Parallelized K-means Clustering Algorithm for Target GPS Coordinates

Osvaldo Salinas

eid: ons239

April 28, 2023

BACKGROUND

The Austin Fire Department (AFD) needs an aircraft that can identify critical targets needing first aid kits and deliver them. Aircraft Design students are performing a mission simulating stranded people on rooftops. This mission is divided into three sub-teams each tackling one of the following tasks: finding potential targets through image recognition, delivering a map to AFD identifying critical and non-critical targets, and dropping a first aid kit to critical targets. The focus of this project is on making target recognition data usable for generating a map of targets.

The target recognition sub-team gets their data from flight footage from an aircraft flying over targets and live coordinates of the plane. Their code uses a YOLO machine learning model to identify targets that works by searching for targets within frames of the video then ties the position in the frame to GPS Coordinates. This results in a multitude of GPS coordinates for only five targets. To estimate the position of the targets, I will use k-means clustering to group similar GPS coordinates together and find the average points. K-means clustering groups data points into a predefined number of clusters to minimize the squared distance between each data point and its centroid. Steps include choosing the number of clusters, initializing centroids, assigning each point to the nearest centroid, recalculating the centroid, and repeating until convergence or a maximum number of iterations. K-means is used for customer and image segmentation and anomaly detection but may not work well with non-spherical data or varying densities. To demonstrate the efficacy of the clustering, I will display the actual coordinates of the targets then the calculated coordinates. This process will be parallelized using OpenMP.

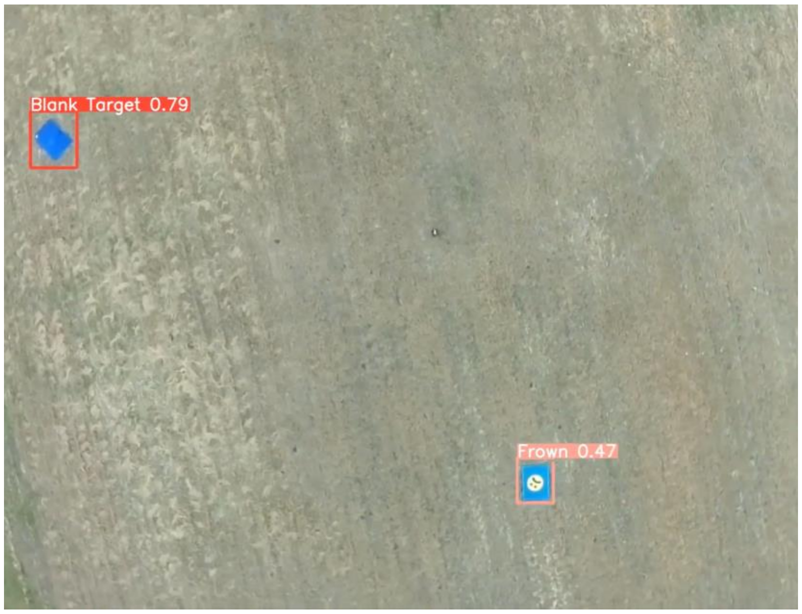


Figure 1: Example of Target Recognition from Footage (link)

COMPILING

1. log onto Frontera from the terminal using:
    a. ssh [TACC Username]@frontera.tacc.utexas.edu.
2. enter your password and TACC token.
3. To locate the code
    a. cd $WORK
    b. cd pcse/ TheArtOfHPC_vol2_parallelprogramming/exercises/exercises-omp-c/
4. initiate an idev session to compile the code by running:
    a. idev
5. Use the following flags to compile the code:
    a. icc kmeans_omp -O2 –xhost –qopenmp
6. Specify the number of threads for the test by setting the environment variable:
    a. export OMP_NUM_THREADS=(#threads)
7. Finally, the basic command to run the code is as follows:
    a. numactl --cpunodebind=0 --preferred=0 ./ kmeans_omp

CODE DESCRIPTION

The C program kmeans_omp.c is a K-Means clustering algorithm implemented in C using OpenMP for parallelism. The program starts by defining constants and data structures, such as the number of clusters (K), the maximum number of iterations, and a point struct. There are fifty GPS points that come from target recognition. The program initializes the centroids and then iterates through the points to assign them to their nearest centroid. It calculates the distance between each point and each centroid using the find_closest_centroid function. The program then updates the centroids' positions by calculating the mean of all the points assigned to each cluster. The means are an estimation of where the actual points are located. This process is repeated until it either reaches 1000 iterations, or the centroids' positions no longer change significantly. To speed up the program's execution, the code uses OpenMP to parallelize the loop that updates the centroids' positions.

The program utilizes the `clock()` function from the `time.h` library to accurately measure the execution time of the computation. By retrieving the number of clock ticks since the program's initiation, the function provides a reliable measure of the duration of the operation. Although the clock ticks' resolution is dependent on the implementation, it typically ranges in nanoseconds. In the code, `clock()` is called twice, firstly to obtain the start time and then again at the end of the computation to obtain the end time. Subtracting the start time from the end time yields the duration of the computation in clock ticks. To convert this value to seconds, it is divided by the constant `CLOCKS_PER_SEC`. The resulting value is then assigned to the `time_taken` variable, which is subsequently returned by the function.

The update_centroids() function is the primary part that has been parallelized, wherein two for loops iterate over the points and centroids, respectively. The outer loop is parallelized using the OpenMP parallel for directive, which distributes the iterations among multiple threads. The inner loop is also parallelized using another OpenMP directive, parallel for reduction. This directive ensures that the sum_x, sum_y, and num_points_in_cluster variables are privately owned by each thread, and their results are combined at the end of the loop using a reduction operation.

Parallelization of update_centroids:

```
double update_centroids(int iteration) {
  int I, j;
  double sum_x[K], sum_y[K];
  int num_points_in_cluster[K];
  double time_taken = 0.0;

  clock_t start_time = clock();

#pragma omp parallel for private(j)
  for (I = 0; I < K; i++) {
    sum_x[i] = sum_y[i] = num_points_in_cluster[i] = 0;

#pragma omp parallel for reduction(+:sum_x[i], sum_y[i],
num_points_in_cluster[i])
    for (j = 0; j < sizeof(points) / sizeof(point); j++) {
      if (find_closest_centroid(points[j]) == i) {
        sum_x[i] += points[j].x;
        sum_y[i] += points[j].y;
        num_points_in_cluster[i]++;
      }
    }

    centroids[i].x = sum_x[i] / num_points_in_cluster[i];
    centroids[i].y = sum_y[i] / num_points_in_cluster[i];
  }

  clock_t end_time = clock();
  time_taken = (double)(end_time – start_time) / CLOCKS_PER_SEC;
  //printf(“Time taken for iteration %d: %f seconds\n”, iteration,
time_taken);

  return time_taken;
}
```

OUTPUT

The output reveals that the algorithm's total running time, an average iteration time, the maximum time taken for a single iteration, and the mean latitude and longitude values for each of the five numbered clusters (0 to 4). Each cluster is represented by a tuple of latitude and longitude values, which correspond to the means of all the data points assigned to that cluster.

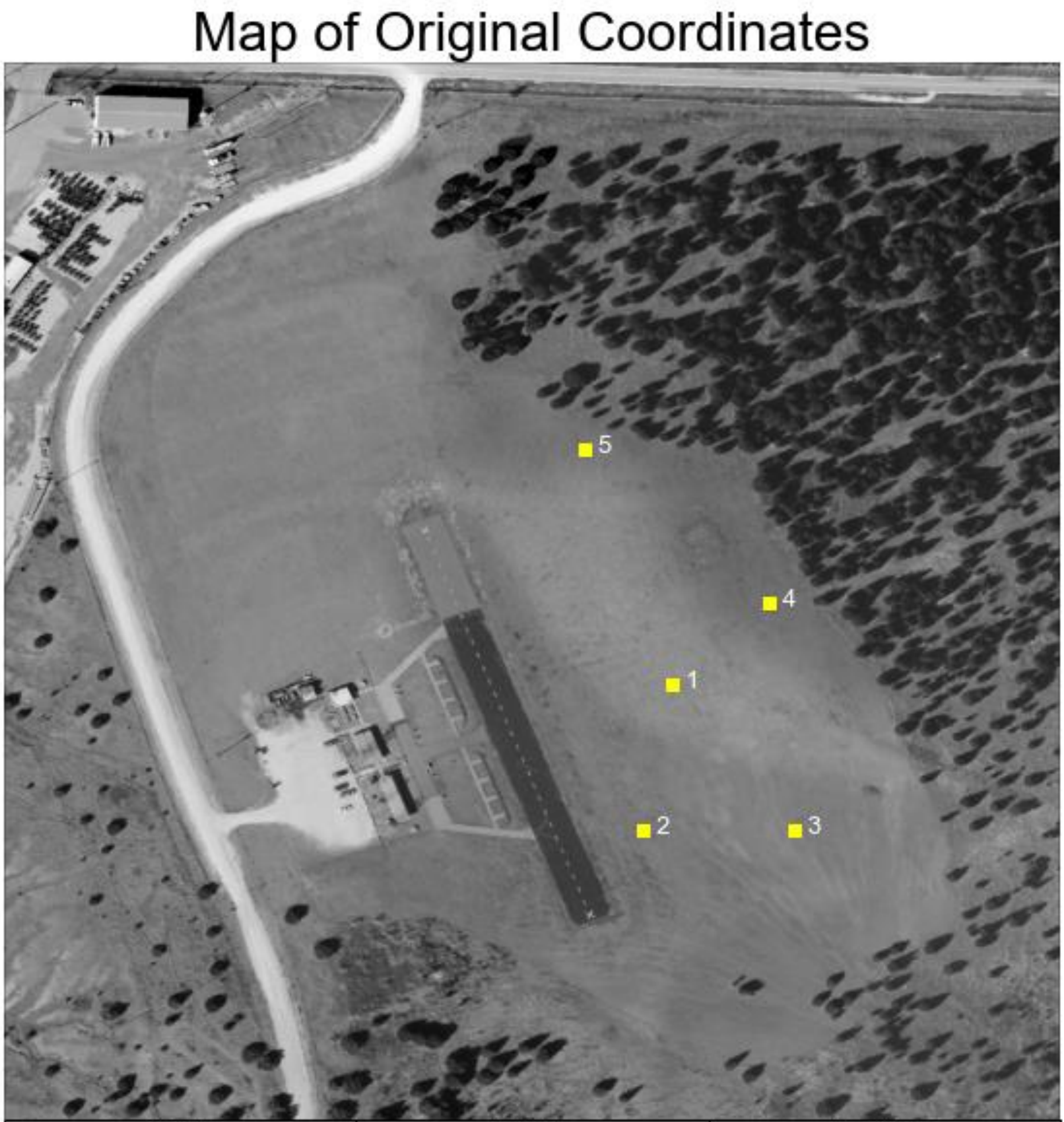
Example output:

```
Max time taken: 1.090000 seconds (iteration 0)
Total time taken: 14.650000 seconds
Average time taken per iteration: 0.014650 seconds
Mean Latitude and Longitude for Each Cluster:
Cluster 0: (30.3242, -97.6027)
Cluster 1: (30.3236, -97.6028)
Cluster 2: (30.3236, -97.6023)
Cluster 3: (30.3245, -97.6024)
Cluster 4: (30.325, -97.6031)
```

The Clustering algorithm kept calculating the same values for latitude and longitude each time. A consistent output suggests a stable and reliable algorithm. This is because producing the same results each time demonstrates that the algorithm is consistent. However, it's important to note that if the clustering algorithm is deterministic, which means it always generates the same output when given the same input, then the results will be identical each time it's run on the same data set. A comparison of the actual vs clustered coordinates for the targets is displayed below in a table and in maps. The map generation was done separately using a python script that uses as satellite image of the Austin Radio Control Association (ARCA) as the background. This is where the flight took place.

Table 1: Actual Coordinates for the 5 targets vs the coordinates calculated through clustering.

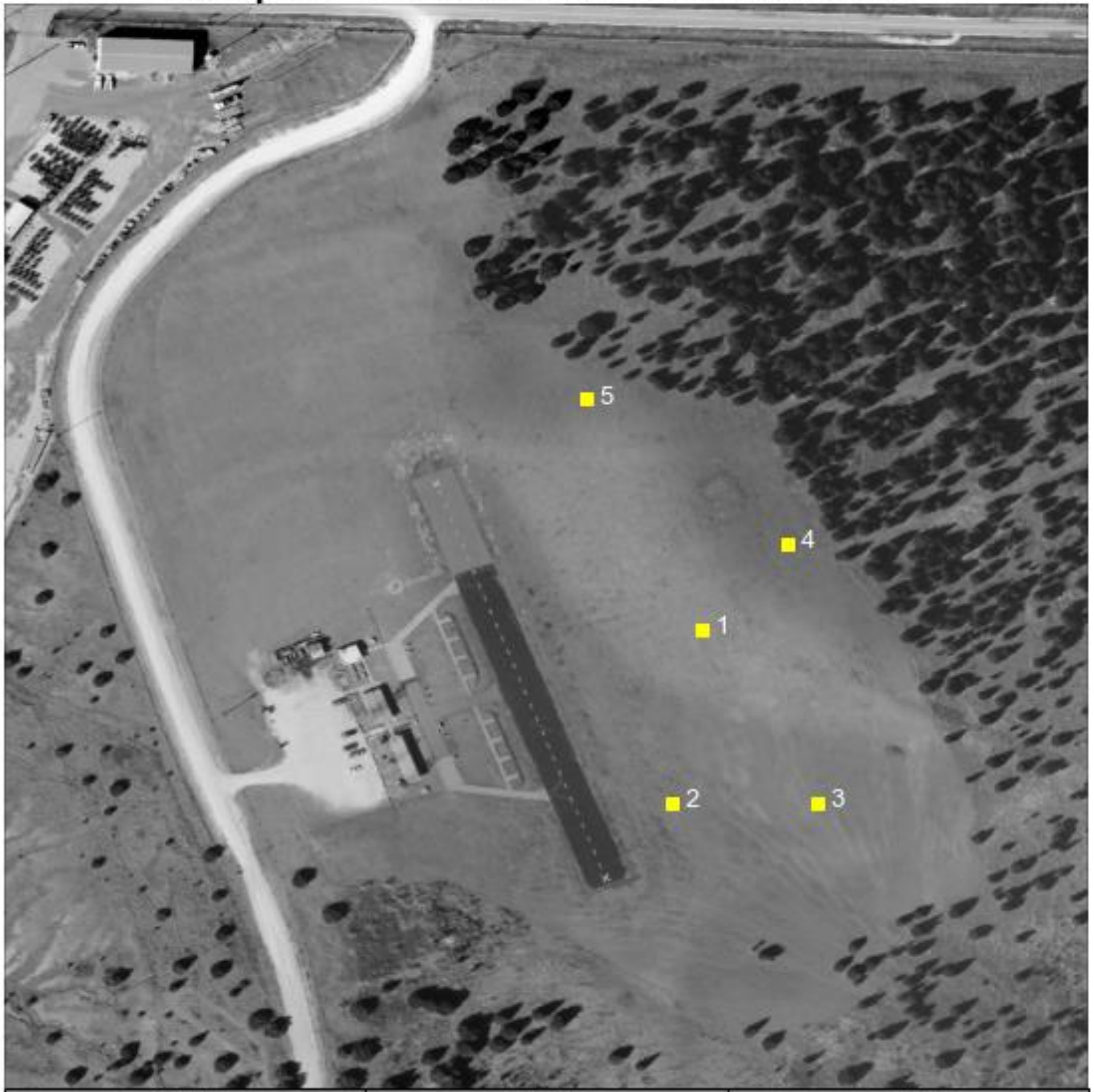
	Actual		Clustered	
Target	**Latitude**	**Longitude**	**Latitude**	**Longitude**
1	30.324161	-97.602743	30.3242	-97.6027
2	30.3236424	-97.6028422	30.3236	-97.6028
3	30.3236424	-97.6023031	30.3236	-97.6023
4	30.324453	-97.602394	30.3245	-97.6024
5	30.324997	-97.603051	30.325	-97.6031



Target	Latitude	Longitude
1.0	30.324161	-97.602743
2.0	30.3236424	-97.6028422
3.0	30.3236424	-97.6023031
4.0	30.324453	-97.602394
5.0	30.324997	-97.603051

Figure 2: Map of the original targets on the airfield. The difference between original an clustered is small and is more noticeable when switching back and forth.

Map of Clustered Coordinates



Target	Latitude	Longitude
1.0	30.3242	-97.6027
2.0	30.3236	-97.6028
3.0	30.3236	-97.6023
4.0	30.3245	-97.6024
5.0	30.325	-97.6031

Figure 3: Map of the targets using calculated values. This map was generated using a python script.

OPENMP RESULTS

The parallelized k-means clustering algorithm was run with different thread counts each time, and 1000 iterations were performed. The runtime for each iteration was measured, and the iteration with the longest runtime, as well as the average iteration time and the total runtimes for all iterations, were determined. The trends for each were shown in figures 4-6.

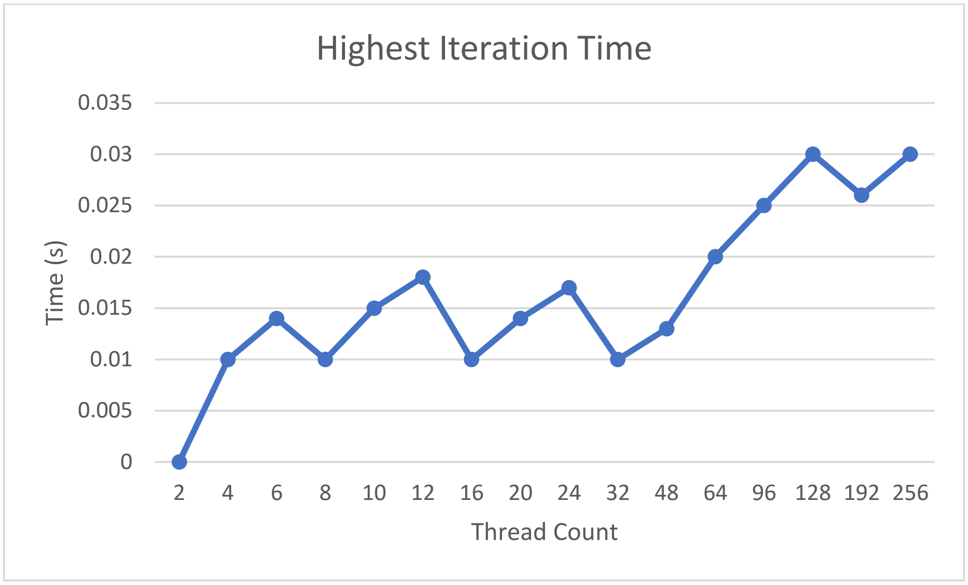


Figure 4: This plot shows the maximum runtime iteration for different thread counts. Despite limited range of time measurements in seconds, an overall increasing trend can be observed in the highest runtime with the increase in the number of threads.

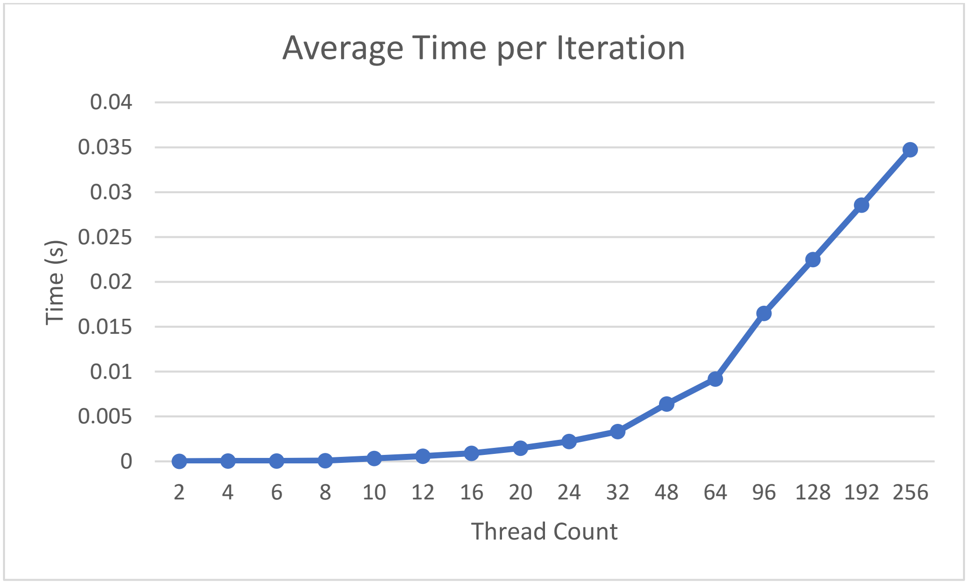


Figure 5: The presented plot illustrates the variation in the average runtime of an iteration across varying thread counts. Notably, the runtimes demonstrate an exponential increase as the number of threads increases.

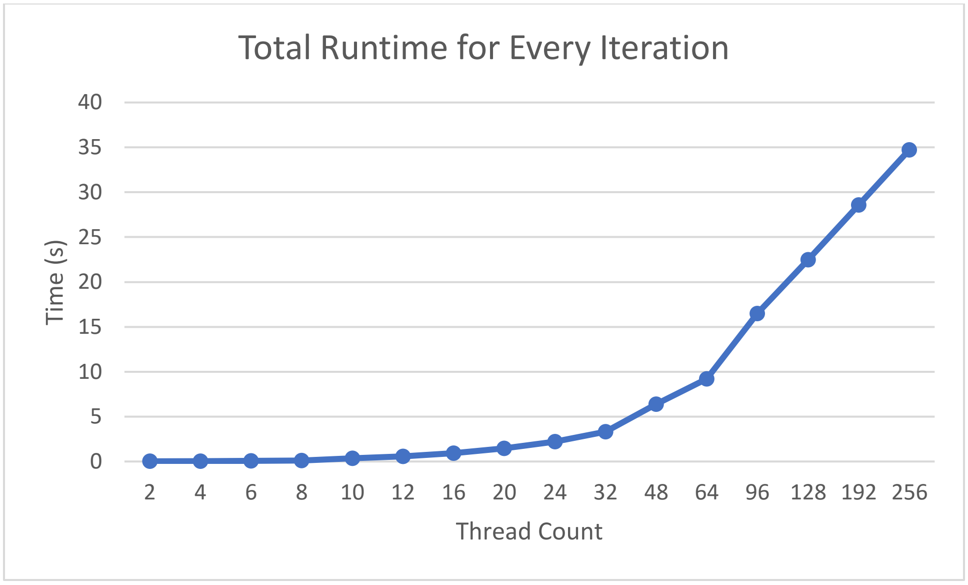


Figure 5: This plot displays the total runtime of all iterations across varying thread counts. It can be observed that there is an exponential increase in runtime as the thread count is increased.

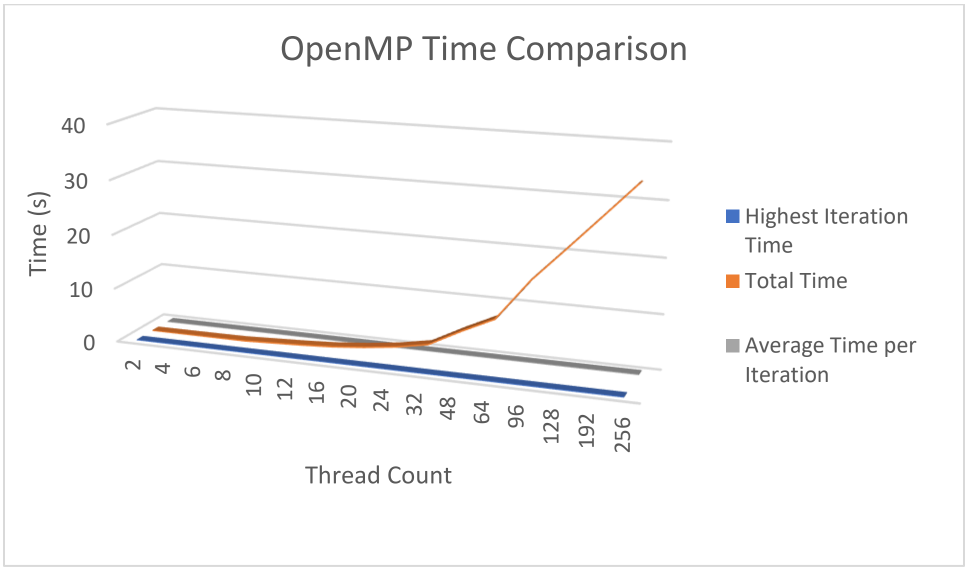


Figure 6: This plot shows how as thread counts increase, the total time increases significantly more than the time it takes to run each iteration individually.

DISCUSSION

The total runtime increased as the thread count increased due to the overhead of managing threads. As more threads are added, the system has to allocate resources such as memory and CPU time to manage them, and they may have to compete for shared resources, leading to inefficiencies and slowing down the execution time. This overhead can outweigh the benefits of parallelism, resulting in longer runtimes. This phenomenon is described by Amdahl's law, which states that the maximum speedup achievable through parallelization is limited by the non-parallelizable fraction of the program. In this case, the overhead of thread creation and synchronization is part of the non-parallelizable fraction, and thus limits the potential speedup that can be achieved.

The increase in total runtime for all iterations as the thread count increased is an important observation that suggests that managing threads incurs significant overhead. The higher runtime and average runtime also increased, albeit not as significantly. By examining both the highest and mean value of a dataset, we can gain a comprehensive understanding of the data's distribution and characteristics. The highest value reveals the upper limit or extreme value, while the mean value provides the central tendency of the data.

Analyzing both values indicates that the iterations in the k-clustering algorithm were successfully parallelized. The efficient distribution of runtime between threads highlights that the parallelization strategy employed was effective. Despite the increase in overall runtime, the algorithm demonstrated the expected parallel speedup, indicating that the parallelization strategy was successful. These findings have important implications for parallelization of algorithms and the efficient use of system resources.

CITATIONS

"K-means clustering." Wikipedia, Wikimedia Foundation, 27 April 2023, https://en.wikipedia.org/wiki/K-means_clustering.