National Technical University of Athens
Electrical and Computer Engineering
Parallel Processing Systems
Semester Assignment 2023-2024

2. Algorithm Parallelization and Optimization in Shared Memory Architectures

Team : parlab03
Ioannis Palaios - el18184

I. Introduction

We were assigned to develop two parallel versions of the K-means clustering algorithm and one for the Floyd-Warshall algorithm assuming a shared memory architecture with OpenMP and given serial implementations of the two algorithms written in C. The goal is to measure the performance of the parallel versions of the two algorithm using 1, 2, 4, 8, 16, 32, 64 cores of one single node (on the lab's 'Serial' cluster called 'Sandman').

II. K means clustering- Shared Clusters Version

First of all, we have to benchmark the given serial implementation of the K-Means clustering algorithm. Its execution time running in 1 thread on the 'Serial' cluster for the configurations:

- {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} was : 7.70s (0.77s for each loop)
- {Size, Coords, Clusters, Loops} = {256, 1, 4, 10} was : 4.97s (0.49s for each loop)

It is worth noting the serial execution time in order to later compare it with the parallel versions and also extracting its results in order to check the validity and correctness of those parallel versions. Then, the next step is to parallelize the given 'Shared Clusters' version of the K-Means Clustering algorithm. Our first task, is to detect the region that can (and should) be parallelized. Generally, we inspected the code of the file: 'omp_naive_kmeans.c'. It is obvious to detect that the compute intensive part of the algorithm is into the do {} while() which essentially executes for either the specified loops (10 in our every configuration tried) or if the variable delta is over a threshold (convergence criterion). But, in every loop of the do{} while() there are major dependencies namely on the clusters array, which is used in the next loop from the function find_nearest_cluster(). So, without major algorithmic changes in our given implementation, it is impossible to parallelize the do{} while(). So, the next place where significant workload distribution can occur is inside each do{} while() loop (which from now on we will just call it 'loop'), and especially in the for loop which traverses the each object and does precisely 'numObjs' iterations which for the configuration Size, Coords, Clusters, Loops = 256, 16, 16, 10 is equal to 2097152. We observe that in each iteration on the set of objects, there is enough compute intensity hidden in the find_nearest_cluster() function, so we expect significant performance benefits from its parallelization. The code of the for loop traversing the each object is show in Listing 1.

```
1  for (i=0; i<numObjs; i++) {
2          // find the array index of nearest cluster
       center
3          index = find_nearest_cluster(numClusters,
       numCoords, &objects[i*numCoords], clusters);
4
5          // if membership changes, increase delta by
       1
6          if (membership[i] != index)
7              delta += 1.0;
8
9          // assign the membership to object i
10         membership[i] = index;
11
12         // update new cluster centers : sum of
       objects located within
13         newClusterSize[index]++;
14         for (j=0; j<numCoords; j++)
15             newClusters[index*numCoords + j] +=
       objects[i*numCoords + j];
16     }
```

Listing 1. K-Means Clustering compute intensive part of the given serial implementation in C.

But, as we naively split the workload to different threads we observe several ways where the result of a naive parallel can be false. This happens due to the race conditions occurring when incrementing the variables 'delta' (line 7), 'newClusterSize[index]' (line 13) and 'newClusters[index*numCoords + j]'(line 15). As a result, we should somehow restrict the access on each of these variables or create private copies and reduce them of the parallel region. But from the assignment we were (subtly) instructed to have the arrays newClusterSize and newClusters shared and to not reduce them at the end of the parallel region. So, in order to prevent many threads accessing and modifying those two arrays using the OpenMP API, we can choose between the *atomic* and *critical* keywords for the operations on these two arrays. The difference between those two keywords is that the *critical* keyword creates a critical section in any arbitrary block of code, while the *atomic* keyword specifies that only the operation following it must be atomic. When the former keyword is used, significant overhead is incurred due to the generality of the critical section (being able to surround any operation), but when the latter keyword is used we pay significantly less overhead but it is not available for a variety of operations and any block of code. It is obvious that if the *atomic* keyword is available for an operation, it should be used instead of the *critical* keyword.

As a result of the above, for the newClusterSize[] simple increment by one it is obvious that the atomic keyword should be used with the update keyword. For the newClusters[] array update, it does not make sense to use the *critical* keyword surrounding all of the for loop (of the lines 14 and 15 - Listing 1) when the *atomic* keyword can just be used for the update operation of line 15 (in Listing 1). For the 'delta' variable, we chose to reduce it via the OpenMP's *reduce(+:delta)* clause, specifying that we just want to sum all of the private 'delta' variables of each thread, in order to have the correct delta in the end of the parallel region.

In order to avoid data sharing between the threads from variables that should be private to each thread, we assign as *private* the variables *i*, *j* and *index* which all are defined inside each iteration and should remain private to each thread. We observe that all of the other variables used, are only read and not written/updated inside the parallel region, so it makes sense to have them shared between the threads. As the OpenMP's default data sharing on the parallel clauses, are shared, we don't explicitly define them as shared.

```
1  #pragma omp parallel for \
2  num_threads(nthreads) \
3  private(index, i, j) \
4  schedule(static) reduction(+:delta)
5  for (i=0; i<numObjs; i++) {
6      // find the array index of nearest cluster
       center
7      index = find_nearest_cluster(numClusters,
       numCoords, &objects[i*numCoords], clusters);
8
9      if (membership[i] != index)
10         delta += 1.0;
11
12     membership[i] = index;
13
14     #pragma omp atomic update
15     newClusterSize[index]++;
16
17     for (j=0; j<numCoords; j++)
18         #pragma omp atomic update
19         newClusters[index*numCoords + j] += objects[
       i*numCoords + j];
20 }
```

Listing 2. K-Means Clustering Shared Clusters version parallel part in C using OpenMP's API.

We benchmark it in the *Serial* cluster using 1, 2, 4, 6, 8, 16, 32 or 64 threads. We were instructed test our version with and without CPU affinity. CPU affinity is essentially binding each thread to a specific CPU core.

We have a variety of observations regarding the comparison of the execution times. First of all, without CPU affinity, we observe that in all of the parallel executions we cannot achieve a speedup in comparison to the Sequential program. We achieve a bigger execution times which effectively means that the parallelism slows down our implementation. This is mainly due to the accesses of the arrays *newClusters* and *newClusterSize* which happen atomically and essentially each thread waits for their turn to execute each atomic operation on these arrays. Also, this slowdown happens because of the memory accesses and data movement that happens throughout all of the NUMA system.

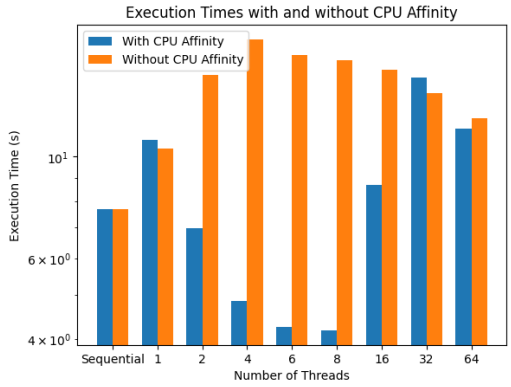

Figura 1. Bar Plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation in logarithmic scale.

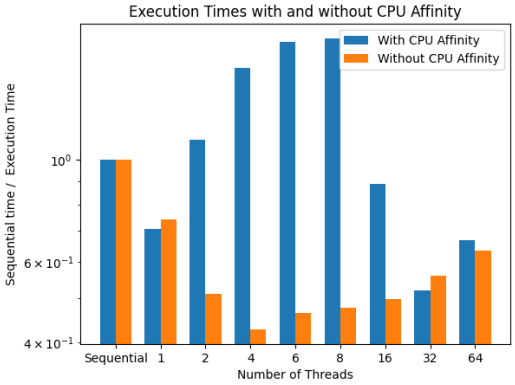

Figura 2. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation in logarithmic scale.

Enabling CPU affinity, what essentially happens is that the threads are bound to one CPU core throughout all of their execution and thus taking advantage of the fact that remnants of a thread that was run on a given processor may remain in that processor's state (in our case the data in the NUMA node's caches and RAM). Thus, data locality is more efficient and degrading effects like cache misses are avoided to a degree.

Regarding the number of threads used when CPU affinity is enabled, from Figures 1 & 2, it is obvious that the execution times drop significantly as we reduce threads for 1 all the way to 8 (including 2,4 and 6). This happens because all the threads stay in the same NUMA node of the Serial Cluster and thus they have better locality to the data they access due to accessing the same L3 cache & RAM. On the contrary, when the thread number increases to 16, 32 and 64, we notice an increase in the execution times despite the increase in processing units because the Operating System places the threads into 2 or more NUMA nodes and thus the data is split between them, making some accesses very time consuming.

III. K Means Clustering - Copied Clusters and Reduce Version

As mentioned before, the reason we had a significant decrease in performance (and increase in execution times) of the parallel version, is the sharing of the two arrays newClusters and newClusterSize between the threads and moderating the updates on these two array via the *atomic* keywords introduced a new overhead for our parallel application. In order to solve this issue, we decided to take another approach in the sharing of these two arrays. We observed that these two arrays have to have the updated value at the end of the parallel region with no dependencies between the threads (if executed correctly). So, as we were assigned, these two arrays went from One-Dimensional to Two-Dimensional having an extra field for the thread number. So if each thread accessed only one row of the Two-Dimensional array which essentially belongs to the thread, then, at the end of the execution we could just reduce these two 2D arrays into 2 1D arrays, summing up all the values for the elements belonging in the same column.

```
// initialize newClusterSize and newClusters to all 0
newClusterSize = (typeof(newClusterSize)) calloc( numClusters, sizeof(*newClusterSize));
newClusters = (typeof(newClusters))  calloc( numClusters * numCoords, sizeof(*newClusters));

int * local_newClusterSize[nthreads];  // [nthreads][numClusters]
double * local_newClusters[nthreads];   // [nthreads][numClusters][numCoords]

// Initialize local (per-thread) arrays (and later collect result on global arrays)
for (k=0; k<nthreads; k++)
{
    local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters, sizeof(**local_newClusterSize));
    local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords, sizeof(**local_newClusters));
}

timing = wtime();
 do {
    // before each loop, set cluster data to 0
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++)
            newClusters[i*numCoords + j] = 0.0;
        newClusterSize[i] = 0;
    }

    delta = 0.0;
    #pragma omp parallel num_threads(nthreads) \
    private(i, j, k)
    {
        for (i=0; i<numClusters; i++) {
                int tid = omp_get_thread_num();
                for (j=0; j<numCoords; j++)
                        local_newClusters[tid][i*numCoords + j] = 0.0;
                local_newClusterSize[tid][i] = 0;
        }
    }
    #pragma omp parallel for num_threads(nthreads) \
    private(i, j, index) schedule(static, 1) reduction(+:delta)
    for (i=0; i<numObjs; i++)
    {
        int tid = omp_get_thread_num();
        //find the array index of nearest cluster center
            index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);

        //if membership changes, increase delta by 1
        if (membership[i] != index)
            delta += 1.0;

        // assign the membership to object i
        membership[i] = index;

        // update new cluster centers : sum of all objects located within (average will be performed later)
        local_newClusterSize[tid][index]++;
        for (j=0; j<numCoords; j++)
            local_newClusters[tid][index*numCoords + j] += objects[i*numCoords + j];

    }
    for(k = 0; k < nthreads; k++) {
            for(i=0; i < numClusters; i++) {
                    for(j=0; j < numCoords; j++) {
                            newClusters[i*numCoords+j] += local_newClusters[k][i*numCoords+j];
                    }
                    newClusterSize[i] += local_newClusterSize[k][i];
            }
    }
    // average the sum and replace old cluster centers with newClusters
    for (i=0; i<numClusters; i++) {
        if (newClusterSize[i] > 0) {
            for (j=0; j<numCoords; j++) {
                clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
            }
        }
    }
    // Get fraction of objects whose membership changed during this loop. This is used as a convergence criterion.
    delta /= numObjs;

    loop++;
    printf("\r\tcompleted loop %d", loop);
    fflush(stdout);
} while (delta > threshold && loop < loop_threshold);
```

Listing 3. K-Means Clustering Copied Clusters Reduce version parallel part in C using OpenMP's API.

The code for this problem is shown on Listing 2. We expect significant performance gains due to the minimization of the dependencies between the threads. The Bar & Speedup plots for the Parallelized Shared Clusters and Reduce version are shown in Figure 3 and 4. It is obvious that as the number of threads increase, the more the execution times get smaller due to the efficient work distribution with well handled dependencies between the threads. In this version, each thread in its execution does not wait the other threads to perform an operation, rather, proceeds independently and produces a result which is later reduced.

The most significant drops of execution time is observed when dropping from 1 thread to 2 threads and from 2 to 4 threads. This essentially happens, between the threads stay in the same NUMA node and benefit from more fast data sharing and accesses. But as we see on Figures 3 and 4, similar

decrease in execution times happen when we further increase the threads to 8, 16, 32 and 64 because of the processing increase and the efficient work distribution from the parallel program. But the decreases might be smaller than before due to operating on different NUMA cores and thus the data transfer between them cost more time-wise.

We also observe that the increase from 32 threads to 64, do not decrease the execution time significantly because of the Cluster we do the benchmarks on have only 32 physical cores but 64 virtual cores. This means that it makes use of hyper-threading, which in our case do not provide significant benefit because of each thread that get executed in each core does the same operations on the CPU which makes the use of hyper-threading useless. Actually, we expected to see some performance degrading due to the hyper-threading's overhead, but it actually did not happen.

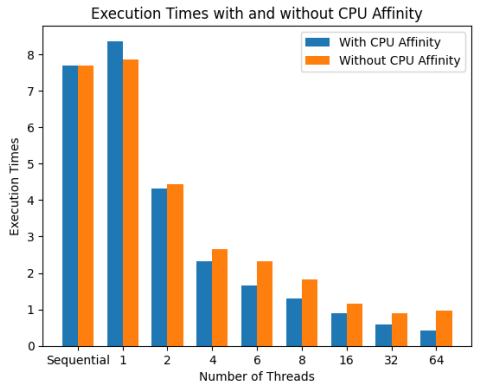

Figura 3. Bar plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation, in linear scale.

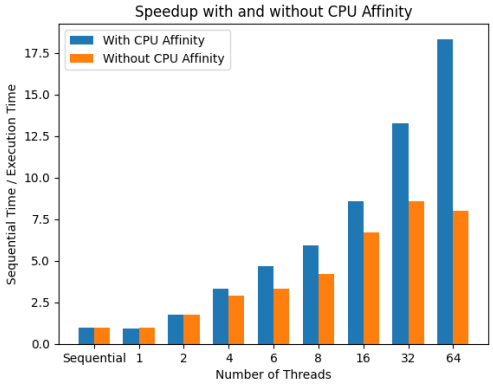

Figura 4. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation, in linear scale.

IV. FALSE SHARING IN THE SHARED CLUSTERS AND REDUCE VERSION

We previously benchmarked the parallel Shared Clusters & Reduce version for the configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}. We notice significant performance benefits in comparison to the serial implementation of K-Means Clustering. But, as expected, when we execute this version for the configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10} we notice a weird phenomenon with the execution times.

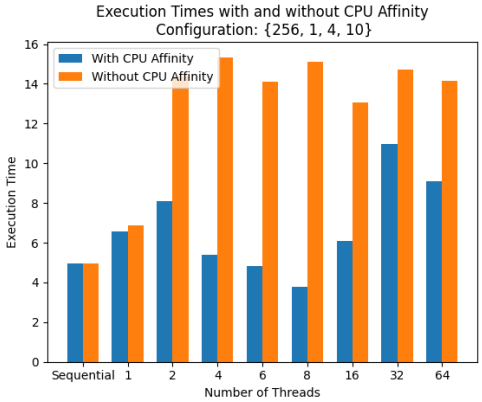

Figura 5. Bar Plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}.

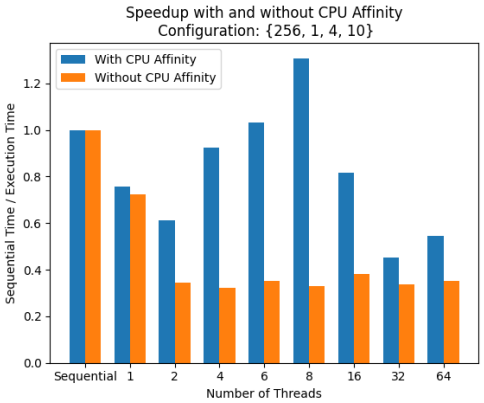

Figura 6. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation for the configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}.

As we see on Figures 5 and 6, without CPU affinity, the execution times get significantly larger than the serial implementation with no performance benefit whatsoever, for any number of threads. With CPU affinity, we notice a performance benefit when using 4,6 and 8 threads which as we mentioned before are placed in the same NUMA node and the data locality is better. The above results leads us to wonder about the reasons of not observing more significant performance benefits for this exact configuration, which is faster than the previous configuration in the Sequential execution.

From code inspection (and the hints in our assignment), we are lead to believe that for this configuration false sharing between the threads is taking place. In order to make sure that false sharing is indeed happening we execute the command *perf c2c* (on another machine). We indeed see that the number

of Hits in cache memory that are modified (and invalidated) are super high.

False sharing is happening when two (or more) threads share a cache line in their caches and when one thread writes, then, all of the other same cache line that exist in the other cores get marked as modified and for the sake of cache coherency, when the threads in the other cores try to read/write in this cache line, then this line has to be loaded from memory again. This process is time consuming and degrades the performance of our implementation for the above configuration.

But why only on this configuration and not not on the previous? The cache line in the 'Serial' cluster consists of 64 bytes. In the previous configuration, we had 16 Coords and 16 Clusters and now we have 1 and 4 respectively. In the arrays local_newClusterSize[number_of_threads]' and *local_newCluster[number_of_threads]* that get allocated in continuous locations in memory, each row has size equal to 4*Clusters bytes and 8*Coords*Clusters bytes respectively. The 4 and 8 stem from the size of integer and double data types in C language. In order to avoid false sharing, both of these sizes have to be over 64 bytes (and aligned or multiple of 64 ?). In the previous configuration, the row sizes are 64 Bytes and 16*64 Bytes, both over and multiples of 64 and that's why it is impossible for one cache line to contain two rows of either one of those two arrays.

But in current configuration, we observe that these rows have size 4 Bytes and 8 Bytes respectively, meaning that it is possible for more than two rows of these arrays to be contained in the same cache line. As a result, the performance of our implementation suffers heavily from the degradation effects of false sharing.

In order to solve the issue we have with false sharing, we dynamically allocate the memory needed for each row in a parallel region, where each thread allocates the memory of the row with its thread id. This makes use of Linux's first touch policy for NUMA systems and allocates the memory for each thread's row closest to this specific thread, Specifically, we change the lines 9-13 of Listing 3 to those shown on Listing 4 below.

```
1 #pragma omp parallel num_threads(nthreads)
2 {
3     int k = omp_get_thread_num();
4     local_newClusterSize[k] = (typeof(*
      local_newClusterSize)) calloc(numClusters,
      sizeof(**local_newClusterSize));
5     local_newClusters[k] = (typeof(*
      local_newClusters)) calloc(numClusters *
      numCoords, sizeof(**local_newClusters));
6 }
```

Listing 4. Array Definition in order to avoid false sharing (instead of the lines 9-13 of Listing 3).

The performance improvements are visible on figures 7 and 8. We observe the most significant performance improvement, once again, for 8 threads, which provides the best balance for this implementation regarding the increase in computation and in memory access latency. But, performance improvements are also visible for the other number of cores but not as significant. Once again, we observe the phenomenon of execution times increasing when going from 32 to 64 threads which, as we mentioned before, is likely due to the hyper-threading overhead.

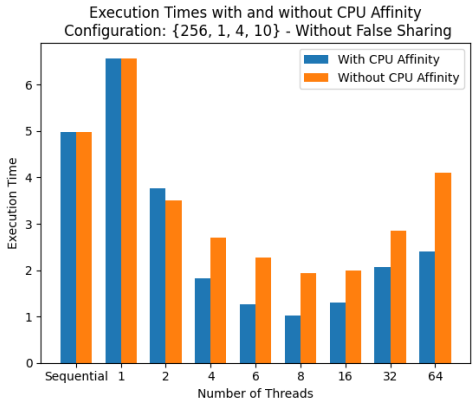

Figura 7. Bar Plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10} when we adapted the code in order to avoid false sharing.

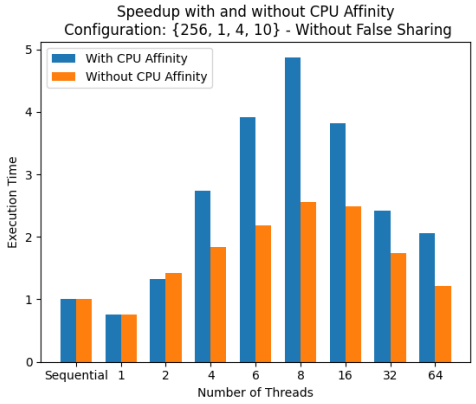

Figura 8. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation when we adapted the code in order to avoid false sharing.

V. Parallelizing the Floyd-Warshall Algorithm

Lastly, we were assigned to parallelize the Recursive version of Floyd-Warshall Algorithm. The Floyd-Warshall Algorithm is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights. The serial-iterative implementation of FW is memory bound and thus in order to parallelize it effectively we need to search for other implementations. As a result, in order to improve the cache performance of the algorithm two alternative implementations of FW have been proposed the first based on recursion and the second based on tiling.

We were assigned to parallelize the Recursive Floyd-Warshall Algorithm using OpenMP threads. The pseudocode of this implementation is shown in Listing 5. We observe that it is not a for loop in which we would simply attach a *pragma omp parallel for* and simply let OpenMP do the work scheduling. In this implementation, we have to somehow

dynamically schedule our work between the available threads, so each thread can take one recursive function call at each time. We can do this with OpenMP tasks which are ideal for dynamically scheduling work between threads.

```
1  FWR (A, B, C)
2      if (base case)
3          FWI (A, B, C)
4      else
5          FWR (A00, B00, C00);
6          FWR (A01, B00, C01);
7          FWR (A10, B10, C00);
8          FWR (A11, B10, C01);
9          FWR (A11, B10, C01);
10         FWR (A10, B10, C00);
11         FWR (A01, B00, C01);
12         FWR (A00, B00, C00);
```

Listing 5. Recursive Floyd-Warshall implementation in pseudocode.

But first, we need to understand the algorithm in order to find the dependencies between each recursive call. The algorithm is called as *FWR(A, A, A)* and the subsequent recursive calls pass quadrants of the input arguments. It is possible for each of A, B and C to point to different submatrices of the initial matrix. Due to the nature of the algorithm, it is only possible for the recursive calls of diagonal submatrices to be computed in parallel. But due to the way we define only the A matrix in the base case, the condition with the diagonal submatrices has to be true only for the first of the three parameters of the recursive function *FWR(A, B, C)*.

As a result, the only tasks that can be computed in parallel are the recursive function calls of lines (6, 7) and separately (10, 11). The parallelized subsection of code inside the else condition is shown on Listing 6.

```
1  #pragma omp parallel num_threads(nthreads)
2  {
3      #pragma omp single
4      {
5          FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol
       , myN/2, bsize);
6          #pragma omp task
7          {
8              FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,
       crow, ccol+myN/2, myN/2, bsize);
9          }
10         #pragma omp task
11         {
12             FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
       bcol,C,crow, ccol, myN/2, bsize);
13         }
14         #pragma omp taskwait
15
16         FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2,
        bcol,C,crow, ccol+myN/2, myN/2, bsize);
17         FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2,
        bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2,
       bsize);
18         #pragma omp task
19         {
20             FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
       bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
21         }
22         #pragma omp task
23         {
24             FW_SR(A,arow, acol+myN/2,B,brow, bcol+
       myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
25         }
26         #pragma omp taskwait
27         FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow
       +myN/2, ccol, myN/2, bsize);
28     }
29 }
```

Listing 6. Recursive Floyd-Warshall implementation in pseudocode.

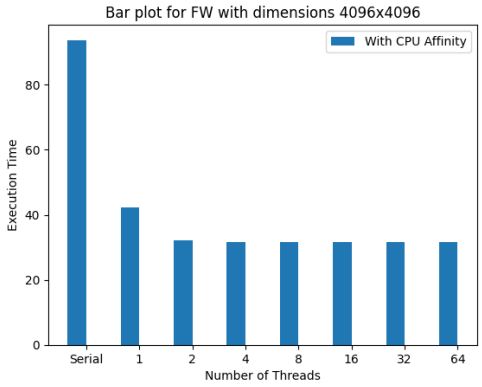

Figura 9. Bar Plot of the execution times for the parallel Recursive FW algorithm for the 4096x4096 dimensions.

In Figure 9, we observe that we have a performance increase when going from the iterative implementation to the recursive, as it is optimized for cache performance. But as we paralellize the recursive implementation we notice significant benefits when going from 1 to 2 threads, but little to negligible performance benefits when going from 2 to more threads. This is likely due to not having enough compute intensive tasks to assign to these extra threads to overcome the overhead of thread management and the memory and thus our implementation cannot get benefits from the extra threads. The same trends of the execution times per thread, continue to be true for the smaller board sizes.

The optimal block size happens to be equal to 64, as it turned out via our experiments. This is likely due to having the entire array needed for computation in each cores L2 caches which is actually 256KB. This happens due to the base case bringing chunks that fit entirely into the L2 cache (there are actually 4 of these chunks). The total memory that is brought in the base case is 64^3 integers * 4 (bytes/integer) = 4*1KB.

VI. CONCLUSION

In this assignment, we effectively parallelized two popular algorithms : K-Means Clustering and the Floyd-Warshall Algorithm with various ways and OpenMP concepts. We saw the performance implications that can occur when we do not take into account the hardware and the Operating System in which our implementation is executed. We also explored various ways the same algorithm can be parallelized while visualizing the results and explaining them. Finally, we explored OpenMP tasks and how this OpenMP concept can be efficiently used to dynamically schedule work to threads in a parallel program.

REFERENCIAS

[1] OpenMP Application Program Interface Version 4.0 - July 2013 https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[2] OpenMP,
http://openmp.org.
[3] J.S. Park, M. Penner, and V. K. Prasanna, Optimizing Graph Algorithms for Improved Cache Performance IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 9, SEPTEMBER 2004.
]