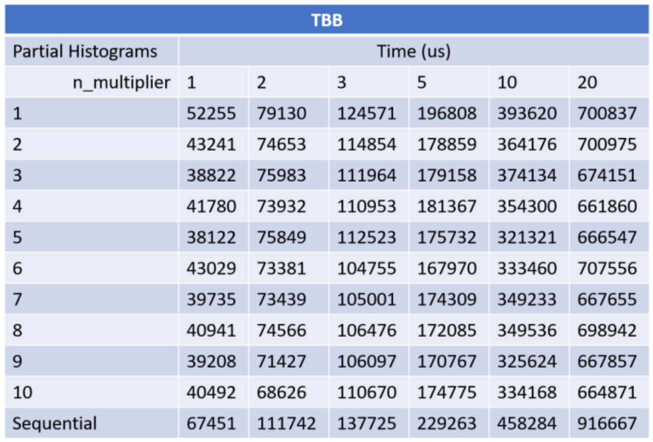
IV. Results

To compare the sequential programming approach and the TBB program approach, each program was run multiple times with varying parameters. The sequential program was run with the parameter of n_multiplier equal to 1 through 10 as well as 20 and 50. The TBB program was run with the same n_multiplier values but at each value, the parameter indicating the number of partial histograms is incremented from 1 to 10. The table with results from a 6 core Intel i7-9750H are shown in Table 1. The results from the Terasic DE2i-150 FPGA Development kit using the Intel Atom N2600 processor are shown in Table 2.



TBB						
Partial Histograms	Time (us)					
n_multiplier	1	2	3	5	10	20
1	2676	5654	7220	11972	19940	41966
2	2760	3782	6231	10856	20901	35039
3	2883	4760	6011	11248	17781	37393
4	2519	4213	7247	9817	17770	35012
5	2407	3658	5884	10051	27005	33910
6	2398	3969	6029	9141	18525	35762
7	2670	3902	5813	9036	17473	34254
8	2474	3851	6012	9517	17920	34106
9	2584	3575	5667	9758	18406	34375
10	2369	4501	5478	11677	19067	34794
Sequential	3865	7639	11746	19428	38947	99799

Table 1 – Computation times for Intel i7-9750H

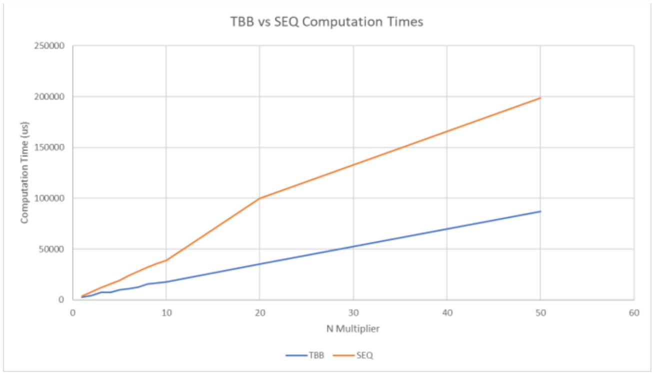


TBB						
Partial Histograms	Time (us)					
n_multiplier	1	2	3	5	10	20
1	52255	79130	124571	196808	393620	700837
2	43241	74653	114854	178859	364176	700975
3	38822	75983	111964	179158	374134	674151
4	41780	73932	110953	181367	354300	661860
5	38122	75849	112523	175732	321321	666547
6	43029	73381	104755	167970	333460	707556
7	39735	73439	105001	174309	349233	667655
8	40941	74566	106476	172085	349536	698942
9	39208	71427	106097	170767	325624	667857
10	40492	68626	110670	174775	334168	664871
Sequential	67451	111742	137725	229263	458284	916667

Table 2 - Computation times for Intel N2600

Table 1 and Table 2 show how dominate the TBB approach is especially for larger values of n_multiplier. The processing time of the N2600 for 6 million elements and optimal number of partial histograms (10) takes about 665 milliseconds to process where the same calculations using a sequential approach takes 917 milliseconds which is roughly a 38% increase.

A line graph showing the computation times as n_multiplier increases for a 6 core Intel i7-9750H, and 4 partial histograms is shown in graph 1. This shows that even for small values of n_multiplier, TBB is the optimal approach.



Graph 1 – Sequential vs TBB computation times

V. CONCLUSIONS

This project dives into automotive data analysis, showcasing the versatility of C++ programming in handling and interpreting CAN data. The parallelization strategy, implemented through a multi-stage pipeline and parallel_for loops, showcases the potential for optimizing performance. The results obtained from a 6-core Intel i7-9750H and an Intel Atom N2600 processor come together to support the TBB approach, with significant reductions in computation times. Additionally, the verification process using a MATLAB script adds an extra layer of confidence in the accuracy of the C++ programs, emphasizing the importance of cross-validation in data analysis projects. In conclusion, this project not only contributes to the field of automotive data analysis but also shows the importance of adopting parallel computing strategies for enhanced computational efficiency.

REFERENCES

[1] Llamocca, D. (n.d.). High-performance embedded programming with the Intel Atom Platform. https://www.secs.oakland.edu/~llamocca/emb_intel.html

[2] Wikimedia Foundation. (2023, November 22). *OBD-II Pids*. Wikipedia. https://en.wikipedia.org/wiki/OBD-II_PIDs

[3] *Intel® oneapi threading building blocks*. Intel. (n.d.). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.26ck4i

Appendix

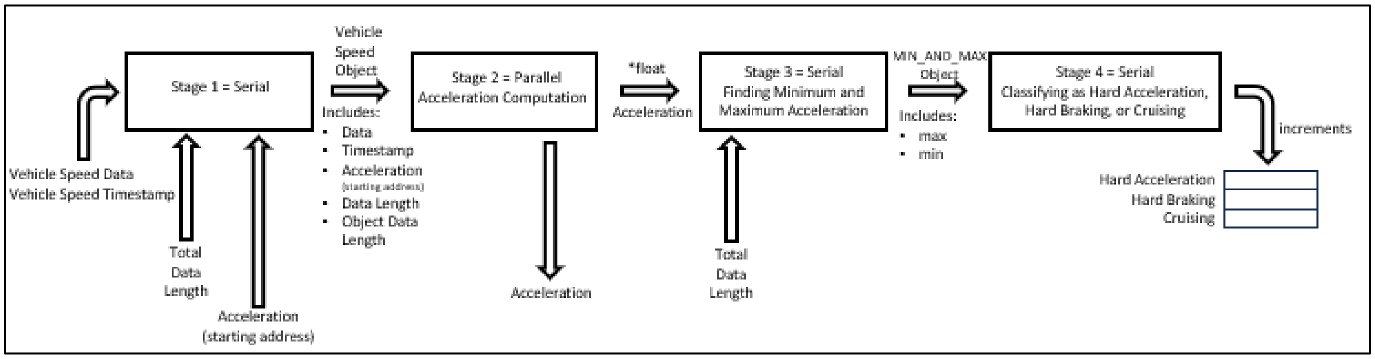


Figure 1 – Parallel Pipeline – TBB Approach Part 1

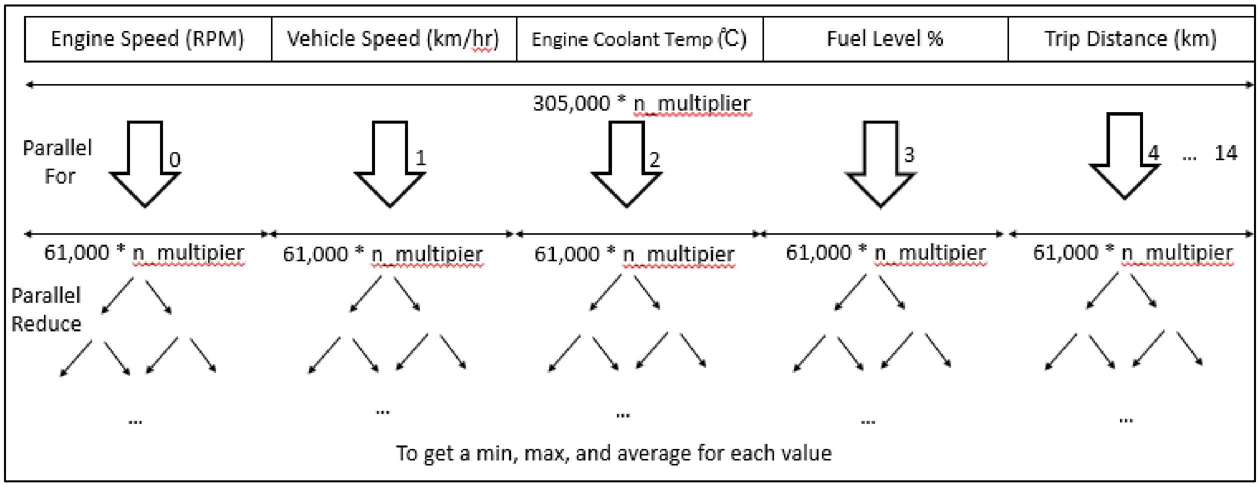


Figure 2 – Main parallel_for – TBB Approach Part 2

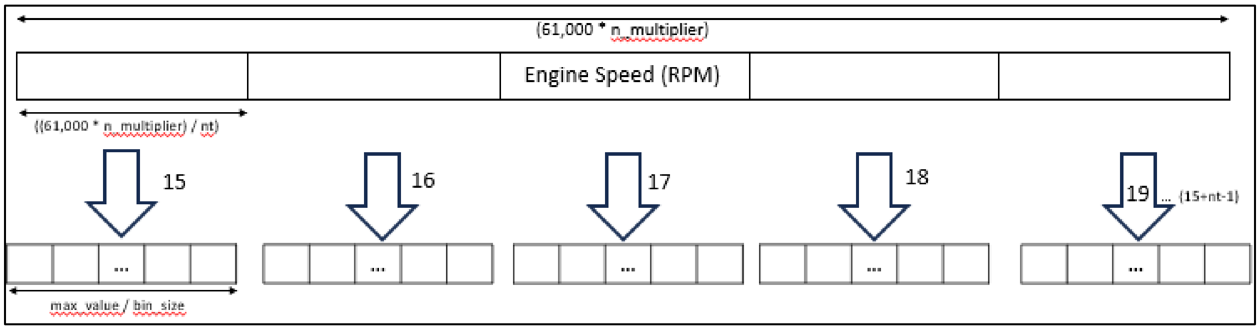


Figure 3 – Main parallel_for – TBB Approach Part 2

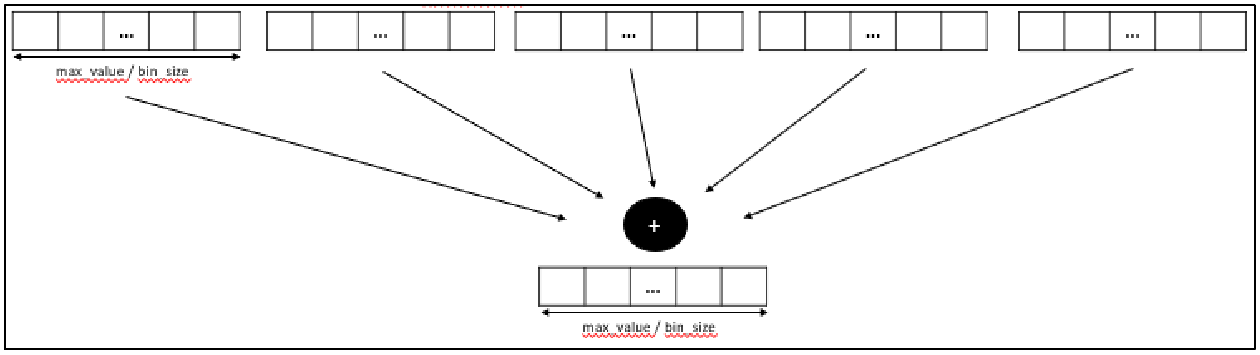
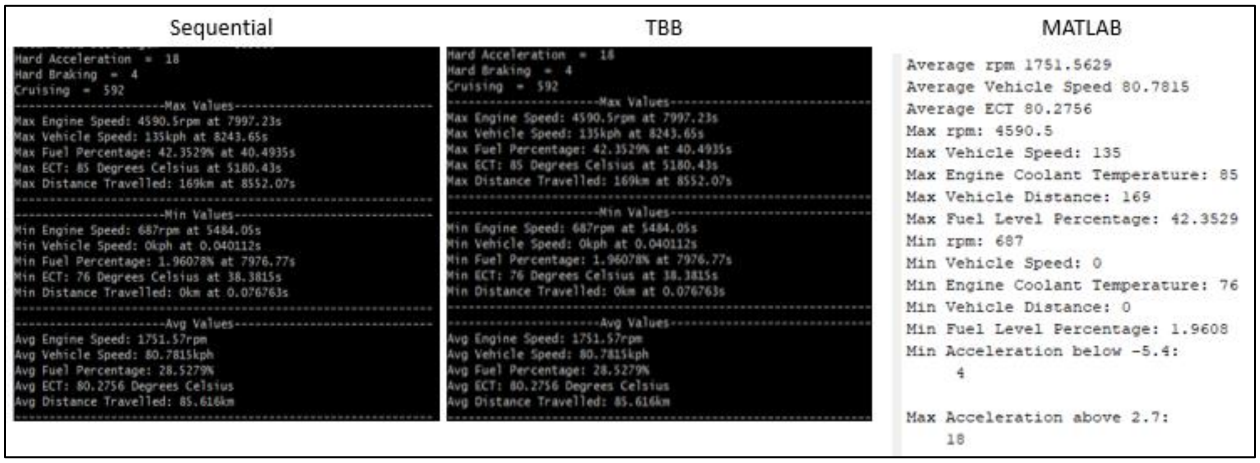


Figure 4 – Parallel Reduction – TBB Approach Part 3



```
Hard Acceleration  =  18
Hard Braking  =  4
Cruising  =  592
---------------------Max Values------------------------------
Max Engine Speed: 4590.5rpm at 7997.23s
Max Vehicle Speed: 135kph at 8243.65s
Max Fuel Percentage: 42.3529% at 40.4935s
Max ECT: 85 Degrees Celsius at 5180.43s
Max Distance Travelled: 169km at 8552.07s
-------------------------------------------------------------
---------------------Min Values------------------------------
Min Engine Speed: 687rpm at 5484.05s
Min Vehicle Speed: 0kph at 0.040112s
Min Fuel Percentage: 1.96078% at 7976.77s
Min ECT: 76 Degrees Celsius at 38.3815s
Min Distance Travelled: 0km at 0.076763s
-------------------------------------------------------------
---------------------Avg Values------------------------------
Avg Engine Speed: 1751.57rpm
Avg Vehicle Speed: 80.7815kph
Avg Fuel Percentage: 28.5279%
Avg ECT: 80.2756 Degrees Celsius
Avg Distance Travelled: 85.616km
-------------------------------------------------------------
```

```
Hard Acceleration  =  18
Hard Braking  =  4
Cruising  =  592
---------------------Max Values------------------------------
Max Engine Speed: 4590.5rpm at 7997.23s
Max Vehicle Speed: 135kph at 8243.65s
Max Fuel Percentage: 42.3529% at 40.4935s
Max ECT: 85 Degrees Celsius at 5180.43s
Max Distance Travelled: 169km at 8552.07s
-------------------------------------------------------------
---------------------Min Values------------------------------
Min Engine Speed: 687rpm at 5484.05s
Min Vehicle Speed: 0kph at 0.040112s
Min Fuel Percentage: 1.96078% at 7976.77s
Min ECT: 76 Degrees Celsius at 38.3815s
Min Distance Travelled: 0km at 0.076763s
-------------------------------------------------------------
---------------------Avg Values------------------------------
Avg Engine Speed: 1751.57rpm
Avg Vehicle Speed: 80.7815kph
Avg Fuel Percentage: 28.5279%
Avg ECT: 80.2756 Degrees Celsius
Avg Distance Travelled: 85.616km
-------------------------------------------------------------
```

```
Average rpm 1751.5629
Average Vehicle Speed 80.7815
Average ECT 80.2756
Max rpm: 4590.5
Max Vehicle Speed: 135
Max Engine Coolant Temperature: 85
Max Vehicle Distance: 169
Max Fuel Level Percentage: 42.3529
Min rpm: 687
Min Vehicle Speed: 0
Min Engine Coolant Temperature: 76
Min Vehicle Distance: 0
Min Fuel Level Percentage: 1.9608
Min Acceleration below -5.4:
     4

Max Acceleration above 2.7:
    18
```

Figure 5 – Computational Accuracy Verification for Max, Min, Avg, and Driving Analysis

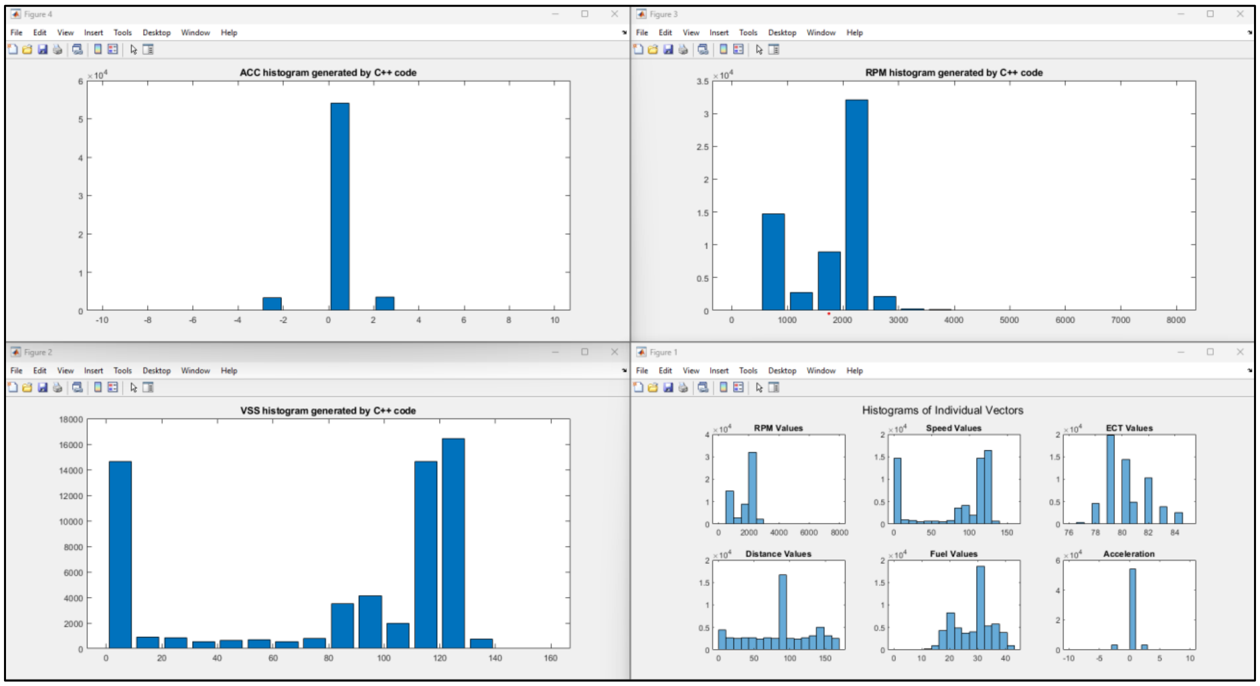


Figure 6 – Computational Accuracy Verification for Sequential Histograms with MATLAB (bottom right)

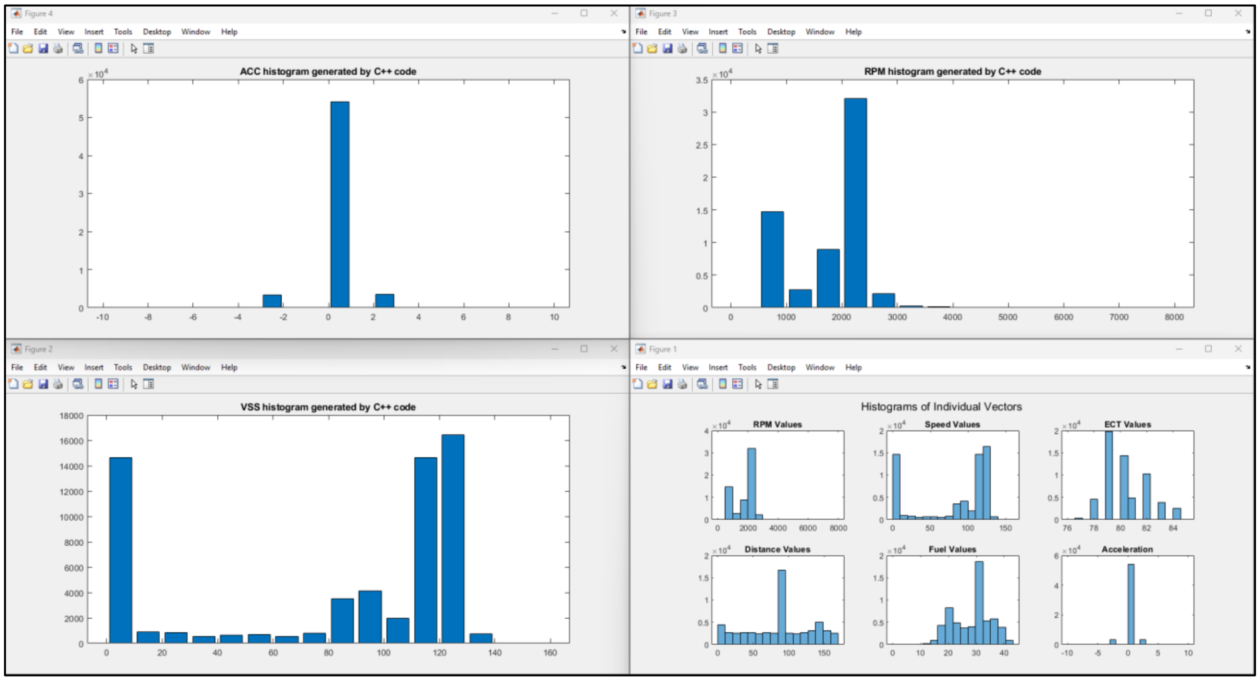


Figure 7 – Computational Accuracy Verification for Parallel Histograms with MATLAB (bottom right)