High Performance Computing

Sequential to Parallel Code with OpenMP

Στολτίδης Αλέξανδρος, Κουτσούκης Νικόλαος

November 18, 2021

Contents

1 Introduction

High Performance Computing requires complete CPU utilization. Sequential code execution doesn't take advantage of modern multi-core and hyper-threading CPUs. Parallel and Concurrent Computing help us take advantage of each core's processing power. Parallelism can however have a negative effect and the code execution might be even slower. This can be attributed to obligatory preservation of memory coherence. Memory accesses must also be altered in order to avoid collisions between threads writing on the same cache lines. In this assignment we were given a K-Means implementation that runs sequentially, we analyzed the execution with a profiler (Intel's Vtune), we implemented OpenMP in order to use multiple threads and finally made a few changes to optimize memory accesses.

1.1 Technical Information

Compiler and Flags

- icc -g -fast -qopenmp -o seq_main ...

2 The K-Means Algorithm

To optimize K-Means we must first understand how it works. An image of an execution diagram and some pseudo-code are listed below in order to analyze the flow of the algorithm. In addition, the way data and data structures are stored in memory is also displayed. This is very helpful since it gives us an understanding of how to optimize both the execution, for parallelism, and the memory, for cache locality.

2.1 Code Execution Diagram

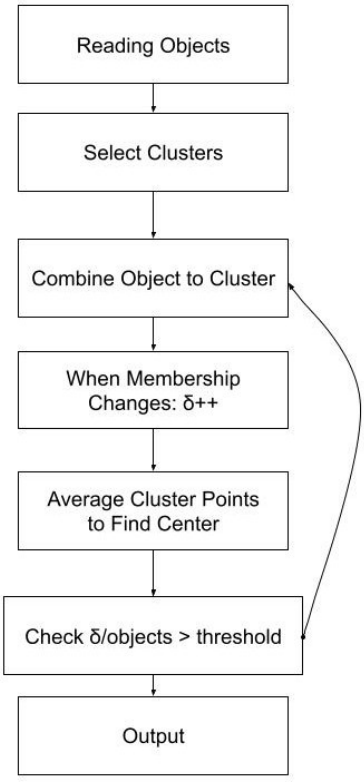

Figure 1: Execution Process of K-Means

2.2 Data Structures

The code mainly works with 4 data structures (Read/Writes) which are dynamically allocated and can be treated as arrays from the programmer's point of view. Some of these data structures however are not exactly arrays since they are stored a bit differently in memory. Here we do not examine how objects[...][...] is stored since it is only read and not changed. Therefore, the objects array would not produce race conditions when adding more threads.

2.2.1 Single Dimensional Arrays

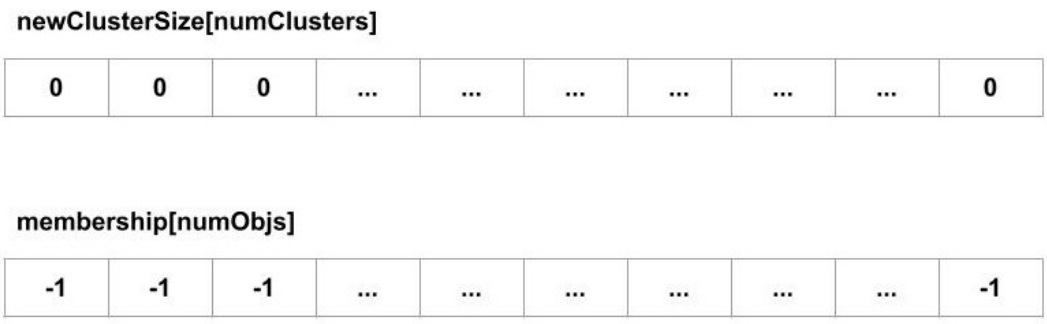

Figure 2: Storing Single Dimensional Arrays in Memory

2.2.2 Multi-Dimensional Arrays

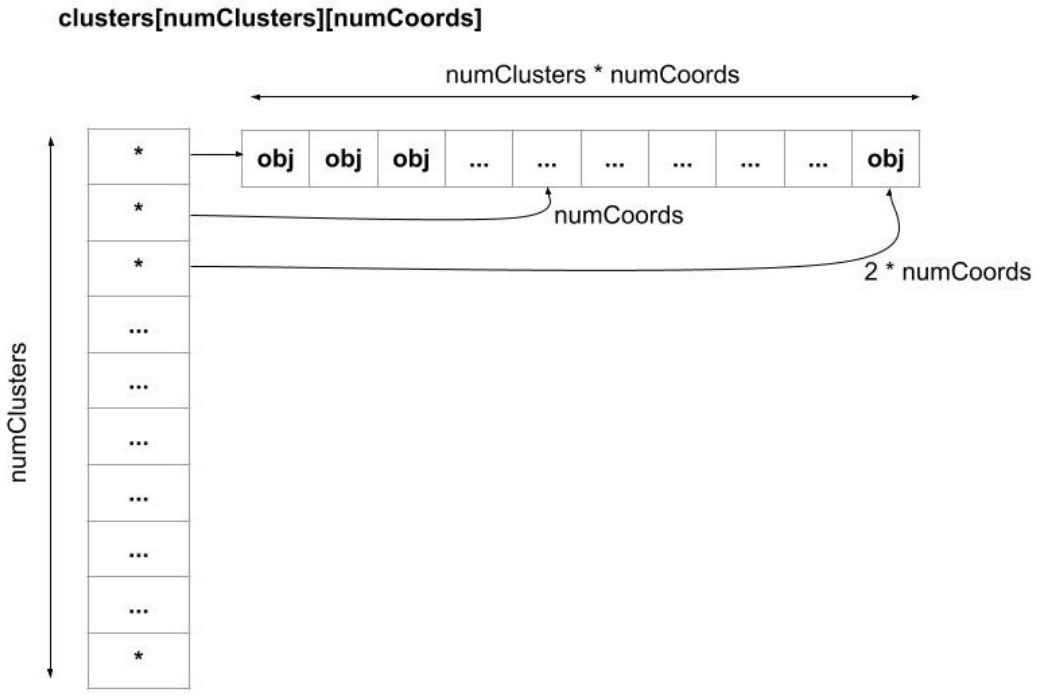

Figure 3: Initialized with Random Objects

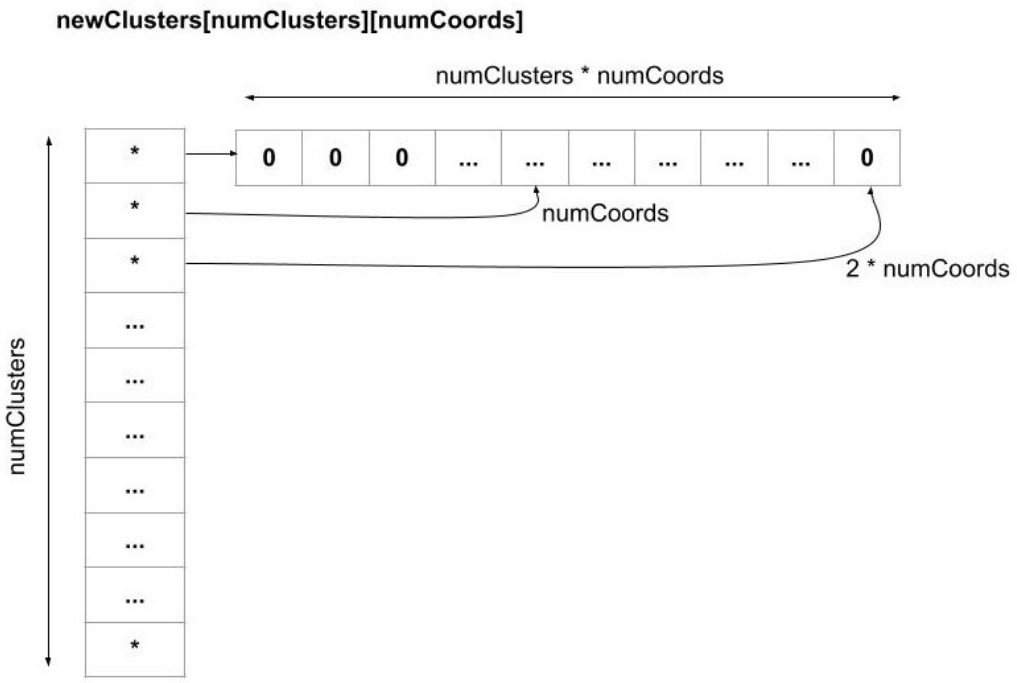

Figure 4: Initialized with Zeroes

2.3 Pseudo-Code

This pseudo-code describes the main calculation loop of K-Means when the code is written for sequential execution. The calculation is mainly consisted of two loops nested inside a threshold loop. The first loop's goal is to select the nearest cluster to object, change membership of objects and accumulate the coordinates of each object associated with a specific cluster in order for a new cluster center to be calculated. The second loop calculates a new cluster center and resets the variables that are needed in the first loop. The latter loop depends on the results of the former loop.

```
do:
    delta <- 0

    //Assign Object to Cluster
    for(i = 0; i < objects; i++):
        //Find Nearest Cluster
        index <- nearest_cluster();

        //Check for Membership Change
        if membership[i] != index:
            delta <- delta + 1

        //Change Membership of Object i to Cluster index
        membership[i] <- index;

        //Update Cluster Center and Set Clusters
        newClusterSize[index] <- newClusterSize[index] + 1;
        for (j = 0; j < coordinates; j++):
            newClusters[index][j] <- newClusters[index][j] + objects[i][j];


    //Average Sum and Replace Contents of cluster[][]
    for (i = 0; i < clusters; i++):
        for (j = 0; j < coordinates; j++):
            if newClusterSize[i] != 0:
                clusters[i][j] = newClusters[i][j] / newClusterSize[i];
            newClusters[i][j] = 0.0;

        newClusterSize[i] = 0;

    delta <- delta / objects
while (delta > threshold)
```

Figure 5: Pseudo-Code of Sequential Execution

2.4 Profiling Sequential Code (VTune)

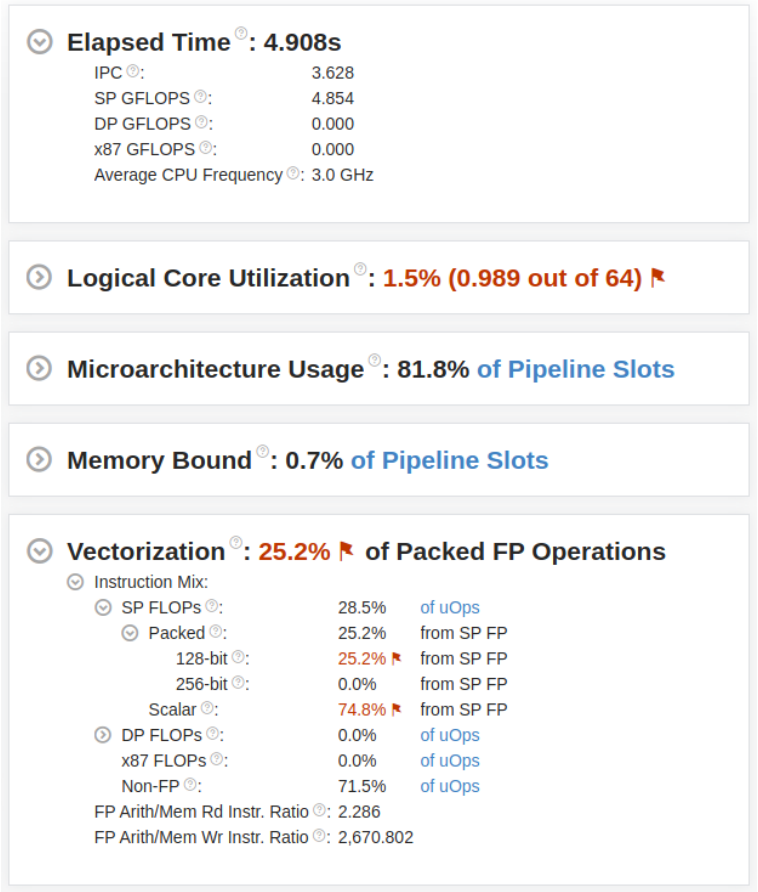

Figure 6: General Snapshot of Sequential Execution

Function / Call Stack	CPU Time ▼		
	Effective Time	Spin Time	Overhead Time
seq_kmeans	4.855s	0s	0s
___fprintf_chk	0.005s	0s	0s
apic_timer_interrupt	0.005s	0s	0s

Figure 7: Bottom-Up Calls and their Corresponding Execution Time

As we can see parallelism is not used and the multi-core CPU is not fully utilized. In order to make the execution faster we must first utilize the resources that the CPU has. Therefore, OpenMP is implemented to distribute the computational load between multiple CPUs. Parallelism is used in seq_kmeans() since it takes most time to complete.

3 Parallel Execution with Critical Section

Sequential execution doesn't take advantage of modern multi-threading CPUs. OpenMP is implemented to make sections of code run in parallel. Each thread can execute a chunk of each loop and the result can be calculated after all threads have finished. Variables that are modified by more than one threads may cause starvation, deadlock or simply wrong results. These variables must be protected by a lock. Locks are implemented in OpenMP as Critical Sections. These critical sections however force threads to synchronize and execute these sections sequentially. We try to get rid of critical sections if possible since they make the code a lot slower.

3.1 Implementation

On the start of the do-while loop all threads are spawned. Threads are spawned in this part of the code in order to minimize the overhead of creating and destroying threads for each nested loop.

```
do:
    delta <- 0

    //Assign Object to Cluster
    for(i = 0; i < objects; i++):
        //Find Nearest Cluster
        index <- nearest_cluster();

        //Check for Membership Change
        if membership[i] != index:
            delta <- delta + 1

        //Change Membership of Object i to Cluster index
        membership[i] <- index;

        //Update Cluster Center and Set Clusters
        newClusterSize[index] <- newClusterSize[index] + 1;
        for (j = 0; j < coordinates; j++):
            newClusters[index][j] <- newClusters[index][j] + objects[i][j];

    //Average Sum and Replace Contents of cluster[][]
    for (i = 0; i < clusters; i++):
        for (j = 0; j < coordinates; j++):
            if newClusterSize[i] != 0:
                clusters[i][j] = newClusters[i][j] / newClusterSize[i];
            newClusters[i][j] = 0.0;

        newClusterSize[i] = 0;

    delta <- delta / objects
while (delta > threshold)
```

Figure 8: Parallel Iterations Critical Section Thread Barrier

3.1.1 First Nested Loop

Threads split the total iterations of the first nested loop into chunks. Since index is a common variable between multiple threads, each thread must have each own copy of this variable and therefore index must be private. In addition, all threads might want to increment one to delta so we use reduction on delta. This feature creates a copy of delta for each thread and finally all copies are merged together when threads meet the implicit barrier at the end of the loop. However, many threads might have the same index value and try to change newClusterSize[index] and newClusters[index][...] at the same time. This race condition may lead to false results since common variables are changed without a lock. In order to avoid unwanted race conditions we insert a critical region to change elements in these arrays. This however turns our parallel code into sequential for a certain amount of time (critical section). Threads must wait, synchronize, and execute the critical region one thread at a time. Critical regions may lead to slower execution times especially when the load on critical section is immense. Critical regions restrict parallel execution and must be removed (if possible) in order to make the code even faster.

3.1.2 Barrier (Implicit)

Since the second nested loop depends on the results of the first one, the implicit barrier that the first loop, by default, has must not be removed.

3.1.3 Second Nested Loop

There are no common variables in the second nested loop. The i iterations are split into chunks and each thread is responsible to execute one chunk. Execution here is completely parallel and time might be only spent in memory operations (Cache Misses and Thread Cache Coherence).

3.2 Profiling Parallel Code with Critical Section (VTune)

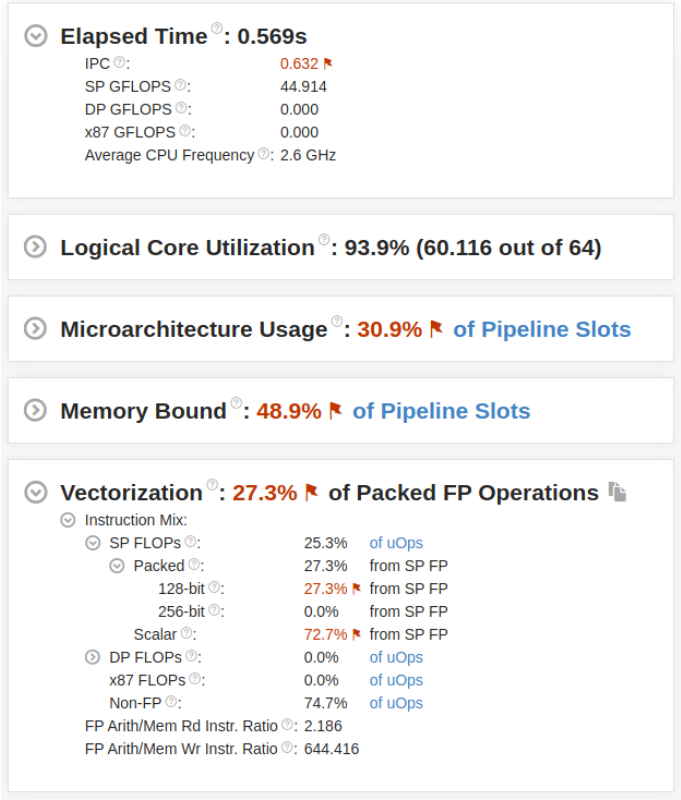

Figure 9: Snapshot Analysis

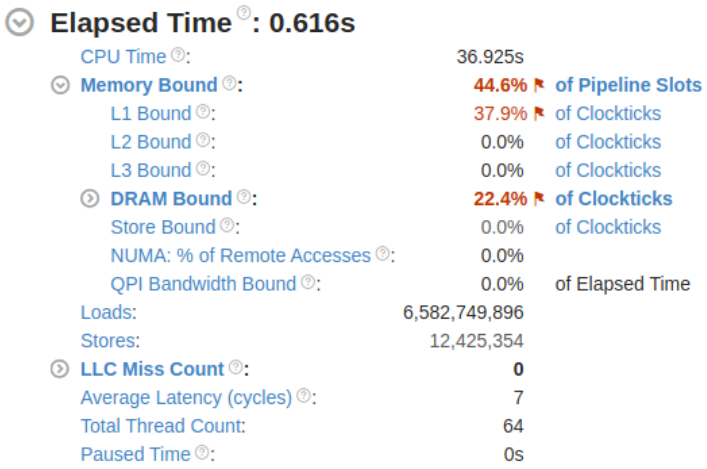

Figure 10: Memory L1 and DRAM Analysis

A critical section creates a small bottleneck when trying to run parallel code. CPU is not fully utilized since threads must wait and synchronize in order to avoid race conditions. In addition, since threading is implemented cache coherence must be maintained. Therefore the program becomes memory bound (L1 Cache Bound). In addition, the pipeline slots are not fully utilized so the CPU receives a lot less data from memory than it can process (DRAM Bound).

4 Parallel Execution

Critical regions do not allow the program to take full advantage of all CPUs at the same time. In addition, maintaining memory coherence also takes a lot of time. Cache lines that are written from one core must also be transferred to all the other cores. Therefore, all threads view the same instance of memory regardless of the CPU that each thread is running on. When multiple threads write on common cache lines a ping-pong effect occurs since cache lines are transferred from one CPU to the others each time a thread performs a write memory operation. Finally, L1 and L2 caches become almost useless since L3 or Main Memory must first be accessed in order to preserve cache coherence.

4.1 Memory Optimizations

To remove critical regions and preserve cache coherence without the unnecessary overhead we can change how data is stored in memory. Each thread can have each own copy of newClusterSize[...] and newClusters[...][...]. The result is calculated by merging the results calculated by each thread together. This is like making data structures that are used my multiple threads private to each thread. Since all threads have their own copy of these data structures, cache lines are accessed by their corresponding thread. This allows the program to maintain cache coherence without transferring cache lines all the time. The critical sections can also be removed since each thread has each own "private" memory and thus no race conditions exist.

4.1.1 Adding a Thread Dimension

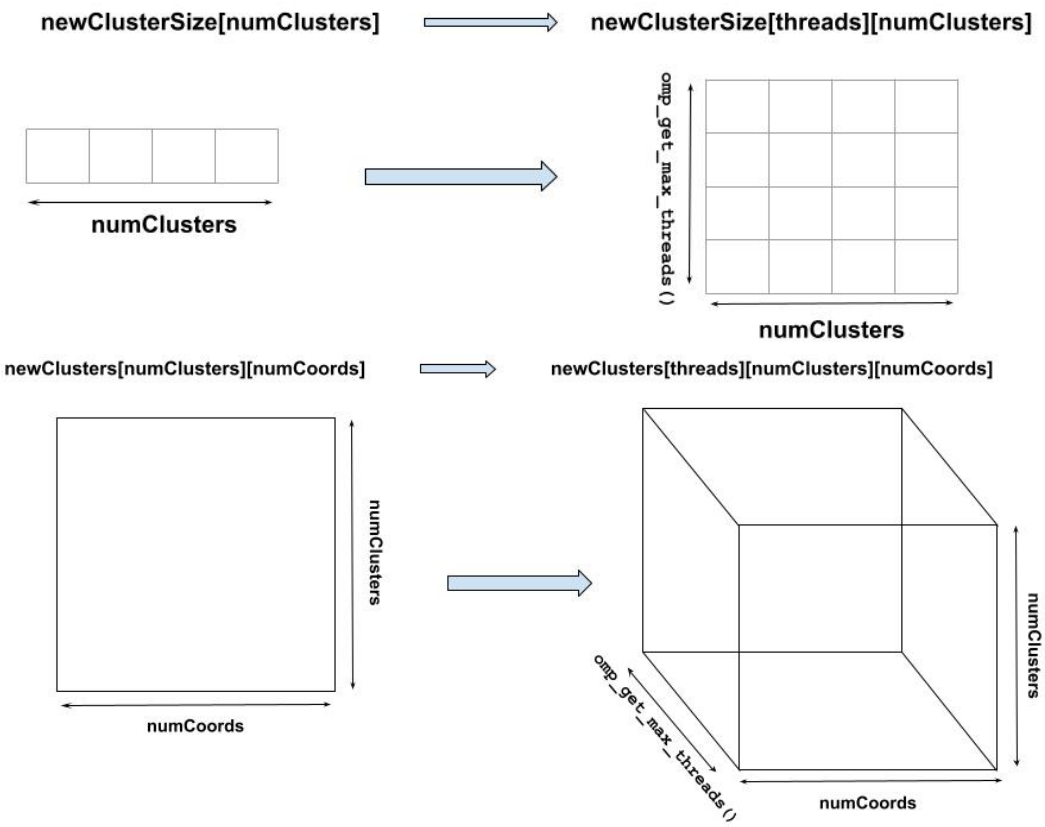

Figure 11: Transformation of Data Structures to Support Multiple Threads

4.2 Implementation

Just like the previous implementation the threads are spawn at the start of the do-while loop. We experimented with different types of loop scheduling and with different chunks of iterations in order to understand whats more optimized.

```
do:
    delta <- 0

    //Assign Object to Cluster
    for(i = 0; i < objects; i++):
        //Get Current Thread Running
        thread <- omp_get_thread_num();

        //Find Nearest Cluster
        index <- nearest_cluster();


        //Check for Membership Change
        if membership[i] != index:
            delta <- delta + 1

        //Change Membership of Object i to Cluster index
        membership[i] <- index;

        //Update Cluster Center and Set Clusters
        newClusterSize[thread][index] <- newClusterSize[thread][index] + 1; //Position Depends on Thread
        for (j = 0; j < coordinates; j++):
            newClusters[thread][index][j] <- newClusters[thread][index][j] + objects[i][j]; //Position Depends on Thread

    //Average Sum and Replace Contents of cluster[][]
    for (i = 0; i < clusters; i++):
        //Get Current Thread Running
        thread = omp_get_thread_num();

        for (j = 0; j < coordinates; j++):
            if newClusterSize[thread][i] != 0:
                clusters[i][j] = newClusters[thread][i][j] / newClusterSize[thread][i];
            newClusters[thread][i][j] = 0.0;

        newClusterSize[thread][i] = 0;

    delta <- delta / objects
while (delta > threshold)
```

Figure 12: Parallel Iterations Thread Barrier

4.2.1 First Nested Loop

As we can see the critical region was removed since each thread has each own "private" copy of memory. Variables like index and thread are private since multiple threads try to retrieve their value and change it. Reduction on delta is still used. Since both newClusterSize[...][...] and newClusters[...][...][...] are initialized with zeroes we do not have to check their values before doing any calculation.

4.2.2 Barrier (Implicit)

Calculations on the second nested loop depend on the results of the first one. Thus, the implicit barrier between those two loops must not be removed.

4.2.3 Second Nested Loop

Thread ID must also be retrieved here in order to access the correct cells of the arrays. Therefore, thread is also a private variable here. When calculating clusters[i][j] we do not check the value of newClusters[thread][i][j] since it is initialized with zeroes and adding zero to the final result doesn't actually change the result.

4.3 Profiling Parallel Code (VTune)

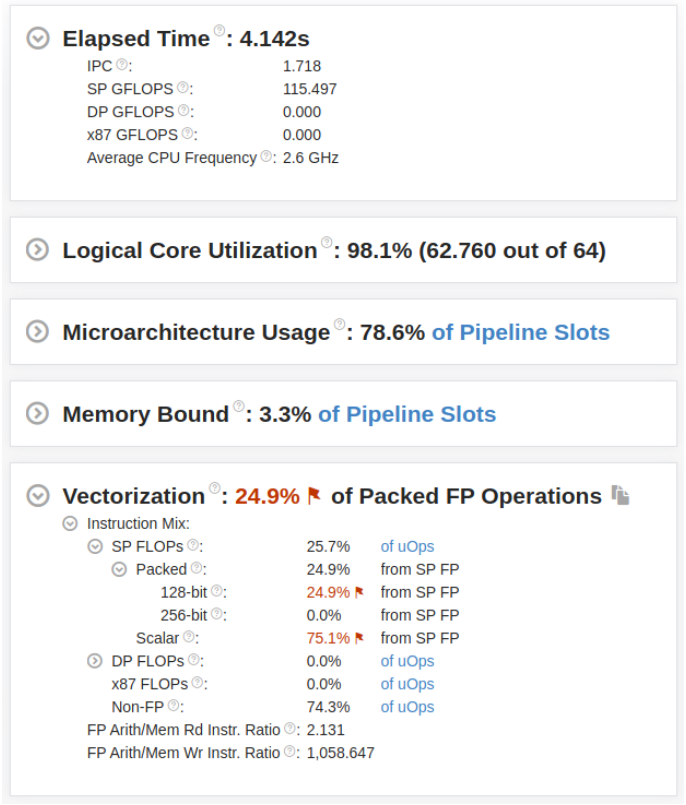

Figure 13: Completely Parallel Execution

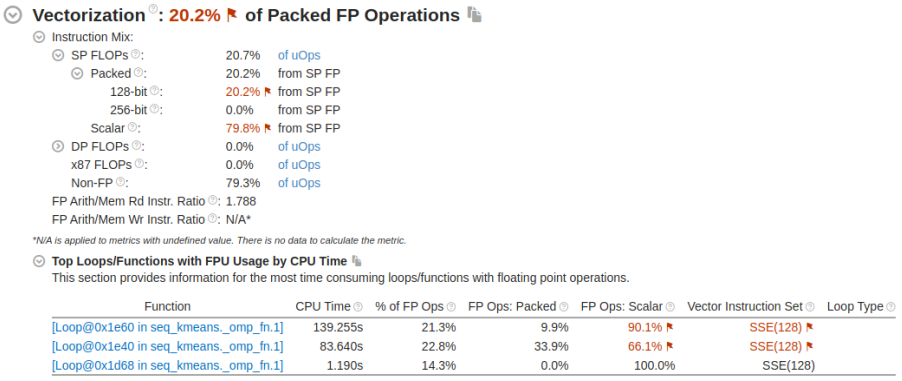

Figure 14: Memory with different Data Structures that Maintain Cache Coherence

All available CPU Threads and pipeline slots are utilized. Memory is also not bound since cache coherence is maintained with the help of the modified data structures used. Absence of Vectorization however creates a huge bottleneck here. More information about scheduling will follow on the statistical analysis.

5 Analysis

5.1 Sequential Execution

Even when more threads are available the execution time is almost constant. Sequential code doesn't try to take advantage of multi-threading.

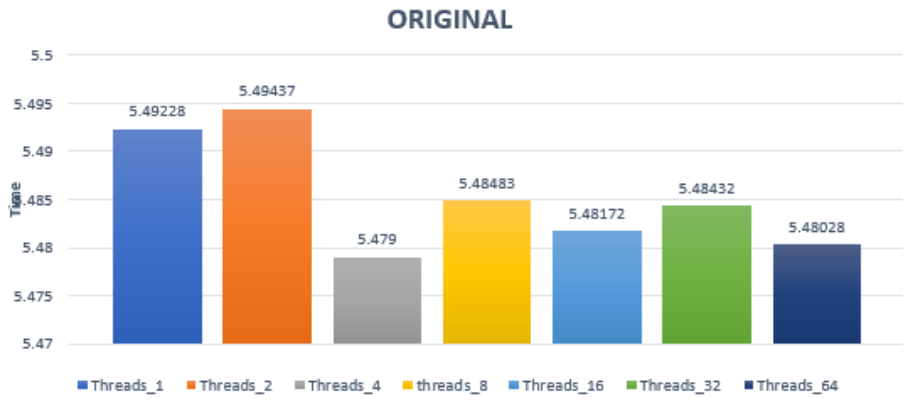

Figure 15: Execution Time Around 5.4 Seconds

5.2 Parallel Execution with Critical Section

When a single thread is available the execution time is the same as it was in the original sequential code. When threads exponentially increase, execution time decreases logarithmically. On 64 threads, execution is a little slower since there is an overhead due to synchronization.

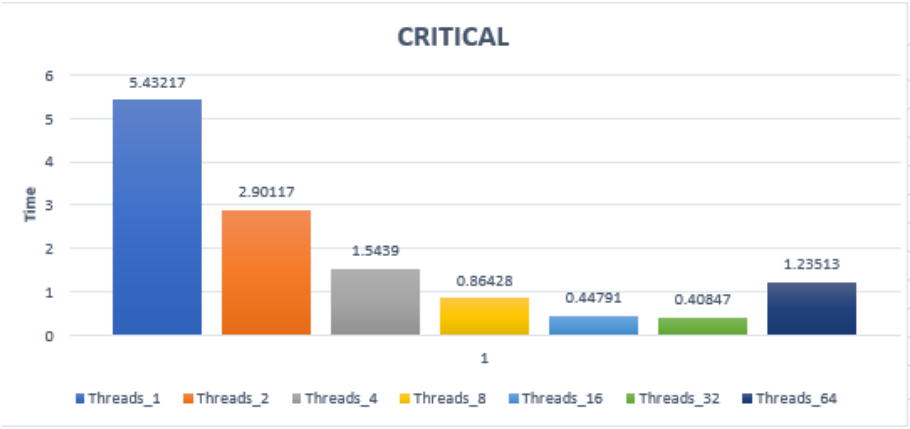

Figure 16: Parallel Execution with Critical Section

5.3 Parallel Execution with Static Scheduling

Static scheduling is recommended for these types of problems since each loop most times executes the same amount of code. When static scheduling is implemented and threads exponentially increase, execution time also decreases logarithmically like before. Contrary to the execution with a critical section here on 64 threads we do not have that much of an overhead since synchronization is not required.

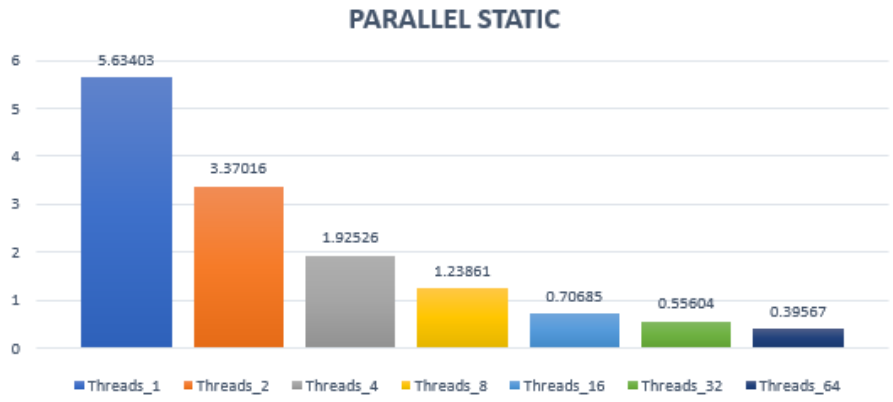

Figure 17: Completely Parallel Execution

5.4 Parallel Execution with Dynamic Scheduling

5.4.1 Default Chunk Size

Each thread takes one iteration to execute. When only a couple of threads are available the execution takes MUCH MUCH longer... This might be due to threads trying to synchronize in order to find the next not taken iteration. Since each iteration in K-Means is not so computation heavy the threads actually spend more time deciding which iteration to take rather than doing actual calculations.

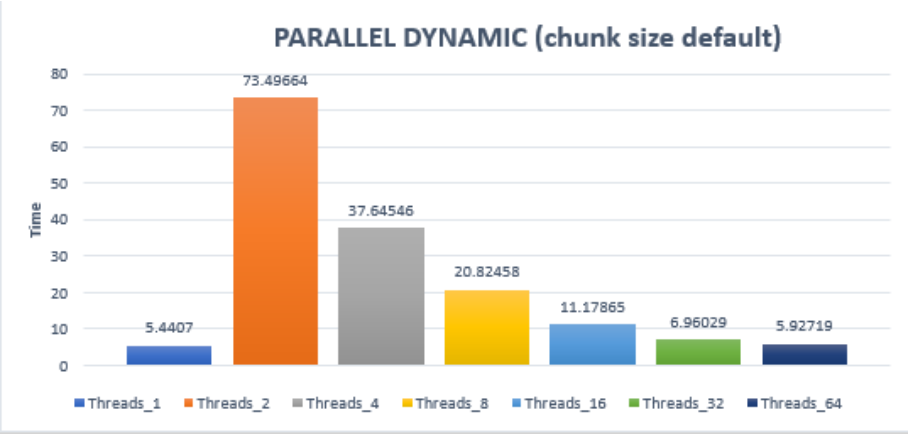

Figure 18: Default Chunk Size (1)

5.4.2 Custom Chunk Size

When a larger than the default chunk size (1) is specified, each thread has much more time to do calculations before trying to find it's next iteration. Execution is generally faster with a custom chunk but not as consistent when few threads are available... This can also be attributed to thread scheduling. Generally, this implementation is the fastest and depends on the system's available threads and the chunk's size compared to the loop's total iterations. However, due to large standard deviation the expected run time of this approach are pretty unpredictable. Thus, this implementation is not recommended since this project focuses on optimizing the code for an unfixed number of threads.

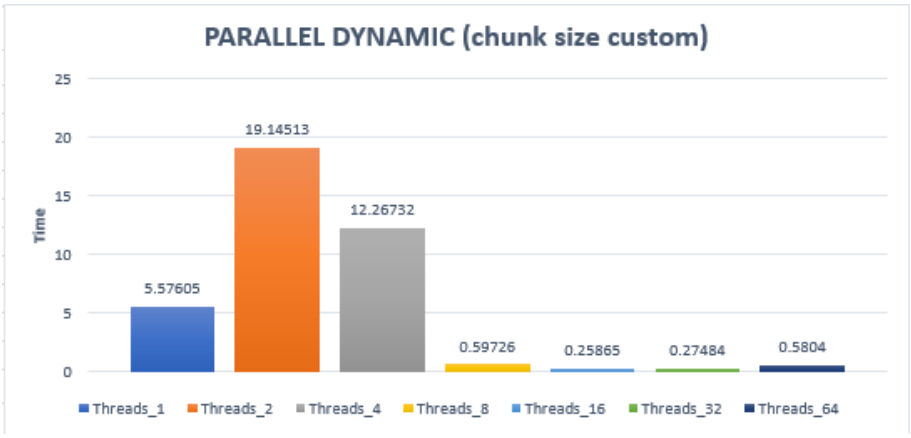

Figure 19: Custom Chunk Size (3000)

5.5 Side by Side Comparison

In order to come to a conclusion and decide which optimization was the most lucrative we compare them side by side. We chose static scheduling since the standard deviation of time is lower and the speedup is almost the same when more threads are available.

5.5.1 Threads: 1

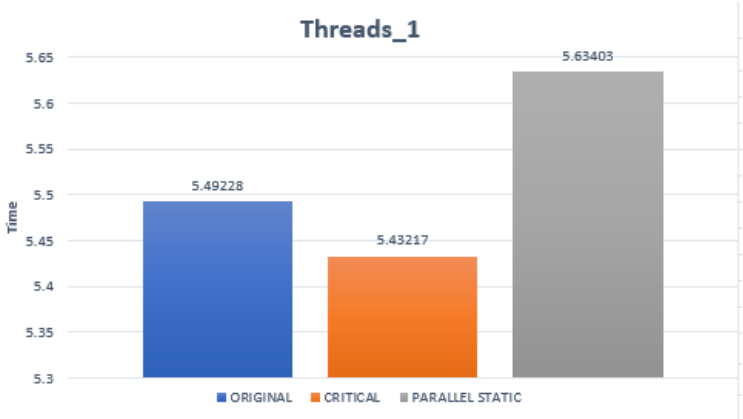

Figure 20: One Thread is Running

5.5.2 Threads: 2

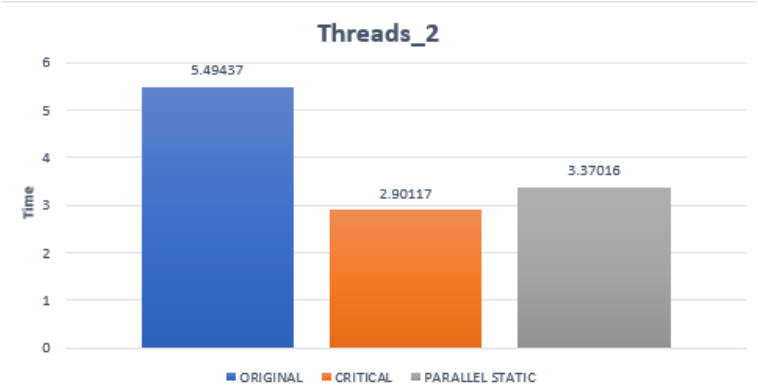

Figure 21: Two Threads are Running

5.5.3 Threads: 4

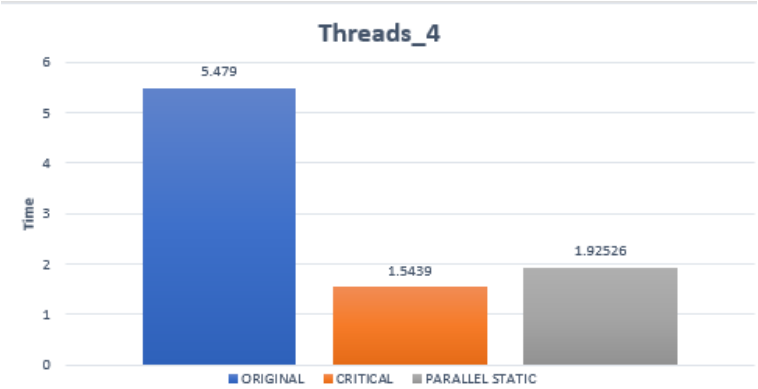

Figure 22: Four Threads are Running

5.5.4 Threads: 16

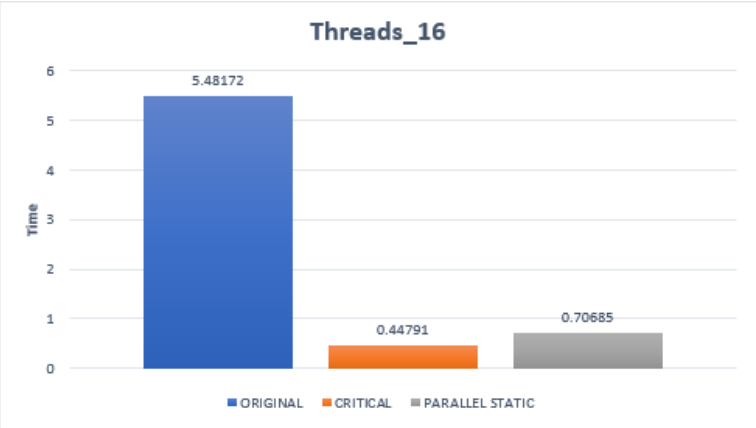

Figure 23: Sixteen Threads are Running

5.5.5 Threads: 32

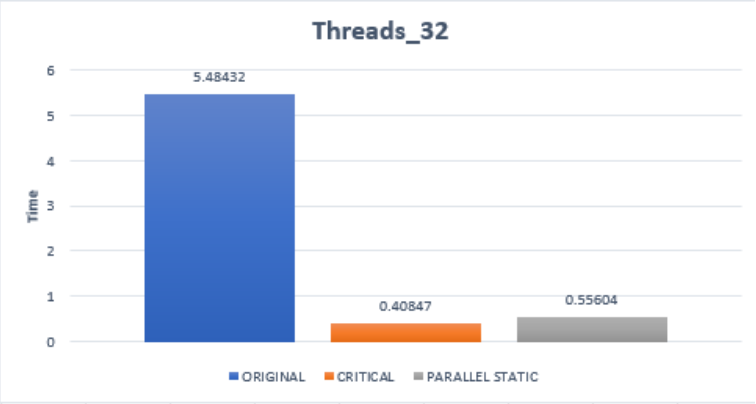

Figure 24: Thirty-two Threads are Running

5.5.6 Threads: 64

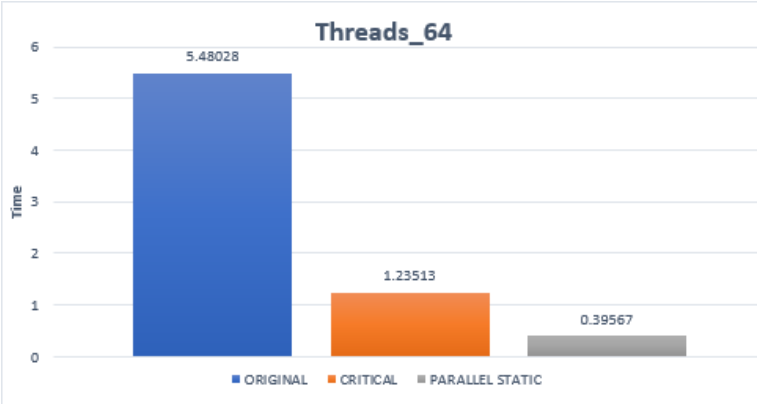

Figure 25: Sixty-four Threads are Running

6 Final Verdict

Parallel execution without a critical section produces better results when more threads are available. This can be observed from the 64 thread graph. Static scheduling is used since dynamic and guided scheduling produce inconsistent results (the standard deviation of time is a lot larger).