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.

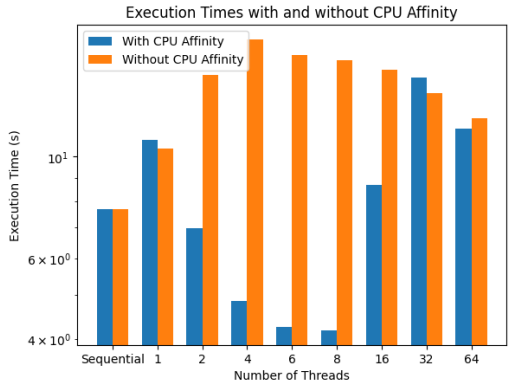

Figure 1. Bar Plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation in logarithmic scale.

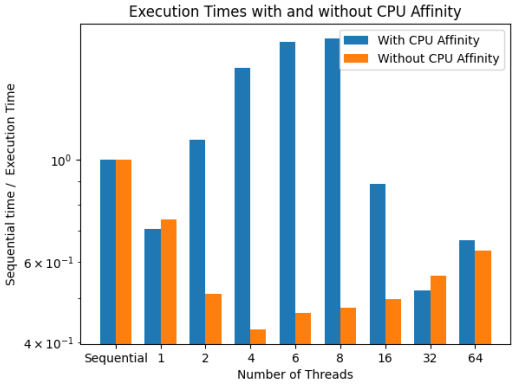

Figure 2. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation in logarithmic scale.

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.

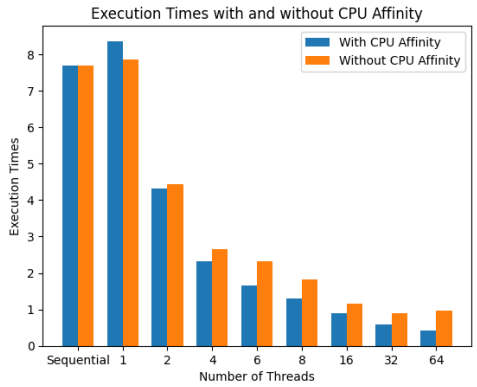

Figure 3. Bar plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation, in linear scale.

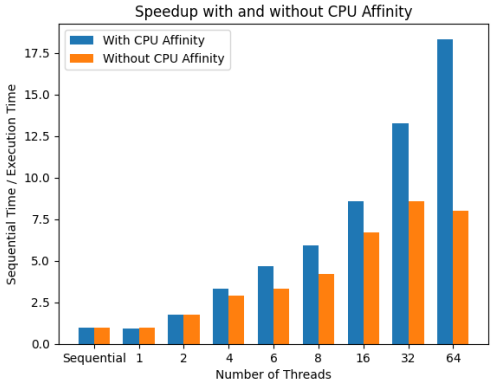

Figure 4. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation, in linear scale.

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.

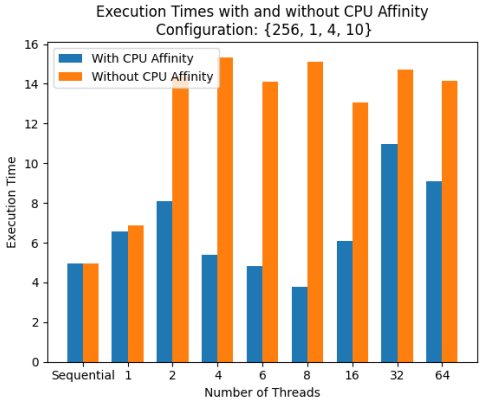

Figure 5. Bar Plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}.

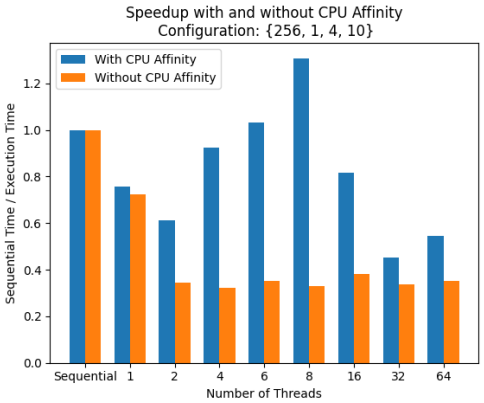

Figure 6. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation for the configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}.

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.

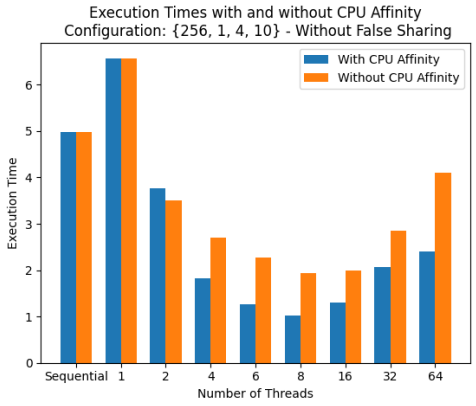

Figure 7. Bar Plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10} when we adapted the code in order to avoid false sharing.

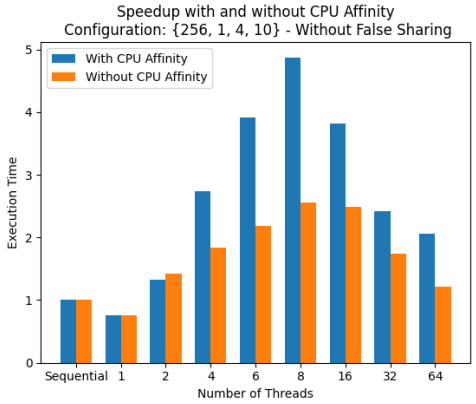

Figure 8. Speedup plot of the execution times of the Shared Clusters parallel version of the K-Means Clustering Algorithm OpenMP implementation when we adapted the code in order to avoid false sharing.

V. Parallelizing K-means clustering using locks

We were assigned to parallelize the K-means Clustering algorithm using Mutual Exclusion and Locks. First of all, we have to identify the locks we are going to apply to our parallel code in order to understand the benefits and drawbacks while also analyzing the given implementation. Our given implementation essentially declares in *lock.h* the available struct called *lock_t*, while it also declares the initializer called *lock_init* and three functions called *lock_free*, *lock_acquire* and *lock_release*. This leaves the implementation dependent on which .c file we use for the linking in the Makefile. So, essentially, the implementation happens inside all the .c files which implement the locks in the
locks directory and as result we have to code inspect those files in order to understand each locks' implementation. Namely, the available locks are :

- **nosync_lock** : This lock effectively does nothing. All of the above functions declared in *lock.h* are defined but essentially the do nothing. This, of course, will lead the parallel program that uses this lock produce incorrect results, as race conditions arise when incrementing the arrays *newClusterSize[]* and *newClusters[]*.
- **pthreads_mutex_lock** : This lock essentially uses the Linux implementation of Pthreads from its POSIX compliant Thread API. When a thread requests a lock and its held by another thread, then it is 'put to sleep'(suspends its execution) and is 'woken up again' (resumes its execution) when the lock is release by the other thread. This pause and continue of the threads execution costs some CPU cycles but, ultimately, it can be worth it, if the time that the lock is held by the other thread(s) is bigger than the time it takes to put to sleep and wake up the thread that requests the lock.
- **pthreads_spin_lock** : Like above, this lock uses the Linux implementation of Pthreads spin lock from its POSIX API. The difference with the above lock, is that if a thread request a lock already held from another lock, then it does not suspend its execution but, instead, resumes its execution until the lock is release by the other thread(s). This can potentially be worth it in comparison to the above lock implementation, if the time interval that each thread holds the lock is smaller than time interval need to pause and resume one threads execution (or some other constant close to that).
- **tas_lock** : This lock implementation makes use of Intel's builtin functions of *__sync_lock_test_and_set* and *__sync_lock_release* which are atomic exchange operations (with an acquired memory barrier). The former function is used for the declared *lock_acquire* function, and guarantees atomicity between a read of a variable and a write in the variable. In the implementation, we read from the state variable (that is of type : *enum lock_state_t*={*UNLOCKED, LOCKED*} of the lock struct that is stored in memory and we assign to it the enum value *LOCKED* while, reading its current value. If its current state value is of type *UNLOCKED* then this means that the lock was not previously held from another thread and it is safe to acquire the lock. If its current state value is of type *LOCKED* this means that it is held by another thread and we do not proceed but continue to ask for the lock. In both cases, the value *LOCKED* is written on the variable and we proceed accordingly. The *__sync_lock_release* has a similar with above implementation but it does not read the variable but rather sets the value 0 to it (which equals to *UNLOCKED*).
- **ttas_lock** : This lock implementation is very similar with the above *tas_lock* implementation, but with a significant optimization. While the lock is held by another thread, it does not read, each time, atomically the state of the lock variable but rather just reads it with a simple non atomic read. Eventually, if one non atomic read provide us with the value of *UNLOCKED* the it proceeds with the function *__sync_lock_test_and_set* as the above implementation, and if this function fails to read the value *UNLOCKED* it returns to the while with the non atomic reads. In this implementation, we expect to see an increase in performance as we decrease resource contention caused by bus locking and we avoid the overhead of the cache coherence protocol on contended locks.
- **array_lock** : As the name implies this is an Array-based queue lock. Essentially, the locking and queuing process is that each thread that requests the lock, takes a slot in the *flag* array is assigned in this array, and does not acquire the lock (and spins) until the the value in this slot becomes true (*flag[mySlot]=TRUE*. The slot is represented as a thread-local integer variable which represents the position of each thread in the queue called *mySlot*. We know what is the next slot to give to the next thread that requests the lock through the tail long variable which is shared between the threads and as we give locks to threads, we increment it by one *FLAG_ENTRY_SIZE*. We start with the initial values for *tail=0* and *flag[0]=TRUE* (and any other value in the flag array equal to false) so the first thread that requests the lock, acquires it. The release of the lock set the *flag[mySlot]=FALSE* and set the next value in the flag array equal to true. We expect the performance of this implementation to be better than above, as while one thread is waiting for the lock to be acquired, it spins on its locally cached copy of *flag[mySlot]* which is not getting invalidated/modified my other threads (only when set true by the previous slot) and thus contention and cache invalidation traffic are greatly reduced. It has to be noted that false sharing, which can be a significant performance problem in this implementation, is successfully avoided. This lock also provides first-lock-first server fairness.
- **clh_lock** : It is very similar with the *array_lock* but instead of a whole array of size equal to the number of threads for one lock, we now define a struct called *clh_node_t* which has an attribute called *locked* (TRUE if its waiting for the lock or having it and False for not having the lock) and for each thread we define two of this structs : one for the current thread and one for the previous. When one thread tries to acquire this lock, it inserts its struct for the current thread as the next pointer of the tail and sets the previous one as the one that was previously the next one of the tail. Of course, the threads waiting to acquire the lock, spin on a local copy of the *locked* variable of the previous node pointer, which has minimal cache coherence traffic. The lock release does not do anything special - it just sets the locked variable of the *myNode* struct to FALSE and make it point to the previous one. The disadvantage of this algorithm is that it might perform poorly on architectures with no cache coherency in the case that the memory location of the attribute *locked* of the previous node is remote.

The difference in performance between **pthreads_mutex_lock** and **pthreads_spin_lock** will be due to the difference in the overhead of the pausing and

resuming the execution of a thread versus the overhead of waiting in a spin (loop) while repeatedly checking whether the lock is available (with the cache coherency traffic). As a result, the time each thread holds a lock will play a significant role in this comparison.

We obviously expect the **ttas_lock** to have better performance than **tas_lock** as it does not continuously atomically read a variable and as a result has lower resource contention caused by bus locking and especially, the cache coherency protocol overhead on the contended locks.

Both the *array_lock* and *clh_lock* guarantee fairness in lock acquisition by using a FIFO queue-based mechanism which does not happen in any of the other lock mechanisms we described above. *The clh_lock approach has significantly smaller memory usage than the array_lock*, since for L locks and N threads (and each thread accessing at most one lock at a time), the *array_lock* uses O(L*N) space (due to creating arrays of size N for each Lock) but the *clh_lock* uses O(L+N) space because we just create N number of structs with the appropriate pointers for each Lock. We do not expect any significant performance differences in terms of execution times between these two implementations.

```
lock_t *lock;
lock = lock_init(nthreads);
            :
            :
lock_acquire(lock);
newClusterSize[index]++;
for (j=0; j<numCoords; j++){;
   newClusters[index*numCoords + j] += objects[i*
   numCoords + j];
}
lock_release(lock);
```

Listing 5. Part of the given program accessed by locks - Implentation with one lock.

We first benchmark the implementation that has locked the critical section as shown in Listing 5 where the lock_t type can be any of the locks above. We benchmarked it for 1,2,4,8,16,32,64 number of threads but the numbers of threads with the biggest interest are 8, 16 and 32. The results are shown in Figures 9,10 and 11 respectively.

We observe that the locking mechanism which has the best performance out of all the locks (excluding nosync_lock of course) are the **Queue-based Locks**. This mostly happens due to the reasons mentioned above like the optimization of having each thread spin on a variable that is not getting invalidated constantly and thus reducing cache-coherence traffic. The small performance difference between *array_lock* and *clh_lock* is likely due to the bigger memory usage and initialization in the former case and small differences in their implementations.

Secondly, the difference in performance between the *ttas_lock* and *tas_lock*, is clearly visible as we expected. This is due to the reasons we mentioned above. The *tas_lock* has clearly the worst performance out of all the implementations, which is evidence to how expensive having a thread spin on a variable and reading it with atomic operations, actually is.

Also, we observe that for 8 or 16 threads the *pthreads_spin_lock* is faster than the *pthreads_mutex_lock* which changes for 32 threads. This happens that when we

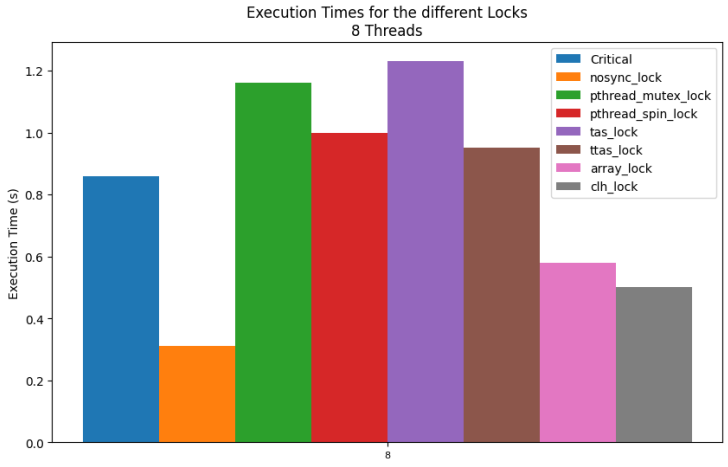

Figure 9. Execution time for the given implementation with locks for 8 threads.

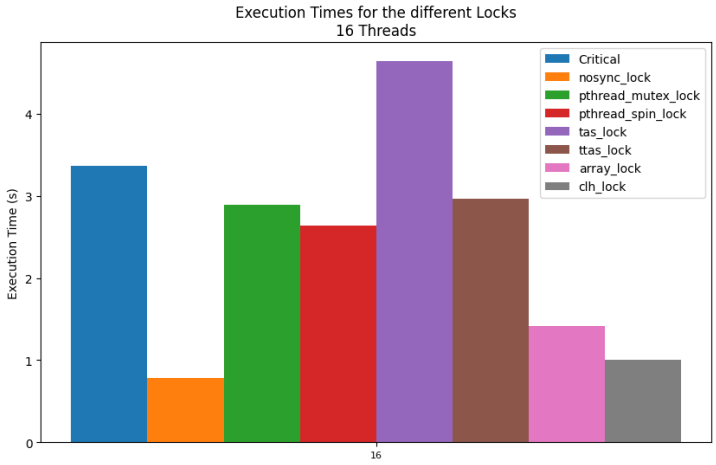

Figure 10. Execution time for the given implementation with locks for 16 threads.

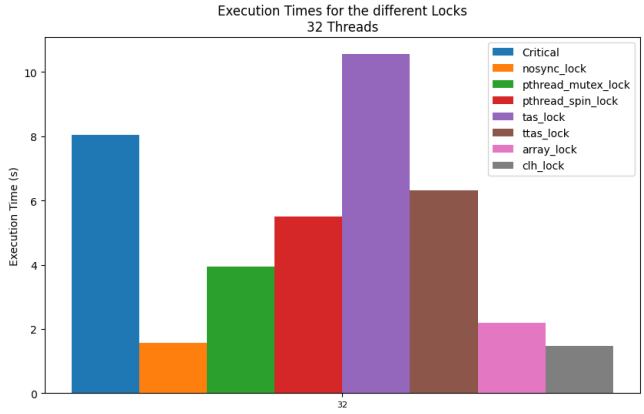

Figure 11. Execution time for the given implementation with locks for 32 threads.

have more threads, then the waiting time for one thread is bigger and thus, it is a better choice to pause and resume a thread rather than keep it alive and have it do busy-waiting. As a result, the more we increase the number of threads, the more the mutex lock mechanism does better than the spin lock. But for number of threads less or equal than 16 the *pthreads_spin_lock* is a clear winner (in comparison to the mutex lock).

Lastly, we observe that the implementation that uses OpenMP's critical clause, does not perform particularly well in comparison to the other locks, especially as we increase the threads. For 8 threads, it actually performs better in comparison to most of the other lock implementations but for 16 or 32 threads it is the second worst. This is likely due to the generality of its implementation or the not optimized OpenMP's implementation.

We are highly optimistic that the performance we achieved with these locks can be increase if instead of coarse-grained locking we can somehow achieve finer-grained locking. Through code inspection, we understand that if we somehow provided more locks for specific parts of the for, the thread waiting can be minimized.

VI. 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.

```
FWR (A, B, C)
    if (base case)
        FWI (A, B, C)
    else
        FWR (A00, B00, C00);
        FWR (A01, B00, C01);
        FWR (A10, B10, C00);
        FWR (A11, B10, C01);
        FWR (A11, B10, C01);
        FWR (A10, B10, C00);
        FWR (A01, B00, C01);
        FWR (A00, B00, C00);
```

Listing 6. 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.

```
#pragma omp parallel num_threads(nthreads)
{
    #pragma omp single
    {
        FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol
    , myN/2, bsize);
        #pragma omp task
        {
            FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,
    crow, ccol+myN/2, myN/2, bsize);
        }
        #pragma omp task
        {
            FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
    bcol,C,crow, ccol, myN/2, bsize);
        }
        #pragma omp taskwait

        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2,
     bcol,C,crow, ccol+myN/2, myN/2, bsize);
        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);
        #pragma omp task
        {
            FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
    bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
        }
        #pragma omp task
        {
            FW_SR(A,arow, acol+myN/2,B,brow, bcol+
    myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
        }
        #pragma omp taskwait
        FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow
    +myN/2, ccol, myN/2, bsize);
    }
}
```

Listing 7. Recursive Floyd-Warshall implementation in pseudocode.

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

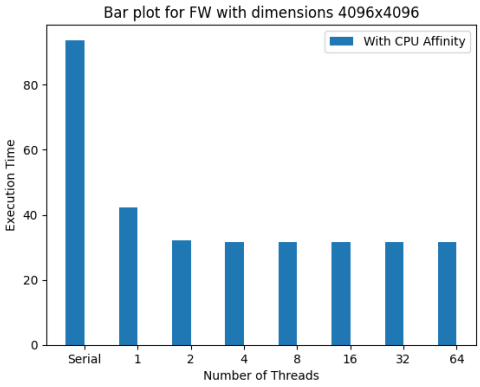

Figure 12. Bar Plot of the execution times for the parallel Recursive FW algorithm for the 4096x4096 dimensions.

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.

VII. Concurrent Data Structures

In this chapter, we were assigned to benchmark and compare 5 given concurrent linked list implementations and their corresponding method implementations : *add()*, *remove()* and *contains()*. The linked list method implementations are :

Coarse-grain locking, where one universal lock has to be acquired for each of those 3 methods. Obviously, this implementation might only have visible performance improvements in comparison to the others, when lock contention between the threads is rare.

Fine-grain locking, where locking happens on a node level. For each of the three methods, the list traversal starts from the head sentinel node and then moves serially to the next nodes. The locking happens in a "hand-over-hand" order, as each thread wanting to lock a node, has to have a lock to it previous node and one thread can have at most 2 locks at a time - the current and previous nodes. Progress is guaranteed because each thread starts from the head sentinel node and thus deadlocks are avoided.

Optimistic synchronization, which is an improvement over Fine-grain locking, where less lock acquisition takes place. The list traversal happens without acquiring locks and eventually those locks are only acquired when we found the right nodes where the node addition or removal would take place. But, performing the corresponding operation might be produce a false outcome if a synchronization conflict causes the wrong nodes to be locked. So after the node locking happens, a function *validate()* is run in order to make sure that these two nodes are still reachable in the linked list. The chances that the locked nodes are unreachable are low (for normal workloads), thus the name *optimistic*.

Lazy synchronization, which by adding a boolean field in each node indicating if it is removed or not, we can skip the re-traversal of the list for the validation and the *contains()* method does not obtain locks as happened in the above implementation (*Optimistic Synchronization*). The removal of a node is split into two phases : logical and physical. Logical removal means that only the boolean field is switched to true and physical removal means that a node is not referred by an unmarked reachable node. Logical removal happens before physical and it is important to note that physical removals can happen in a later time (and can be batched in a time where the linked list is being accessed by very few threads). There is room for improvement for this implementation, due to the disadvantage that the *add()* and *remove()* methods are blocking.

Non-blocking synchronization, which by using atomic operations can make all three methods (*add()*, *remove()* and *contains()* to be non-blocking - the first two can be lock-free and the third wait-free. Our implementation uses *__sync_val_compare_and_swap(3)* in order to check (with an atomic operation) if the node has the expected next field. We still use the marked boolean field in each node to check if a node is logically deleted or not (as in the above *Lazy synchronization* implementation) and the physical removal happens when searching for a node for all of our three methods (add(), remove() and contains()). As this synchronization method is the only non-blocking one mentioned so far, it is expected to have better performance except in the cases where we have contention among threads (forcing the threads to restart their traversal) because of possible concurrent cleanups of removed nodes.

As implied above, each implementation has its benefits and drawbacks that might influence performance :

- **Coarse-grain locking**'s benefits can only be seen when lock contention is rare (or practically non-existent). It is expected to have worse performance than any of the other implementations.
- **Fine-grain locking**'s benefit can be increased concurrency than the Coarse-grain locking synchronization method. Its main drawback is that it imposes a potentially long sequence of lock acquisitions and releases and threads accessing different part of the list can still block each other. We expect this method to have worse performance than any of the below implementations.
- **Optimistic synchronization**'s benefit is that it can traverse the array without obtaining locks. But its drawback is the call of the *validate()* function that re-traverses the list and repeats the whole process in the case it fails. So we expect the benchmarks containing a high number of removes to make this implementation to perform particularly worse.
- **Lazy synchronization**'s benefit is that it avoids the validation() process happening above and it will obviously faster than all of the above. But one of its main drawbacks is that the add() and remove() functions obtain locks and thus they are blocking and do not guarantee progress

in the case of delays. For these reasons, we expect it to perform worse as frequency of add() and remove() method calls increase, in comparison to the Non-blocking implementations.

- **Non-blocking synchronization**'s benefit is that it makes the add() and remove() methods non-blocking. We expect it to perform better than all of the implementations above for the reasons mentioned above.

We were asked to benchmark the linked list implementations for a number of threads : (1, 2, 4, 8, 16, 32, 64 and 128), list sizes (1024, 8192) and the percentage of operations contains()-add()-remove() (100% contains, 80% contains-10% additions-10% removals, 20% contains-40% additions-40% removals and 50% additions-50% removals). We will choose specific benchmarks to compare and comment in order to compare them efficiently. The figure that we will do the comparisons on is the throughput which essentially is the amount of operations that happen for each second (KOps/sec).

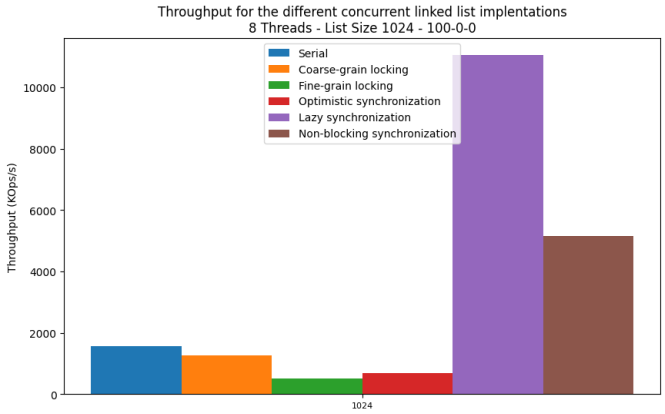

Figure 13. Bar Plot of the throughput for each of the above concurrent linked list implementations for linked list size of 1024 and for 100% of the operations being 'contains()' and with using 8 threads.

In figure 13, we benchmarked the above implementations using only the 'contains()' methods. The performance difference is visible. The Coarse-grain, Fine-grain implementations are performing 1.2x and 3x worse than the serial implementation respectively, which was expected and is due to each of their drawbacks mentioned above (lock contention etc.). Comparatively, Coarse-grain performs better than Fine-grain, likely due to the locking overhead, and the blocking of the threads while traversing the list for the contains() calls (in the fine-grained implementation).

The optimistic synchronization implementation performs also more than 2x worse than the serial, which is likely due to lock contention for nodes that the method contains looks for. We expect this performance degradation to be less worse for a bigger list size. Indeed, for a list of size 8192 (instead of 1024) this implementation performs the same as the serial implementation as contention for the contains() method is significantly less.

But, We also observe that the Non-blocking synchronization implementation is almost 2x worse than the lazy synchronization. This is likely due to 2 reasons. First, the validate() method of the lazy implementation never returns false (because only contains() methods are performed which do not change the list) and thus the linked list re-traversal does not ever need to happen, and as a result the lazy implementation can perform reasonably well in comparison to the other. Second, it is likely that the overhead of the atomic operations used in the Non-blocking implementation cause this performance degradation.

As we increase the thread number, some interesting things happen. First of all, we observe that the worst performers are the Fine grained and Optimistic synchronization. The best performer is clearly the Lazy synchronization implementation achieving a **29x speedup** from the serial implementation, for 32 threads.

Almost the same trends continue to be true for the benchmarks done for 80% contains() operations, and 10% add() and remove() each.

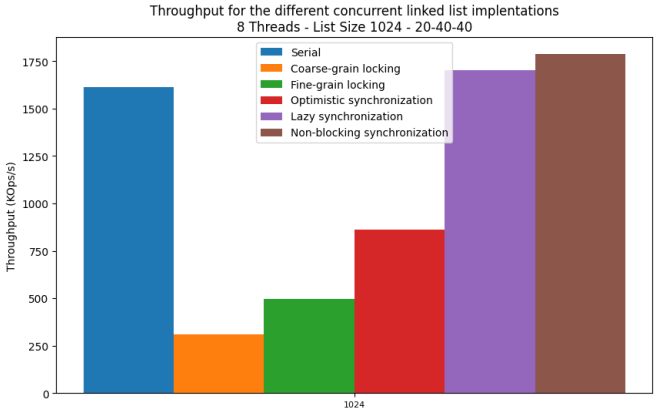

Figure 14. Bar Plot of the throughput for each of the above concurrent linked list implementations for linked list size of 1024 and for 20% of the operations being 'contains()', 40% being 'add()' and another 40% being 'remove()', with using 8 threads.

Now, in figure 14, with the same list size and 8 threads as in the figure 13, we observe the throughput of our implementations for 20% contains() calls, and 40% node additions and removals respectively. First of all, again, in comparison to the serial implementation, the coarse and fine grain implementations perform 5x and 3x worse respectively. There is obvious performance degradation in the case of Coarse-grain locking where in Figure 13, it performed 1.2x worse than serial, which is due to the increased lock contention happening due to the increased lock contention caused by more time requiring operations (like add() and remove()). The performance of the Lazy and Non-blocking implementations fell from 7x and 3x to almost the same as the serial(both of them).

For the lazy implementation, this happened because the validate() function returns false significantly more often than previously (because of more node removals) and this makes the corresponding thread to re-traverse the whole list. Benchmarking this implementation for the bigger list size (8192 instead of 1024) achieves 4x speedup from the serial, because conflicts cause the validate() method to fail are significantly more rare. A similar observation can be made for the Non-

blocking implementation, because there is the delay from the contentions of the atomic operations used by multiple threads on the same nodes (instead of the validate() failing). Those two implementation performed identically in this exact benchmark.

As the number of threads increase, we observe the optimal speedup with 32 threads (running on all of the physical cores) with the Non-blocking implementation with almost 3x speedup from the serial (while the lazy has 2.3x speedup). This is due to the reason that the atomic operations are better than the spin lock used by the lazy implementation which spins on a remote memory location (of another NUMA node) causing memory bus contention.

The same trends continue to be true for the benchmarks which did 50% addition operations and 50% remove operations.

As mentioned above, we did not plot the benchmarks done for 80-10-10 and 0-50-50 (contains()-add()-remove()). This was done for the reason that the plots did not differ substantially from the two above cases. We also did not plot for all the different number of threads but we instead commented on interesting differences happening with the increase and decrease of the threads.

VIII. 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 explored various ways the same algorithm can be parallelized while visualizing the results and explaining them. We discovered how different locks are implemented and their differences in performance. We explored OpenMP tasks and how this OpenMP concept can be efficiently used to dynamically schedule work to threads in a parallel program. Also, we researched different concurrent linked list implementations and then benchmarked them, visualizing and explaining the results.

REFERENCES

[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.

[4] The Art of Multiprocessor Programming by Herlihy, Maurice and Shavit, Nir