ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Parallel Processing Systems

Laboratory Exercises

Ioannis Rekkas
03119049

Anastasios Stefanos Anagnostou
03119051

Georgios Anastasiou
03119112

September 5, 2025

Contents

I Exercise 1 4

1 Data Collection 4

2 Comparison of Measurements 5

3 Interpretation of Results 5

4 Appendix 7

II Exercise 2 9

5 K-MEANS 9

5.1 shared clusters . . . 9

5.2 copied clusters and reduce . . . 10

6 Floyd-Warshall 13

7 Appendix 15

III Exercise 3 18

8 K-MEANS 18

8.1 Mutual Exclusion — Locks . . . 18

8.2 Concurrent Data Structures . . . 21

IV Exercise 4 24

9 Naive 24

10 Transpose 25

11 Shared 25

12 Bottleneck Analysis 27

12.1 2 Coordinates . . . 27

12.2 16 coordinates . . . 29

13 Bonus: Full Offload (All-GPU) version 31

14 Appendix 32

V Exercise 5 42

15 K-MEANS 42

15.1 MPI Implementation . . . 42

15.2 BONUS Question: Comparison with OpenMP . . . 42

16 2D Heat Diffusion **43**
16.1 Measurements with Convergence Check 43
16.2 Measurements without Convergence Check 43

17 Appendix **46**

Part I

Exercise 1

1 Data Collection

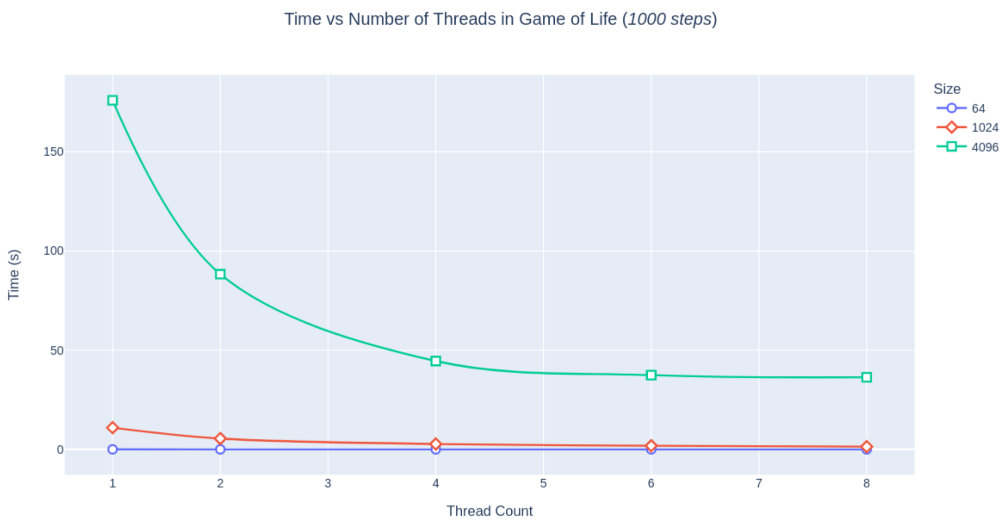
Initially, the program was parallelized using the appropriate OpenMP pragma, as shown in code 1 (see Appendix). For data collection, a compute node with eight cores was reserved and script 4 was executed. The executable was produced by compiling as shown in the Makefile 3. The output data are shown below.

```
Thread Count = 1
GameOfLife: Size 64 Steps 1000 Time 0.023253
GameOfLife: Size 1024 Steps 1000 Time 10.968962
GameOfLife: Size 4096 Steps 1000 Time 175.976354
Thread Count = 2
GameOfLife: Size 64 Steps 1000 Time 0.013578
GameOfLife: Size 1024 Steps 1000 Time 5.457094
GameOfLife: Size 4096 Steps 1000 Time 88.317703
Thread Count = 4
GameOfLife: Size 64 Steps 1000 Time 0.010049
GameOfLife: Size 1024 Steps 1000 Time 2.723685
GameOfLife: Size 4096 Steps 1000 Time 44.545233
Thread Count = 6
GameOfLife: Size 64 Steps 1000 Time 0.009147
GameOfLife: Size 1024 Steps 1000 Time 1.833331
GameOfLife: Size 4096 Steps 1000 Time 37.423192
Thread Count = 8
GameOfLife: Size 64 Steps 1000 Time 0.009746
GameOfLife: Size 1024 Steps 1000 Time 1.376557
GameOfLife: Size 4096 Steps 1000 Time 36.365997
```

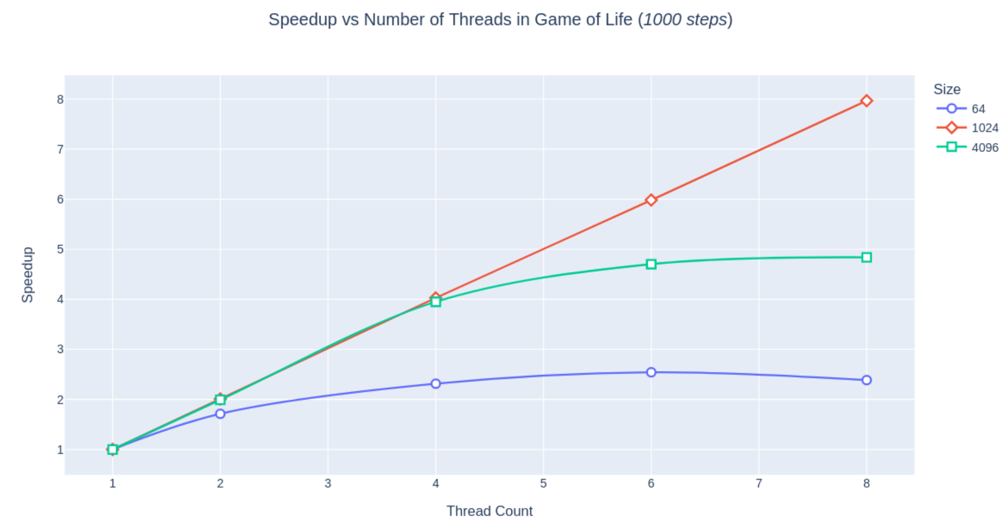
Figure 1: Output Data Game Of Life

2 Comparison of Measurements

The measurements are summarized in the following graphs.



(a) Execution time of Game Of Life versus the number of processes for various game sizes.



(b) Speedup of Game Of Life for various game sizes.

In Figure 2a, three curves are shown—one for each Game of Life simulation size—depicting runtime for various thread counts. Figure 2b shows three curves depicting the speedup for each simulation size.

3 Interpretation of Results

Both 64×64 and 4096×4096 do not scale.

For 64×64, execution is so fast that thread creation alone introduces enough overhead to prevent scaling. From 2 threads onward, scaling is non-ideal. There is speedup, just not as much as expected.

For 4096×4096, runtime is long enough that thread creation overhead is negligible. However, bus contention occurs because the data no longer fit in the cache, so threads request data from main memory more frequently. The shared bus serializes part of the program, forcing some threads to wait for others to obtain their data. This emerges from 6 threads and above. Up to 4 threads, scaling is almost ideal.

In contrast, at 1024×1024 we observe ideal linear speedup for all thread counts. Here, the parameters align so that there is no bus contention and the work distribution across threads is excellent.

4 Appendix

Listing 1: Parallelized Game Of Life

```c
...
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for private(nbrs, i, j) shared(previous, current)
    for ( i = 1 ; i < N-1 ; i++ )
        for ( j = 1 ; j < N-1 ; j++ ) {
            nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
                 + previous[i][j-1] + previous[i][j+1] \
                 + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
            if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
                current[i][j]=1;
            else
                current[i][j]=0;
        }
        #ifdef OUTPUT
        print_to_pgm(current, N, t+1);
        #endif
        //Swap current array with previous array
        swap=current;
        current=previous;
        previous=swap;
}
...
```

Listing 2: Bash Script to build Game Of Life

```bash
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_omp_gol

## Output and error files
#PBS -o make_omp_gol.out
#PBS -e make_omp_gol.err

## How many machines should we get?
#PBS -l nodes=1:ppn=1

##How long should the job run for?
#PBS -l walltime=00:01:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab30/a1
make
```

Listing 3: Makefile

```makefile
all: Game_Of_Life

Game_Of_Life: Game_Of_Life.c
        gcc -O3 -fopenmp -o Game_Of_Life Game_Of_Life.c

clean:
        rm Game_Of_Life
```

Listing 4: Bash Script to Gather Measurements

```bash
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_gol

## Output and error files
#PBS -o .run.out
#PBS -e .run.err

## How many machines should we get?
#PBS -l nodes=1:ppn=8

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab30/a1

threads=(1 2 4 6 8)
sizes=(64 1024 4096)
speed=1000

for thread in "${threads[@]}"
do
    export OMP_NUM_THREADS="$thread"
    for size in "${sizes[@]}"
    do
        ./Game_Of_Life "$size" "$speed"
    done
done
```

Part II
Exercise 2

5 K-MEANS

5.1 shared clusters

The code was parallelized by adding the appropriate OMP compiler directives at the parallelizable points of the program (see 5). The measurements are presented in Figure 3. It is observed that, in every case, regardless of thread count, the parallel version of the algorithm is slower than the serial version. This is because the attempted parallelization was naive. Specifically, all threads share the arrays with cluster data, which wastes a lot of compute time on synchronization. Notably, even for a single thread, the parallel version is slower. This is due to the use of atomic instructions for synchronization, which lock the bus and interfere with processor acceleration features. In the serial algorithm, atomic operations are absent because atomicity is inherent.

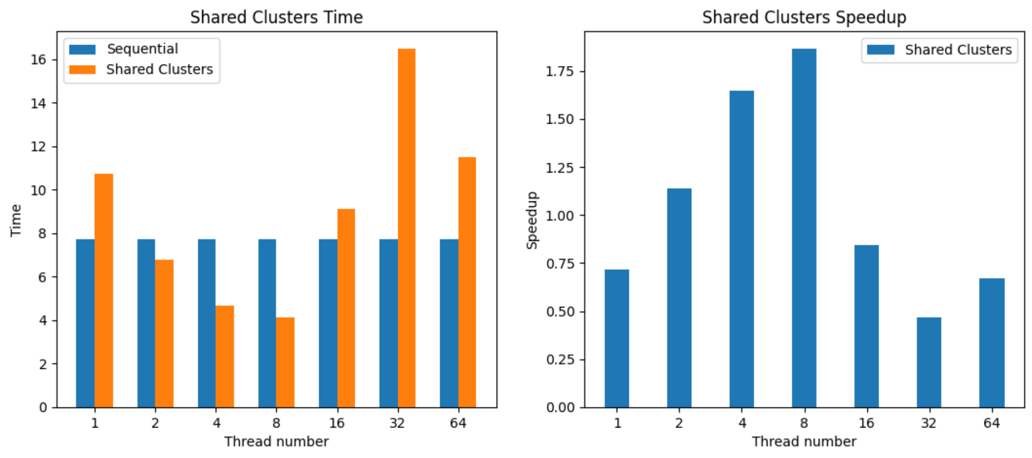


Figure 3: Execution Time and Speedup of Naive Kmeans Shared Clusters

If the environment variable GOMP_CPU_AFFINITY is set appropriately, threads remain pinned to a processor during their lifetime. This ensures they are not moved to another core and that they retain data locality.

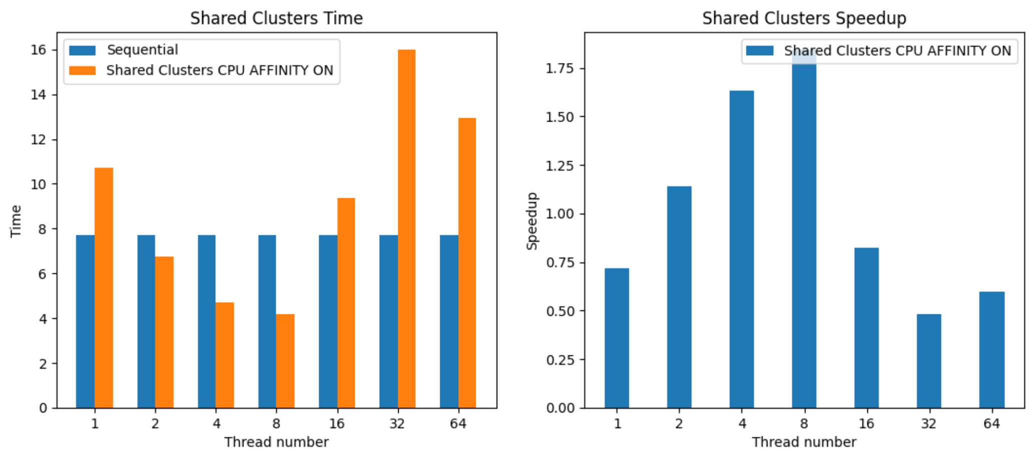


Figure 4: Speedup of Naive Kmeans Shared Clusters with CPU Affinity on

It appears to have little impact on runtime, which is expected, since the underlying issue was different and not related to thread migration across cores.

5.2 copied clusters and reduce

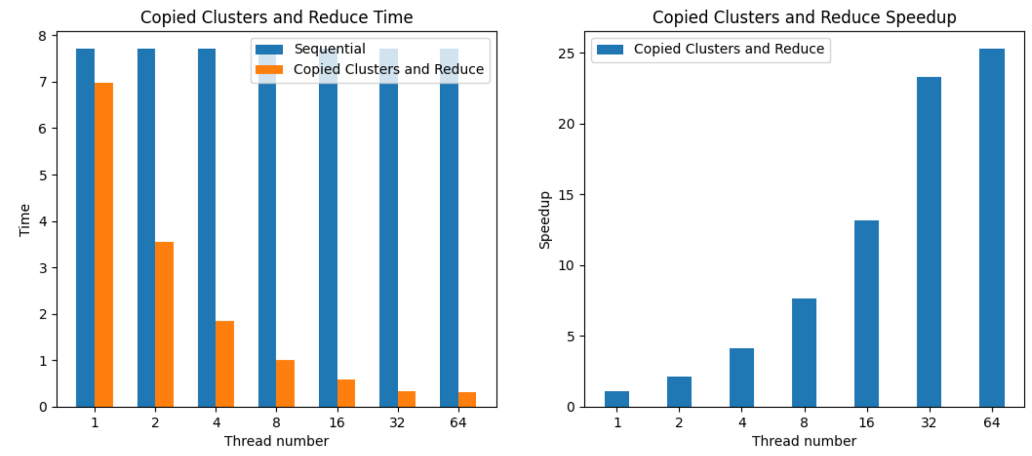


Figure 5: Speedup of Kmeans Copy Reduce

We observe much better scaling and nearly perfect linear speedup. In this implementation, each thread has its own copy of the clusters, and these are combined at the end of each iteration. As a result, there is no resource contention, greatly reducing runtime. There is only a small synchronization overhead at the end of each iteration. Note that good performance is also helped by CPU_AFFINITY, which reduces thread migrations across processors and preserves data locality.

Running getconf -a — grep CACHE on scirouter shows each cache level has cache line size = 64 Bytes. In configuration {256, 1, 4, 10}, the center of each cluster is a point represented by a double = 8 bytes. This means one cache line fits two copies of the local newClusters arrays. Consequently, data from each thread are not actually isolated; they end up on the same cache line, copies of that line are stored in caches of many

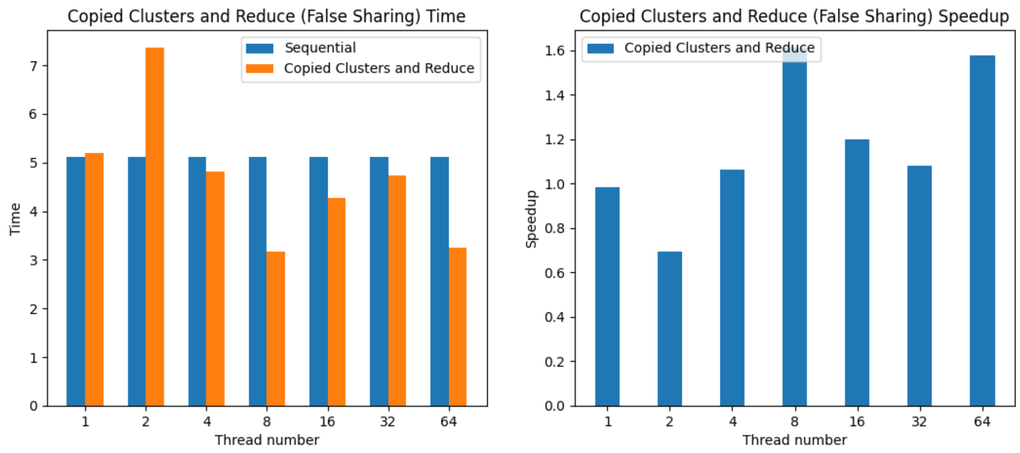


Figure 6: Speedup of Naive Kmeans Shared Clusters

cores, and the coherence protocol is triggered, even though there is no real data sharing. This causes bus traffic and significant delay.

One solution is to exploit Linux's first touch policy (6).

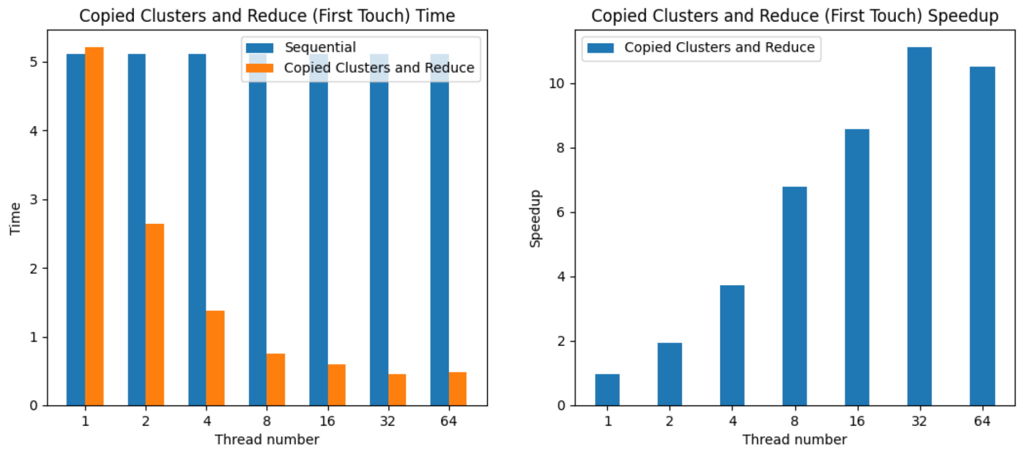


Figure 7: Speedup of Kmeans Copy Reduce First Touch

Another solution is to allocate more space than necessary per thread, effectively inserting zeros in the cache line and thus avoiding false sharing (7).

Of course, both methods can be used simultaneously.

Ultimately, the Zero Padding method scales better.

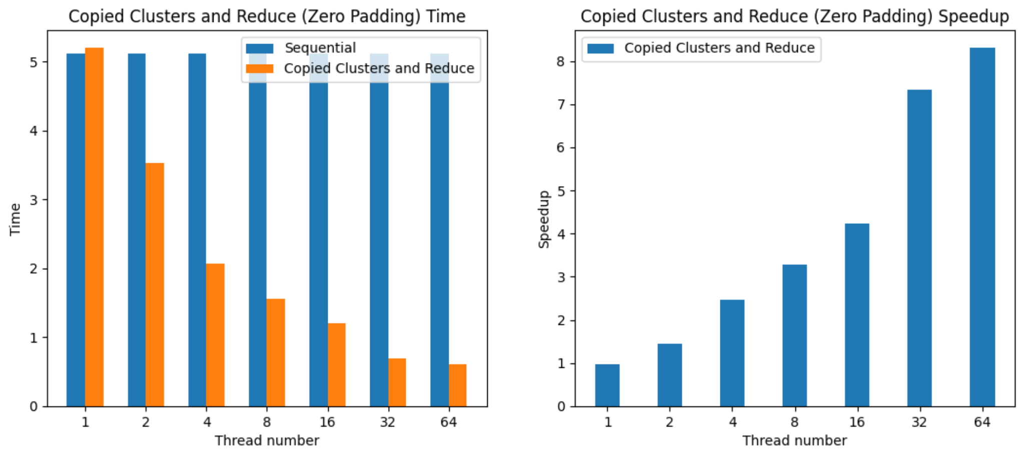


Figure 8: Speedup of Kmeans Copy Reduce Zero Padding

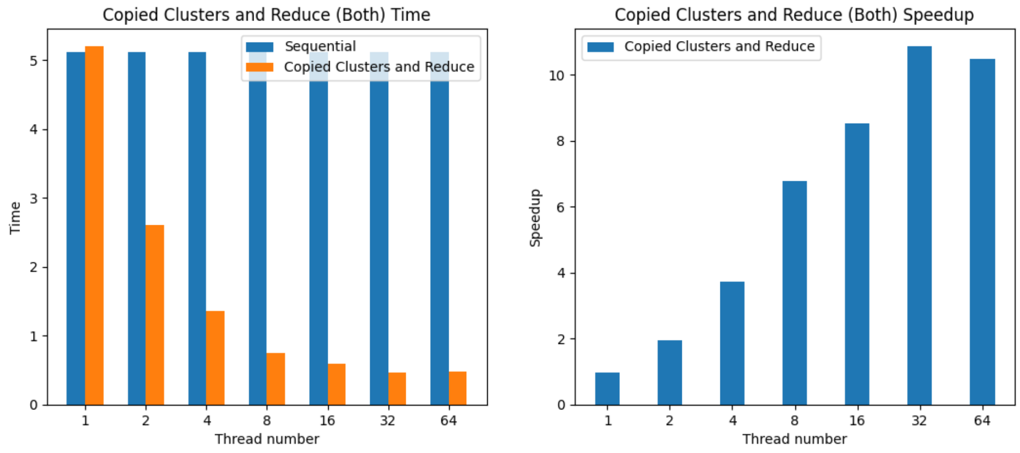
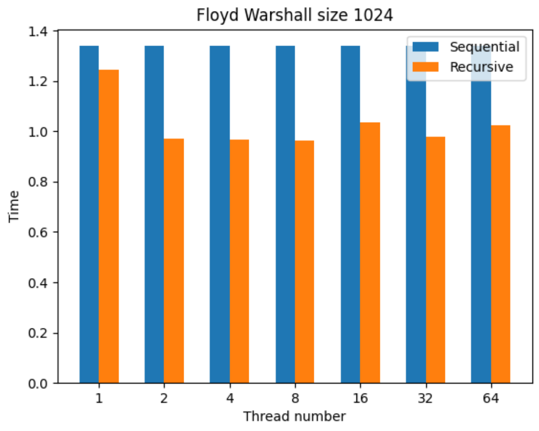


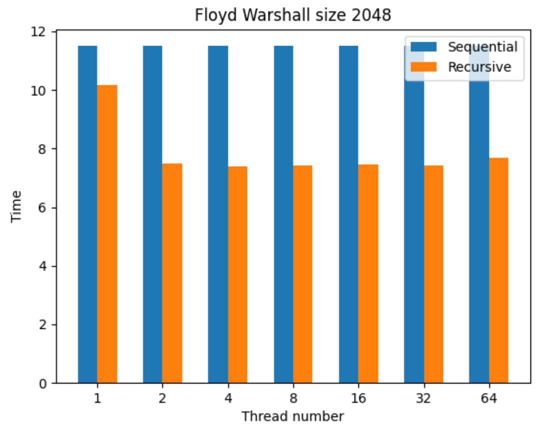
Figure 9: Speedup of Kmeans Copy Reduce Both

6 Floyd-Warshall

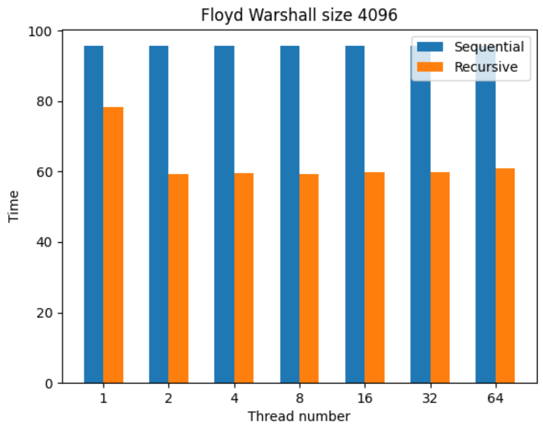
The recursive implementation 8 of the Floyd–Warshall algorithm, while it initially improves runtime, does not scale satisfactorily. This is expected because the task graph does not exhibit much parallelism. Of the eight recursive calls, only four are pairwise parallel, i.e., their data do not depend on one another. Therefore, we parallelize only one of the two tasks for two total calls out of eight, and effectively only one extra thread is used regardless of how many were created. Thus, going from 1 to 2 threads reduces runtime by about 25% minus thread-creation overhead, since the 8 tasks of the recursion complete in the time of 6 tasks. Beyond this point, extra threads do not further reduce runtime; they only add creation overhead.



(a) Floyd Warshall recursive 1024

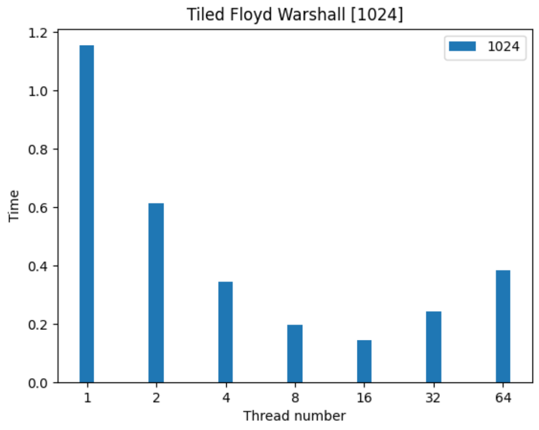


(b) Floyd Warshall recursive 2048

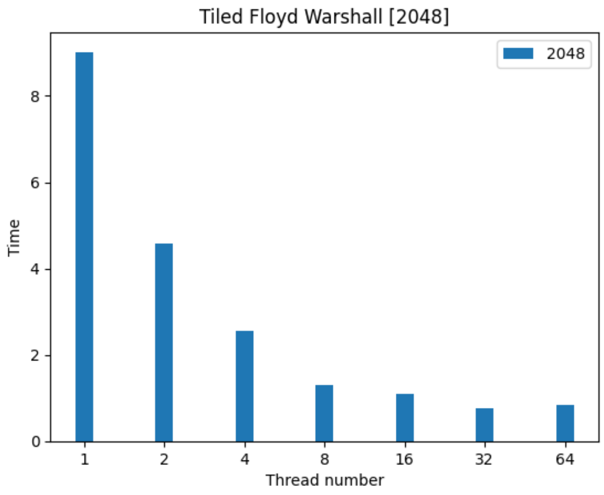


(c) Floyd Warshall recursive 4096

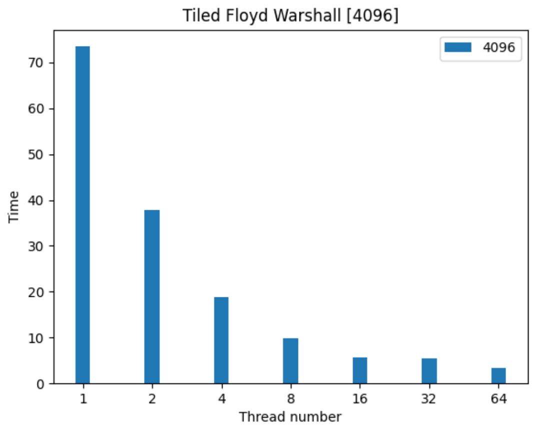
The tiled version 9 exhibits more parallelism, which is reflected in the runtime plots: scaling is clearly satisfactory. Note that scaling degrades earlier for small sizes than for large ones. This relates to problem size versus the cost of spawning more threads. Scaling "breaks" where thread-creation time exceeds the time saved by parallelization. The time saved by parallelization varies with problem size, while the time to create a given number of threads remains constant, leading to different breakpoints depending on problem size. Specifically, at 1024 the breakpoint is at 16 threads; at 2048, somewhere between 32–64 threads; and at 4096, at 64 threads and beyond.



(a) Floyd Warshall tiled 1024



(b) Floyd Warshall tiled 2048



(c) Floyd Warshall tiled 4096

7 Appendix

Listing 5: Naive Kmeans Shared Clusters

```c
...
#pragma omp parallel for private(i, j, index) shared(numObjs, delta, newClusterSize,
    newClusters)
for (i=0; i<numObjs; i++) {
    // 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)
        #pragma omp atomic
        delta += 1.0;

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

    // update new cluster centers : sum of objects located within
    /*
     * TODO: protect update on shared "newClusterSize" array
     */
    #pragma omp atomic
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        /*
         * TODO: protect update on shared "newClusters" array
         */
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
}
// 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];
        }
    }
}
...
```

Listing 6: Kmeans First Touch

```c
...
#pragma omp parallel for private(k)
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));
}
...
```

Listing 7: Kmeans Zero Padding

```c
...
#define CACHE_LINE_SIZE 64
#define PADDING_DOUBLE (CACHE_LINE_SIZE - sizeof(double))

// Each thread calculates new centers using a private space. After that, thread 0 does an
    array reduction on them.
int * local_newClusterSize[nthreads];  // [nthreads][numClusters]
```

```
double * local_newClusters[nthreads];   // [nthreads][numClusters][numCoords]

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 +
     numClusters * PADDING_DOUBLE, sizeof(**local_newClusters));
}
...
```

Listing 8: FW Recursive

```
...
else {  // This is called as FW_SR(A, A, A) so A, B, C are the same array.
    #pragma omp parallel
    {
        #pragma omp single
        {
            FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize); // A00, B00, C00

            #pragma omp task
            FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize); // A01,
     B00, C01
            #pragma omp task if(0)
            {
                    FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
     // A10, B10, C00
            }
            #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); // A11, B10, C01


            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); // A11, B11, C11

            #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); // A10, B11, C10
            #pragma omp task if(0)
            {
                    FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
    myN/2, bsize); // A01, B01, C11
            }
            #pragma omp taskwait
            FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize); // A00,
     B01, C10
        }
    }
}
...
```

Listing 9: FW Tiled

```
...
for(k=0;k<N;k+=B){
        FW(A,k,k,k,B); // top-left corner

        # pragma omp parallel for private(i)
        for(i=0; i<k; i+=B)
                FW(A,k,i,k,B);

        # pragma omp parallel for private(i)
        for(i=k+B; i<N; i+=B)
                FW(A,k,i,k,B);

        # pragma omp parallel for private(j)
        for(j=0; j<k; j+=B)
                FW(A,k,k,j,B);

        # pragma omp parallel for private(j)
        for(j=k+B; j<N; j+=B)
                FW(A,k,k,j,B);

        # pragma omp parallel for private(i, j)
        for(i=0; i<k; i+=B)
                for(j=0; j<k; j+=B)
                        FW(A,k,i,j,B);

        # pragma omp parallel for private(i, j)
        for(i=0; i<k; i+=B)
                for(j=k+B; j<N; j+=B)
                        FW(A,k,i,j,B);

        # pragma omp parallel for private(i, j)
        for(i=k+B; i<N; i+=B)
                for(j=0; j<k; j+=B)
                        FW(A,k,i,j,B);

        # pragma omp parallel for private(i, j)
        for(i=k+B; i<N; i+=B)
                for(j=k+B; j<N; j+=B)
                        FW(A,k,i,j,B);
}
...
```

Part III

Exercise 3

8 K-MEANS

8.1 Mutual Exclusion — Locks

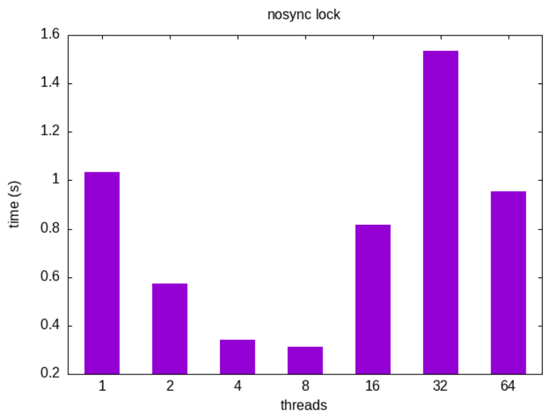
Qualitatively, all implementations—except array lock and clh lock—show the same behavior: they maintain speedup up to 4 threads at best and exhibit large slowdowns for 8 threads and above. Their differences are mainly in absolute timings. This behavior stems from locks being mainly suitable for sporadic accesses to shared data to maintain correctness. Here, all threads simultaneously request access to shared data throughout execution, leading to congestion and, effectively, serialization.

Specifically, pthread mutex lock (Fig. 12b) performs worse already at 8 threads and shows a sharp runtime increase for more threads, reaching up to 4 seconds in the worst case. pthread spin lock (Fig. 12c) behaves similarly but is even slower beyond 16 threads because mutex locks put threads to sleep (freeing resources), whereas spin locks busy-wait and consume compute resources.

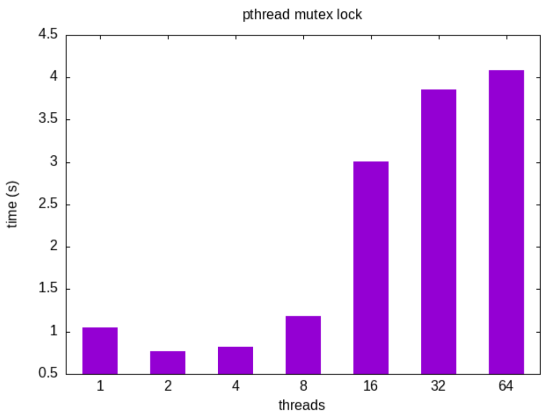
This thread count is not arbitrary: the system has 4 sockets with 4 cores each, supporting up to 2 hardware threads per core—i.e., up to 32 hardware threads in the best case—assuming varied workloads (not true here), so contention is severe.

test and set (Fig. 12d) and test and test and set (Fig. 12e) locks show similar behavior and are even slower, as they are hardware-level primitives typically suitable for short critical sections. Our application locks relatively large critical sections. TTAS is faster than TAS because TAS triggers coherence traffic (bus read x) on every attempt, while TTAS first reads (bus read) and only escalates if needed.

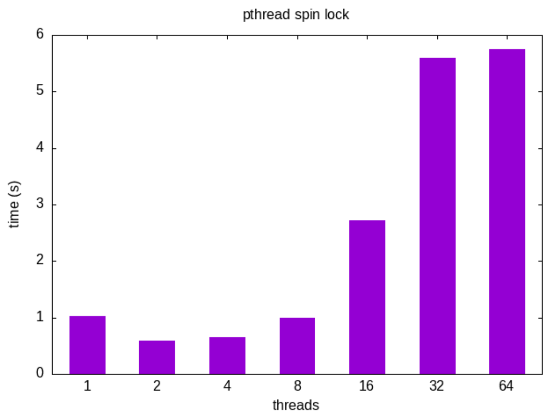
array lock (Fig. 12f) and clh lock (Fig. 12g) perform much better. They allow speedup up to 8 threads and remain faster than other locks even when parallel execution becomes slower than serial (> 16 threads). They coordinate across multiple locks at different memory locations rather than contending on a single lock. In array lock, a boolean array of length equal to the number of threads assigns each thread a lock position; a thread proceeds only when its position is true, then hands off to the next. CLH improves upon array lock with less bus contention and memory usage.



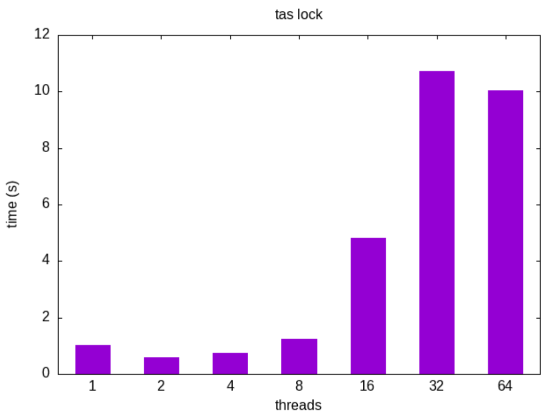
(a) no sync lock



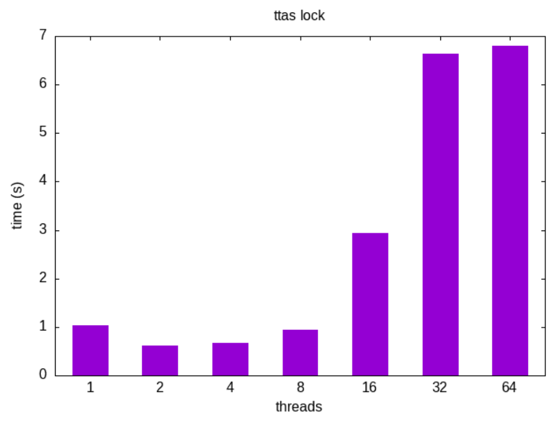
(b) pthread mutex lock



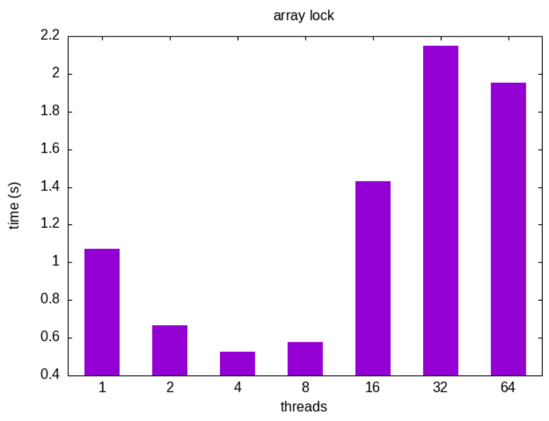
(c) pthread spin lock



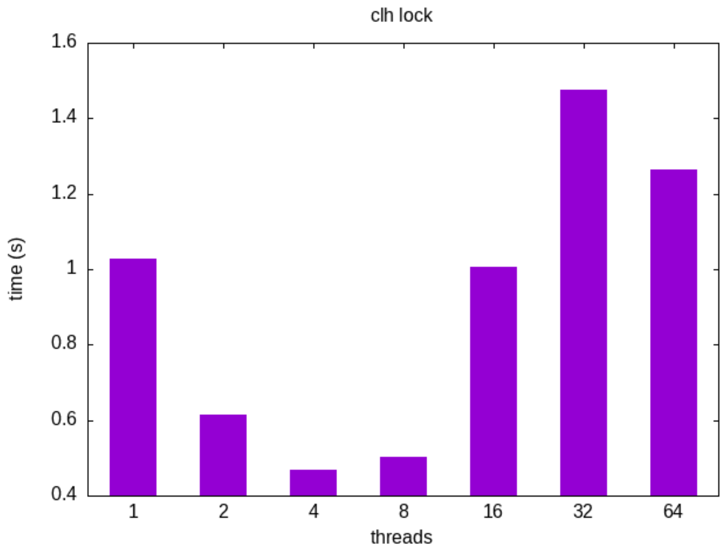
(d) test and set lock



(e) test and test and set lock



(f) array lock



(g) clh lock

8.2 Concurrent Data Structures

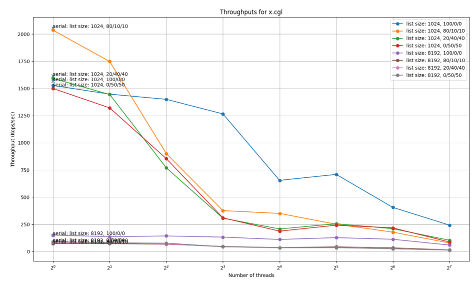
The coarse-grain locking implementation (Fig. 13a) shows decreasing throughput as threads increase, as the entire structure is locked for any operation. Even with 100% lookups, performance drops. The effect is worse when operations beyond search are involved (locks held longer). On longer lists the effect is less pronounced, since search is already time-consuming.

The fine-grain locking implementation (Fig. 13b) has worse single-thread throughput than coarse (lock overhead), but it does not drop further as threads increase; it remains almost flat.

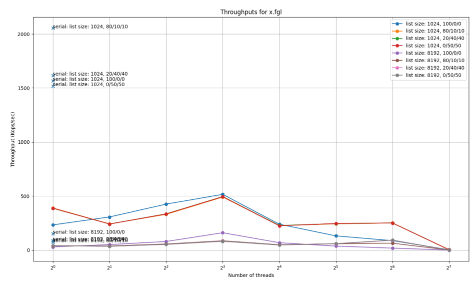
The optimistic implementation (Fig. 13c) scales throughput better by locking only the node that needs modification, eliminating repeated lock/unlock overhead on every node. At 128 threads, a drop appears—likely many threads attempt many operations concurrently and impede each other.

The lazy locking implementation (Fig. 13d) scales well, especially for read-only workloads, because reads do not lock and traverse the list without locks. Other cases also improve over optimistic because validate does not traverse the entire list. Scaling "stalls" near 128 threads, likely due to the limitations of hyperthreading when many hardware threads contend for identical resources.

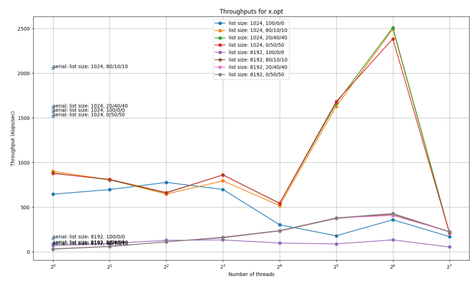
The non-blocking implementation (Fig. 13e) scales better than lazy except for read-only cases, likely due to higher structural overhead but more scalable add/remove (no blocking). Up to 16 threads lazy is more efficient; at 32–64 they are similar; at 128 non-blocking is faster.



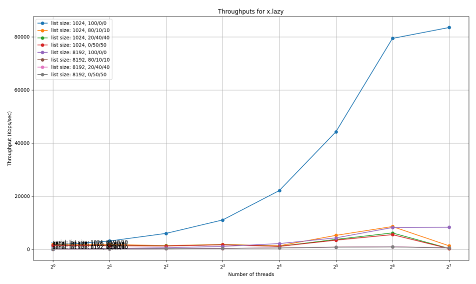
(a) coarse-grain locking



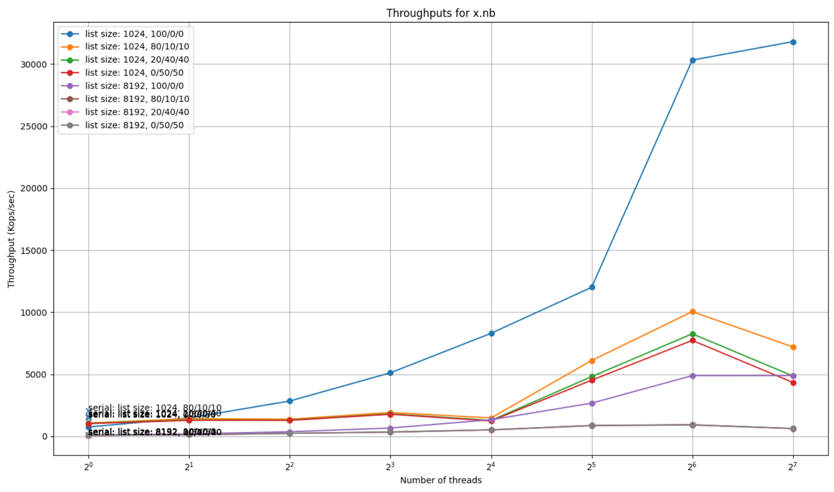
(b) fine-grain locking



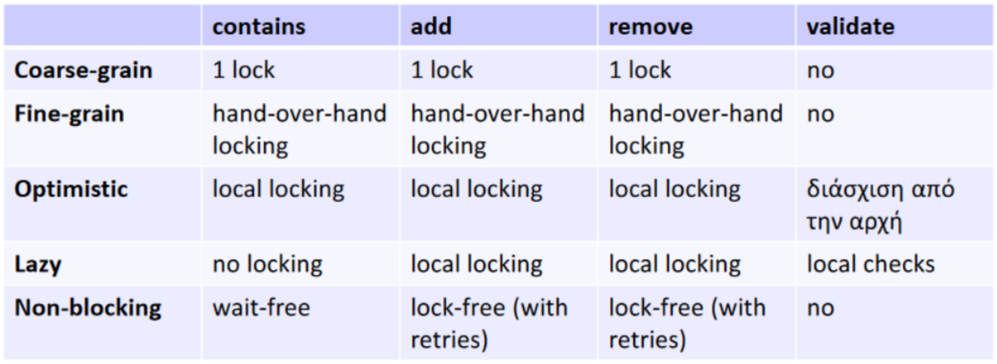
(c) optimistic locking



(d) lazy locking



(e) non-blocking locking



	contains	add	remove	validate
Coarse-grain	1 lock	1 lock	1 lock	no
Fine-grain	hand-over-hand locking	hand-over-hand locking	hand-over-hand locking	no
Optimistic	local locking	local locking	local locking	διάσχιση από την αρχή
Lazy	no locking	local locking	local locking	local checks
Non-blocking	wait-free	lock-free (with retries)	lock-free (with retries)	no

(f) Source: course notes

The comparison of implementations is also summarized very briefly by the table below:

Part IV

Exercise 4

9 Naive

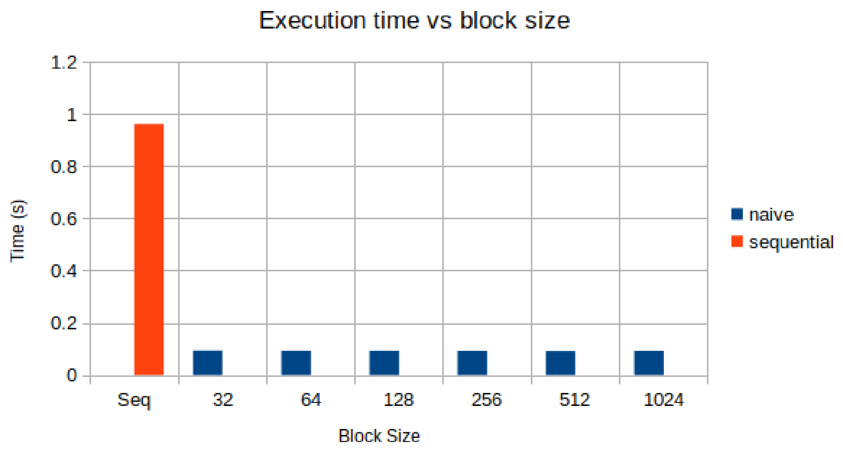


Figure 14: Execution time

The parallel implementation is faster than the serial baseline. Offloading the heavy compute to the GPU yields almost an order-of-magnitude speedup (about $\times 10$). Regarding block size, performance is relatively stable. Generally, performance versus block size is influenced via occupancy. Thus either occupancy is constant and independent of block size (possibly 100%) or total time is dominated by CPU–GPU communication and the GPU executes the compute portion quickly. On our GPU:

$$block\_size\_for\_max\_occupancy \geq \frac{number\_of\_max\_blocks}{total\_threads} = \frac{2048}{32} = 64 \quad (1)$$

This is reflected in the plots: a slight speedup for block size ≥ 64 threads.

10 Transpose

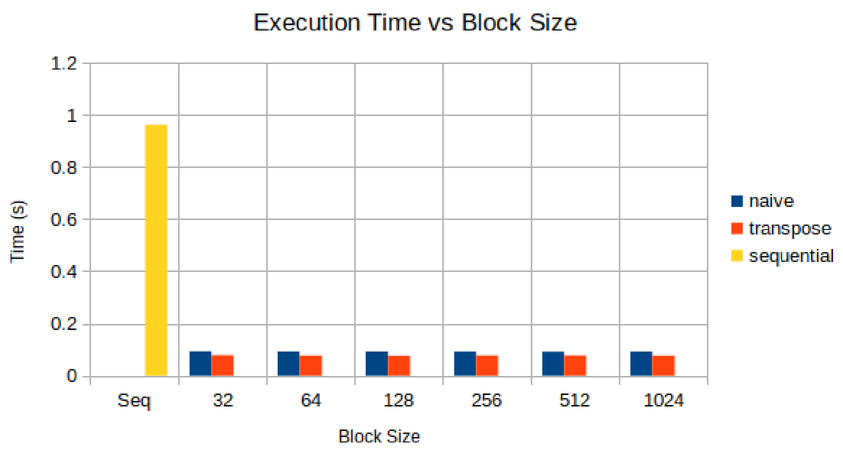


Figure 15: Execution time

The transpose implementation again outperforms naive yet remains relatively independent of block size. This likely stems from better memory coalescing with transpose: data needed by each warp are contiguous—something that may not hold in the naive approach. Transpose changes how the GPU accesses data, not algorithmic complexity.

11 Shared

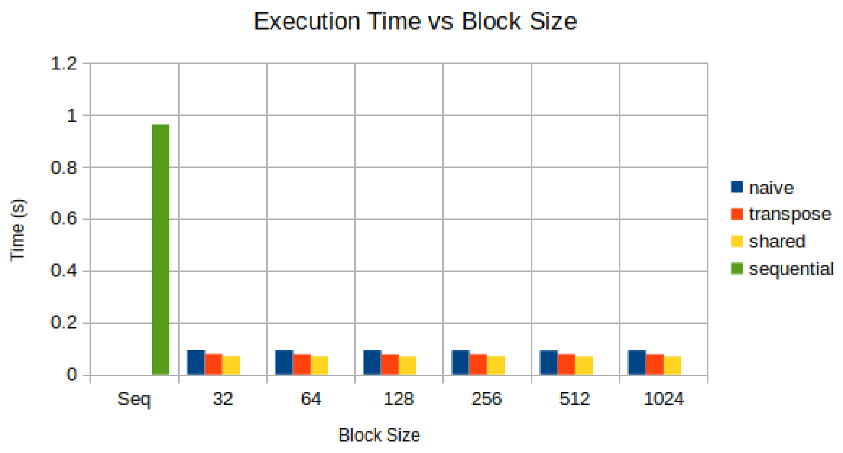
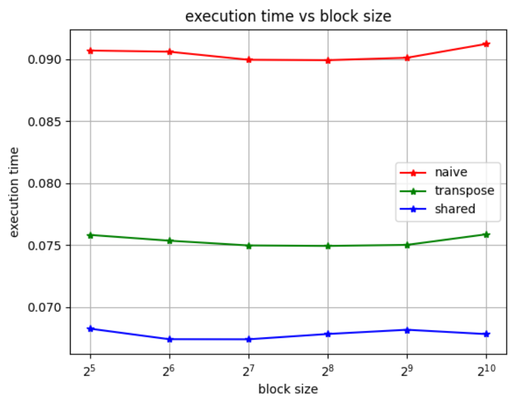
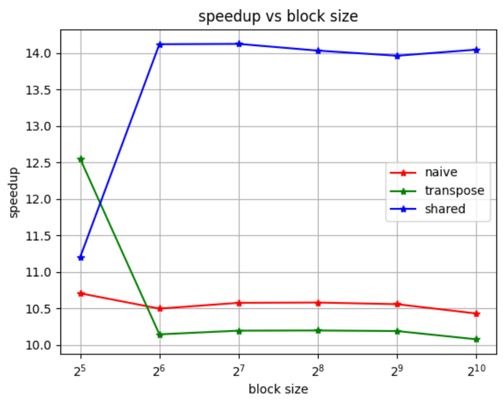


Figure 16: Execution time

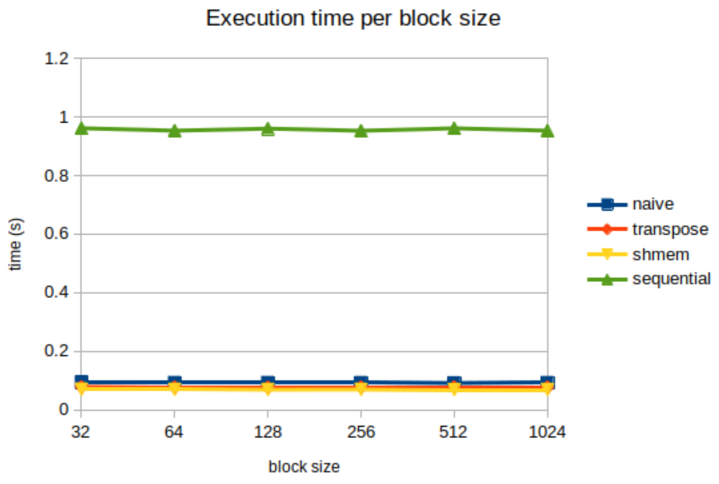
The shared memory implementation provides another performance step beyond transpose, and again appears fairly independent of (thread) block size. Overall runtime is dominated by data movement; shared memory mitigates this by allowing threads to update on-chip shared memory. Hence, execution time is not strongly affected by block size.



(a) Execution time



(b) speedup



(c) Execution time compared to serial

12 Bottleneck Analysis

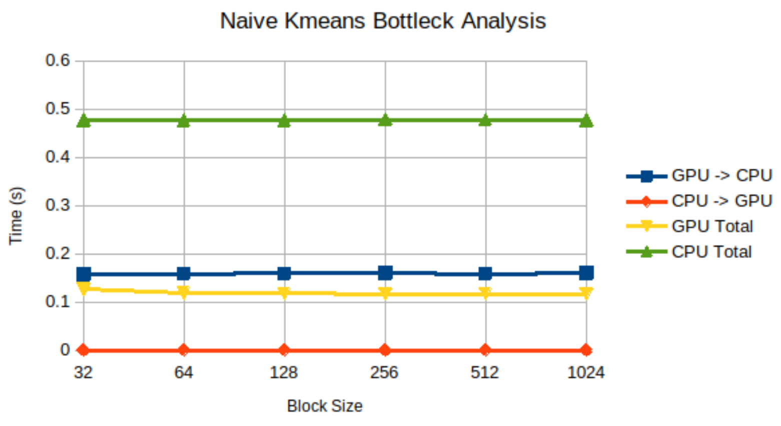
12.1 2 Coordinates

For two dimensions (see Fig. 18a), CPU time dominates in every case and is unaffected by block size or method (naive, transpose, shared), since the CPU-side work is unchanged. GPU→CPU transfer time is also stable across methods and block sizes; CPU→GPU transfer is about three orders of magnitude smaller and nearly constant (variation $< 5\%$), so a log scale adds little value.

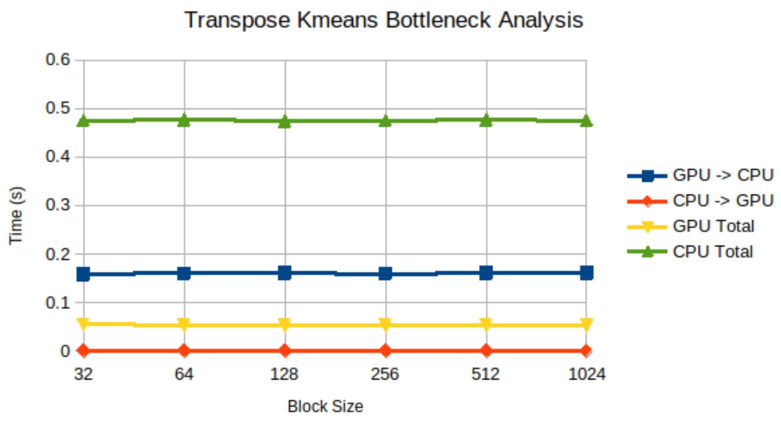
GPU compute time decreases from naive to transpose (better coalescing). Shared also improves memory access vs naive but not much over transpose in 2D because the huge number of objects introduces many race conditions. With 16D, shared will show clearer gains.

Regarding block size, in 2D all times remain relatively stable regardless of method, likely because block size is much larger than the maximum parallelism the system can exploit. Objects vastly outnumber the maximum block sizes, so rounding to a larger block size does not matter.

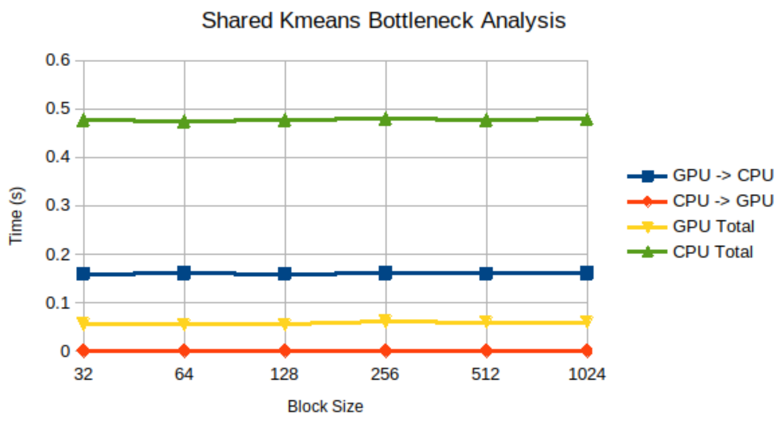
The GPU provides 32 FP64 units per SM; since the kernel is FP64-heavy, not all thread demands can be satisfied simultaneously.



(a) naive execution time versus block size



(b) transpose execution time versus block size



(c) shared execution time versus block size

Figure 18: configuration with coordinate dim = 2

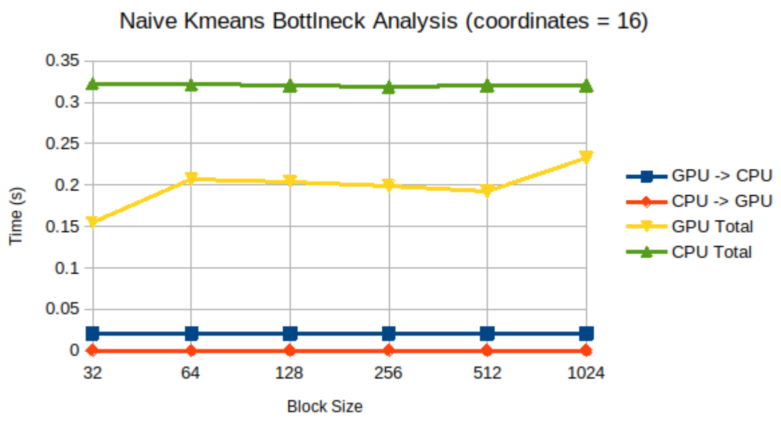
12.2 16 coordinates

Although the total data volume remains constant, CPU compute time decreases. The relevant code is shown for clarity:

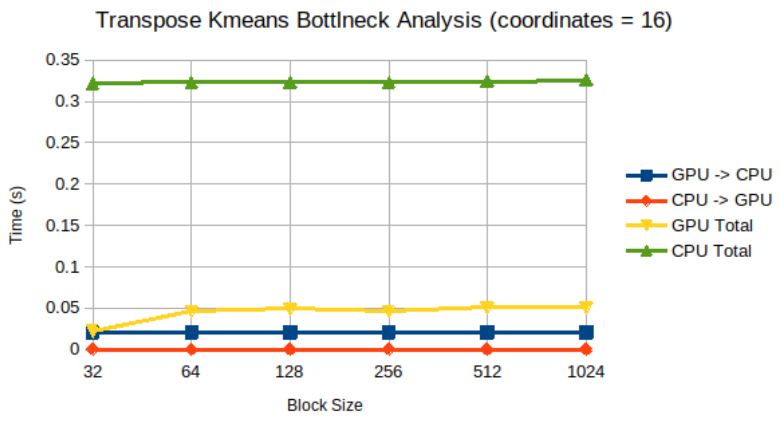
```
/* CPU part: Update cluster centers*/
for (i=0; i<numObjs; i++) {
    /* find the array index of nestest cluster center */
    index = membership[i];

    /* update new cluster centers : sum of objects located within */
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        newClusters[j][index] += objects[i*numCoords + j];
}
/* average the sum and replace old cluster centers with newClusters */
for (i=0; i<numClusters; i++) {
    for (j=0; j<numCoords; j++) {
        if (newClusterSize[i] > 0)
            clusters[i*numCoords + j] = newClusters[i][j] / newClusterSize[i];
        newClusters[j][i] = 0.0;   /* set back to 0 */
    }
    newClusterSize[i] = 0;   /* set back to 0 */
}
```

The number of objects is reduced by a factor of eight. Therefore, the first double loop executes $16777216 * 3 - 2097152 * 3 = 44{,}040{,}192$ fewer instructions (notably fewer accesses to membership and newClusterSize). The second double loop executes $16 * 16 - 16 * 2 = 224$ more instructions—negligible compared to the large reduction in the first loop.

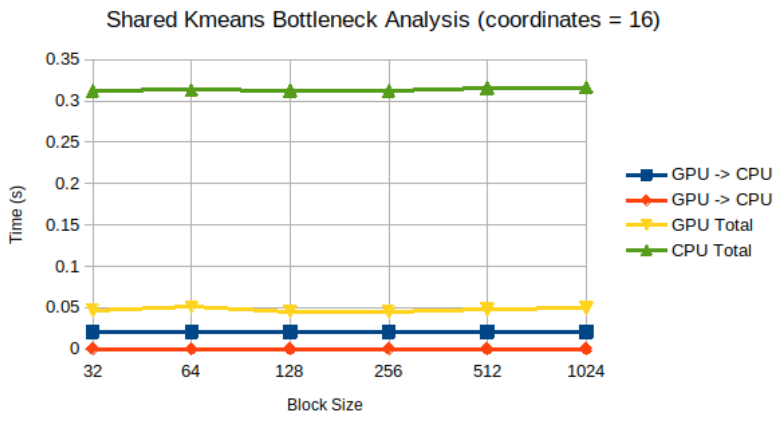
GPU→CPU transfer time is also divided by eight, meaning that increased parallelism enabled by more dimensions (independent per dimension) is fully reflected in transfers (time drops from ~ 0.16 to ~ 0.02). Transfer cost scales per object; e.g., the membership buffer has size equal to the number of objects.



(a) naive execution time versus block size



(b) transpose execution time versus block size



(c) shared execution time versus block size

Figure 19: configuration with coordinate dim = 16

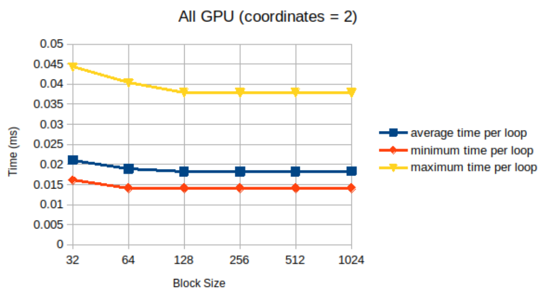
13 Bonus: Full Offload (All-GPU) version

For the “ALL GPU” version, with more coordinates and the same total data volume (thus eight times fewer objects), execution is faster because coordination costs for cluster sizes and centroid updates are smaller: one-eighth the objects compete for eight times the per-dimension resources.

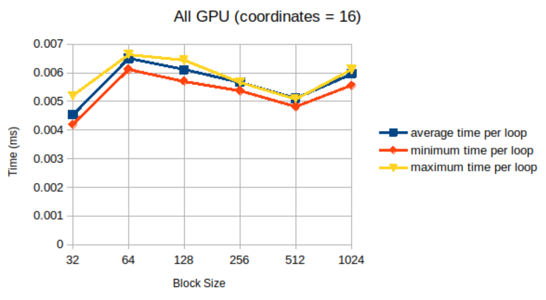
update_centroids was implemented partly in find_nearest_cluster: immediately after finding the nearest cluster, atomic updates effectively perform reduce(+) of members and cluster sizes. Then update_centroids assigns each thread a cluster and a dimension and divides by the cluster size to compute the mean (see 13).

As noted, the GPU provides 32 FP64 units per SM; not all thread demands can be satisfied simultaneously. In 2D, each thread uses resources for less time than in 16D, so warp schedulers can issue other warps more often. With 4 warp schedulers per SM, any improvement will be visible up to 128 threads per block. Beyond that, contention is very high.

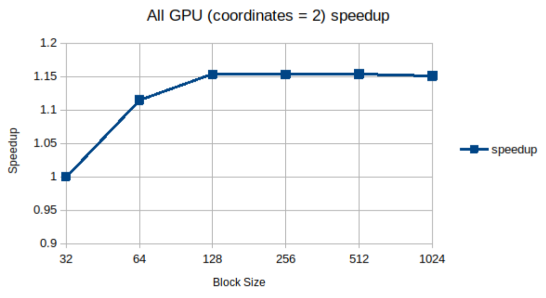
The 16D experiment appears to suffer from warp blockage. An SM cannot evict a block and replace it with another until the last warp completes. With large blocks (many warps), SM resources are more likely to be exhausted, delaying some warps and thus the entire block. Also, with larger blocks at fixed problem size, there are fewer blocks overall—fewer choices for scheduling can also cause delays. This is visible up to 64 threads per block. The behavior at other block sizes likely relates to moving update_centroids to the GPU.



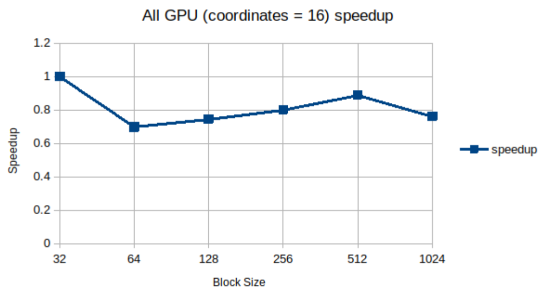
(a) all GPU execution time, 2 coordinates



(b) all GPU execution time, 16 coordinates



(c) All GPU Speedup, 2 coordinates



(d) All GPU Speedup, 16 coordinates

update_centroids is well-suited to the GPU because it can be implemented with perfect parallelism. From the bottlenecks, the worst time is CPU Total (CPU compute time). The all-GPU approach eliminates a large portion of this time and thus mitigates the bottleneck.

The second configuration has more dimensions; the performance difference arises because it performs eight times more per-object coordinate work on the GPU but with far fewer objects and better parallel efficiency.

14 Appendix

Skip the code below: *Skip Code*

Listing 10: Kmeans GPU Naive

```
__device__ int get_tid(){
  return blockIdx.x * blockDim.x + threadIdx.x; /* TODO: copy me from naive version... */
}

/* square of Euclid distance between two multi-dimensional points */
__host__ __device__ inline static
double euclid_dist_2(int    numCoords,
                    int    numObjs,
                    int    numClusters,
                    double *objects,     // [numObjs][numCoords]
                    double *clusters,    // [numClusters][numCoords]
                    int    objectId,
                    int    clusterId)
{
    int i;
    double ans=0.0;

  /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
    clusters*/
    // was empty
    for (i=0; i<numCoords; i++)
        ans += (objects[objectId*numCoords + i] - clusters[clusterId*numCoords + i]) *
               (objects[objectId*numCoords + i] - clusters[clusterId*numCoords + i]);

    return(ans);
}

__global__ static
void find_nearest_cluster(int numCoords,
                          int numObjs,
                          int numClusters,
                          double *objects,           //  [numObjs][numCoords]
                          double *deviceClusters,    //  [numClusters][numCoords]
                          int *deviceMembership,          //  [numObjs]
                          double *devdelta)
{

  /* Get the global ID of the thread. */
    int tid = get_tid();

  /* TODO: Maybe something is missing here... should all threads run this? */
    if (tid < numObjs) { // was 1
        int   index, i;
        double dist, min_dist;

        /* find the cluster id that has min distance to object */
        index = 0;
        /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
        min_dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters,
    tid, 0); // was empty

        for (i=1; i<numClusters; i++) {
            /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
            dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters,
    tid, i); // was empty

            /* no need square root */
            if (dist < min_dist) { /* find the min and its array index */
                min_dist = dist;
                index    = i;
            }
        }

        if (deviceMembership[tid] != index) {
          /* TODO: Maybe something is missing here... is this write safe? */
            atomicAdd(devdelta, 1.0); // was (*devdelta)+= 1.0;
        }

        /* assign the deviceMembership to object objectId */
        deviceMembership[tid] = index;
    }
}


    const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize)? blockSize: numObjs
    ;
    /* TODO: Calculate Grid size, e.g. number of blocks. */
    const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
    numThreadsPerClusterBlock; // was -1
    const unsigned int clusterBlockSharedDataSize = 0;


    do {
        timing_internal = wtime();

    /* GPU part: calculate new memberships */
        #ifdef TIMER_ANALYSIS
        time_start = wtime();
        #endif

        /* TODO: Copy clusters to deviceClusters
        checkCuda(cudaMemcpy(...)); */
        checkCuda(cudaMemcpy(deviceClusters, clusters,
              numClusters*numCoords*sizeof(double), cudaMemcpyHostToDevice));

        checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));

        #ifdef TIMER_ANALYSIS
        TIME(cpu_gpu_time);
        #endif

        find_nearest_cluster
            <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
            (numCoords, numObjs, numClusters,
             deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);

        cudaDeviceSynchronize(); checkLastCudaError();

        #ifdef TIMER_ANALYSIS
        TIME(gpu_time);
        #endif

    /* TODO: Copy deviceMembership to membership
        checkCuda(cudaMemcpy(...)); */
        checkCuda(cudaMemcpy(membership, deviceMembership,
              numObjs*sizeof(int), cudaMemcpyDeviceToHost));

      /* TODO: Copy dev_delta_ptr to &delta
        checkCuda(cudaMemcpy(...)); */
        checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
              sizeof(double), cudaMemcpyDeviceToHost));

        #ifdef TIMER_ANALYSIS
        TIME(gpu_cpu_time);
        #endif

    /* CPU part: Update cluster centers*/

        for (i=0; i<numObjs; i++) {
            /* find the array index of nestest cluster center */
            index = membership[i];

            /* update new cluster centers : sum of objects located within */
            newClusterSize[index]++;
            for (j=0; j<numCoords; j++)
                newClusters[index][j] += objects[i*numCoords + j];
        }

        /* average the sum and replace old cluster centers with newClusters */
        for (i=0; i<numClusters; i++) {
            for (j=0; j<numCoords; j++) {
                if (newClusterSize[i] > 0)
                    clusters[i*numCoords + j] = newClusters[i][j] / newClusterSize[i];
                newClusters[i][j] = 0.0;   /* set back to 0 */
            }
            newClusterSize[i] = 0;   /* set back to 0 */
        }

        delta /= numObjs;
        //printf("delta is %f - ", delta);
        loop++;
        ...
    } while (delta > threshold && loop < loop_threshold);
```

Listing 11: Kmeans GPU Transpose

```
__device__ int get_tid(){
  return blockIdx.x * blockDim.x + threadIdx.x; /* TODO: copy me from naive version... */
}

/* square of Euclid distance between two multi-dimensional points using column-base format
    */
  __host__ __device__ inline static
double euclid_dist_2_transpose(int numCoords,
    int    numObjs,
    int    numClusters,
    double *objects,     // [numCoords][numObjs]
    double *clusters,    // [numCoords][numClusters]
    int    objectId,
    int    clusterId)
{
  int i;
  double ans=0.0;

  /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
    clusters, but for column-base format!!! */
  for (i = 0; i < numCoords; i++)
    ans += (objects[i*numObjs + objectId] - clusters[i*numClusters + clusterId]) *
      (objects[i*numObjs + objectId] - clusters[i*numClusters + clusterId]);

  return(ans);
}

  __global__ static
void find_nearest_cluster(int numCoords,
    int numObjs,
    int numClusters,
    double *objects,           //  [numCoords][numObjs]
    double *deviceClusters,    //  [numCoords][numClusters]
    int *membership,          //  [numObjs]
    double *devdelta)
{
  /* TODO: copy me from naive version... */

  /* Get the global ID of the thread. */
  int tid = get_tid();

  /* TODO: Maybe something is missing here... should all threads run this? */
  if (tid < numObjs) { // was 1
    int   index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
    deviceClusters, tid, 0); // was empty

    for (i=1; i<numClusters; i++) {
      /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
      dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
    deviceClusters, tid, i); // was empty

      /* no need square root */
      if (dist < min_dist) { /* find the min and its array index */
        min_dist = dist;
        index    = i;
      }
    }

    if (membership[tid] != index) {
      /* TODO: Maybe something is missing here... is this write safe? */
      atomicAdd(devdelta, 1.0); // was (*devdelta)+= 1.0;
    }

    /* assign the deviceMembership to object objectId */
    membership[tid] = index;
  }
}
...
/* TODO: Transpose dims */
double  **dimObjects = NULL; //calloc_2d(...) -> [numCoords][numObjs]
double  **dimClusters = NULL;  //calloc_2d(...) -> [numCoords][numClusters]
double  **newClusters = NULL;  //calloc_2d(...) -> [numCoords][numClusters]
dimObjects  = (double**) calloc_2d(numCoords, numObjs, sizeof(double));
dimClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));
newClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));
...

//  TODO: Copy objects given in [numObjs][numCoords] layout to new
//  [numCoords][numObjs] layout
for (i = 0; i < numCoords; i++) {
  for (j = 0; j < numObjs; j++) {
    dimObjects[i][j] = objects[j*numCoords + i];
  }
}
...
do {
  timing_internal = wtime();

  /* GPU part: calculate new memberships */

#ifdef TIMER_ANALYSIS
  time_start = wtime();
#endif

  /* TODO: Copy clusters to deviceClusters
     checkCuda(cudaMemcpy(...)); */
  checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
        numClusters*numCoords*sizeof(double), cudaMemcpyHostToDevice));

  checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));

#ifdef TIMER_ANALYSIS
  TIME(cpu_gpu_time);
#endif

  find_nearest_cluster
    <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
    (numCoords, numObjs, numClusters,
     deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);

  cudaDeviceSynchronize(); checkLastCudaError();

#ifdef TIMER_ANALYSIS
  TIME(gpu_time);
#endif

  /* TODO: Copy deviceMembership to membership
     checkCuda(cudaMemcpy(...)); */
  checkCuda(cudaMemcpy(membership, deviceMembership,
        numObjs*sizeof(int), cudaMemcpyDeviceToHost));

  /* TODO: Copy dev_delta_ptr to &delta
```

```c
     checkCuda(cudaMemcpy(...)); */
  checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
        sizeof(double), cudaMemcpyDeviceToHost));

#ifdef TIMER_ANALYSIS
  TIME(gpu_cpu_time);
#endif

  /* CPU part: Update cluster centers*/

  for (i=0; i<numObjs; i++) {
    /* find the array index of nestest cluster center */
    index = membership[i];

    /* update new cluster centers : sum of objects located within */
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
      newClusters[j][index] += objects[i*numCoords + j];
  }

  /* average the sum and replace old cluster centers with newClusters */
  for (i=0; i<numClusters; i++) {
    for (j=0; j<numCoords; j++) {
      if (newClusterSize[i] > 0)
        dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
      newClusters[j][i] = 0.0;   /* set back to 0 */
    }
    newClusterSize[i] = 0;   /* set back to 0 */
  }

  delta /= numObjs;
  //printf("delta is %f - ", delta);
  loop++;
  //printf("completed loop %d\n", loop);
  ...
} while (delta > threshold && loop < loop_threshold);

/*TODO: Update clusters using dimClusters. Be carefull of layout!!! clusters[numClusters][
    numCoords] vs dimClusters[numCoords][numClusters] */
for (i = 0; i < numCoords; i++) {
  for (j = 0; j < numClusters; j++) {
    clusters[j*numCoords + i] = dimClusters[i][j];
  }
}
```

Listing 12: Kmeans GPU Shared

```
__device__ int get_tid(){
  return blockIdx.x * blockDim.x + threadIdx.x; /* TODO: copy me from naive version... */
}

/* square of Euclid distance between two multi-dimensional points using column-base format
    */
  __host__ __device__ inline static
double euclid_dist_2_transpose(int numCoords,
    int    numObjs,
    int    numClusters,
    double *objects,     // [numCoords][numObjs]
    double *clusters,    // [numCoords][numClusters]
    int    objectId,
    int    clusterId)
{
  int i;
  double ans=0.0;

  /* TODO: Copy me from transpose version*/

  /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
    clusters, but for column-base format!!! */
  for (i = 0; i < numCoords; i++)
    ans += (objects[i*numObjs + objectId] - clusters[i*numClusters + clusterId]) *
      (objects[i*numObjs + objectId] - clusters[i*numClusters + clusterId]);

  return(ans);
}
  __global__ static
void find_nearest_cluster(int numCoords,
    int numObjs,
    int numClusters,
    double *objects,           //  [numCoords][numObjs]
    double *deviceClusters,    //  [numCoords][numClusters]
    int *deviceMembership,          //  [numObjs]
    double *devdelta)
{
  extern __shared__ double shmemClusters[];

  /* TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
BEWARE: Make sure operations is complete before any thread continues... */
  for (int i = 0; i < numClusters; i++) {
    for (int j = 0; j < numCoords; j++) {
      shmemClusters[j * numClusters + i] = deviceClusters[j * numClusters + i];
    }
  }

  __syncthreads();

  /* Get the global ID of the thread. */
  int tid = get_tid();

  /* TODO: Maybe something is missing here... should all threads run this? */
  if (tid < numObjs) { // was 1
    int   index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
    shmemClusters, tid, 0); // was empty

    for (i=1; i<numClusters; i++) {
      /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
      dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters
    , tid, i); // was empty

      /* no need square root */
      if (dist < min_dist) { /* find the min and its array index */
        min_dist = dist;
        index    = i;
      }
    }

    if (deviceMembership[tid] != index) {
      /* TODO: Maybe something is missing here... is this write safe? */
      atomicAdd(devdelta, 1.0); // was (*devdelta)+= 1.0;
    }

    /* assign the deviceMembership to object objectId */
    deviceMembership[tid] = index;
  }
}
...
/*  Define the shared memory needed per block.
  - BEWARE: We can overrun our shared memory here if there are too many
  clusters or too many coordinates!
  - This can lead to occupancy problems or even inability to run.
  - Your exercise implementation is not requested to account for that (e.g. always assume
    deviceClusters fit in shmemClusters */
const unsigned int clusterBlockSharedDataSize = numClusters*numCoords*sizeof(double);
...
do {
  timing_internal = wtime();
  /* GPU part: calculate new memberships */
#ifdef TIMER_ANALYSIS
  time_start = wtime();
#endif
  /* TODO: Copy clusters to deviceClusters
     checkCuda(cudaMemcpy(...)); */
  checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
        clusterBlockSharedDataSize, cudaMemcpyHostToDevice));

  checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
#ifdef TIMER_ANALYSIS
  TIME(cpu_gpu_time);
#endif
  //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
    shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock,
    clusterBlockSharedDataSize/1000);
  find_nearest_cluster
    <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
    (numCoords, numObjs, numClusters,
     deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);

  cudaDeviceSynchronize(); checkLastCudaError();
  //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
#ifdef TIMER_ANALYSIS
  TIME(gpu_time);
#endif
  /* TODO: Copy deviceMembership to membership
     checkCuda(cudaMemcpy(...)); */
  checkCuda(cudaMemcpy(membership, deviceMembership,
        numObjs*sizeof(int), cudaMemcpyDeviceToHost));

  /* TODO: Copy dev_delta_ptr to &delta
     checkCuda(cudaMemcpy(...)); */
```

```c
  checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
        sizeof(double), cudaMemcpyDeviceToHost));
#ifdef TIMER_ANALYSIS
  TIME(gpu_cpu_time);
#endif
  /* CPU part: Update cluster centers*/
  for (i=0; i<numObjs; i++) {
    /* find the array index of nestest cluster center */
    index = membership[i];

    /* update new cluster centers : sum of objects located within */
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
      newClusters[j][index] += objects[i*numCoords + j];
  }
  /* average the sum and replace old cluster centers with newClusters */
  for (i=0; i<numClusters; i++) {
    for (j=0; j<numCoords; j++) {
      if (newClusterSize[i] > 0)
        dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
      newClusters[j][i] = 0.0;   /* set back to 0 */
    }
    newClusterSize[i] = 0;   /* set back to 0 */
  }
  delta /= numObjs;
  loop++;
  ...
} while (delta > threshold && loop < loop_threshold);

/*TODO: Update clusters using dimClusters. Be carefull of layout!!! clusters[numClusters][
    numCoords] vs dimClusters[numCoords][numClusters] */
for (i = 0; i < numCoords; i++) {
  for (j = 0; j < numClusters; j++) {
    clusters[j*numCoords + i] = dimClusters[i][j];
  }
}
```

Listing 13: Kmeans All GPU

```
__global__ static
void find_nearest_cluster(int numCoords,
        int numObjs,
        int numClusters,
        double *deviceobjects,           //  [numCoords][numObjs]
        /*
                                    TODO: If you choose to do (some of) the new centroid
    calculation here, you will need some extra parameters here (from "update_centroids").
         */
        int *devicenewClusterSize,            //  [numClusters]
        double *devicenewClusters,     //  [numCoords][numClusters]
                                      //added above two
        double *deviceClusters,    //  [numCoords][numClusters]
        int *deviceMembership,           //  [numObjs]
        double *devdelta)
{
    extern __shared__ double shmemClusters[];

    /* TODO: copy me from shared version... */
    . . .
    /* TODO: additional steps for calculating new centroids in GPU? */
    atomicAdd(&devicenewClusterSize[index], 1);
    for (i = 0; i < numCoords; ++i)
        atomicAdd(&devicenewClusters[i*numClusters + index], deviceobjects[i*numObjs + tid])
    ;
}

    __global__ static
void update_centroids(int numCoords,
        int numClusters,
        int *devicenewClusterSize,            //  [numClusters]
        double *devicenewClusters,     //  [numCoords][numClusters]
        double *deviceClusters)    //  [numCoords][numClusters])
{

    /* TODO: additional steps for calculating new centroids in GPU? */
    //was empty
    const int tid = get_tid();
    if (tid >= numClusters*numCoords) return;
    int cluster = tid % numClusters; // tid = coord*numClusters + cluster, which makes
    access bellow fast af
    if (devicenewClusterSize[cluster] > 0)
        deviceClusters[tid] = devicenewClusters[tid]/devicenewClusterSize[cluster];
    devicenewClusters[tid] = 0.0;
    // apparently synchronizing here doesn't change the results (also each thread does it
    lol, could add if (coord == 0))
    devicenewClusterSize[cluster] = 0;
}
```

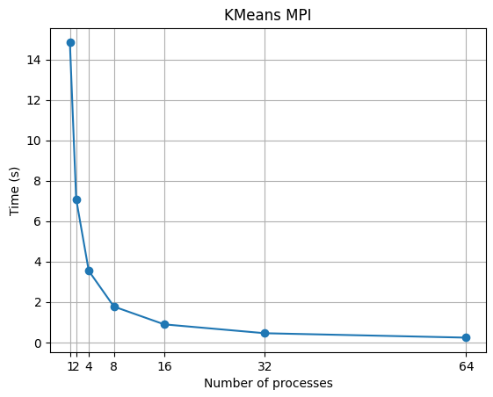
Part V

Exercise 5

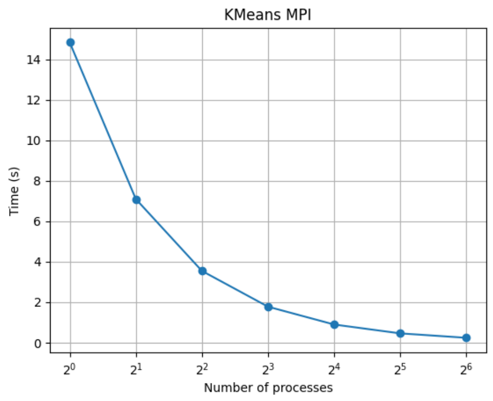
15 K-MEANS

15.1 MPI Implementation

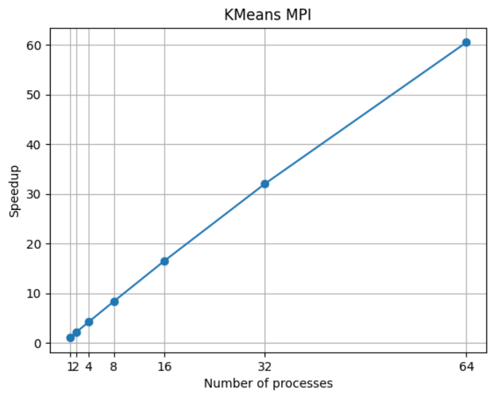
From the plots, scaling is highly satisfactory, achieving nearly ideal speedup.



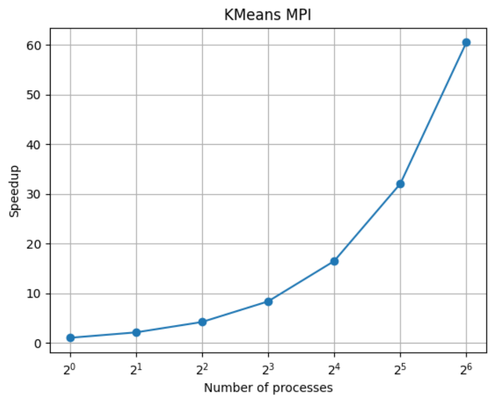
(a) Execution time, comparison (excluding serial)



(b) Execution time (log scale)



(c) Speedup



(d) Speedup (log scale)

15.2 BONUS Question: Comparison with OpenMP

Ignoring specific numeric values, the MPI implementation (distributed memory model) scales better and does not show a tendency to degrade at higher process counts and data sizes, unlike the OpenMP version. See, e.g., 8 vs 21d.

16 2D Heat Diffusion

16.1 Measurements with Convergence Check

Figure 22 shows output from the parallel Jacobi algorithm on a 1024×1024 grid with 64 processes. Most time is spent on inter-process communication. This is mainly due to the convergence check, which each process performs independently on its subgrid, and the use of MPI_Allreduce to combine results before proceeding.

We also note that the required iterations to converge are on the order of a million and that the temperature at the grid center is non-zero.

```
Jacobi X 1024 Y 1024 Px 8 Py 8 Iter 798201
ComputationTime 39.826432 MessagingTime 179.486283 TotalTime 222.866609 midpoint 5.431022
```

Figure 22: Output Data Jacobi 1024 × 1024

16.2 Measurements without Convergence Check

Here (Fig. 23) the center temperature is zero, implying the algorithm likely did not converge and execution stopped early. This is expected: as noted earlier, convergence requires on the order of a million iterations, whereas here the limit is only 256 iterations.

```
Jacobi X 2048 Y 2048 Px 1 Py 1 Iter 256 CompTime 8.417019 MessTime 0.000289 Total 8.417435 midpoint 0.000
Jacobi X 2048 Y 2048 Px 2 Py 1 Iter 256 CompTime 4.212652 MessTime 0.107972 Total 4.319512 midpoint 0.000
Jacobi X 2048 Y 2048 Px 2 Py 2 Iter 256 CompTime 2.111995 MessTime 0.251033 Total 2.360420 midpoint 0.000
Jacobi X 2048 Y 2048 Px 4 Py 2 Iter 256 CompTime 1.052504 MessTime 0.336826 Total 1.384202 midpoint 0.000
Jacobi X 2048 Y 2048 Px 4 Py 4 Iter 256 CompTime 0.663224 MessTime 0.364288 Total 1.012555 midpoint 0.000
Jacobi X 2048 Y 2048 Px 8 Py 4 Iter 256 CompTime 0.290805 MessTime 0.373848 Total 0.628165 midpoint 0.000
Jacobi X 2048 Y 2048 Px 8 Py 8 Iter 256 CompTime 0.053300 MessTime 0.335123 Total 0.385897 midpoint 0.000
Jacobi X 4096 Y 4096 Px 1 Py 1 Iter 256 CompTime 33.634679 MessTime 0.000419 Total 33.635258 midpoint 0.000
Jacobi X 4096 Y 4096 Px 2 Py 1 Iter 256 CompTime 16.832444 MessTime 0.152904 Total 16.985511 midpoint 0.000
Jacobi X 4096 Y 4096 Px 2 Py 2 Iter 256 CompTime 8.436994 MessTime 0.731161 Total 9.168280 midpoint 0.000
Jacobi X 4096 Y 4096 Px 4 Py 2 Iter 256 CompTime 4.224044 MessTime 1.028663 Total 5.246204 midpoint 0.000
Jacobi X 4096 Y 4096 Px 4 Py 4 Iter 256 CompTime 3.039436 MessTime 1.134723 Total 4.155380 midpoint 0.000
Jacobi X 4096 Y 4096 Px 8 Py 4 Iter 256 CompTime 2.785139 MessTime 1.475225 Total 3.954300 midpoint 0.000
Jacobi X 4096 Y 4096 Px 8 Py 8 Iter 256 CompTime 1.979021 MessTime 2.434826 Total 3.208476 midpoint 0.000
Jacobi X 6144 Y 6144 Px 1 Py 1 Iter 256 CompTime 75.709465 MessTime 0.000452 Total 75.710063 midpoint 0.000
Jacobi X 6144 Y 6144 Px 2 Py 1 Iter 256 CompTime 37.881797 MessTime 0.216098 Total 38.083830 midpoint 0.000
Jacobi X 6144 Y 6144 Px 2 Py 2 Iter 256 CompTime 18.976311 MessTime 1.516096 Total 20.477842 midpoint 0.000
Jacobi X 6144 Y 6144 Px 4 Py 2 Iter 256 CompTime 9.496531 MessTime 2.161902 Total 11.645876 midpoint 0.000
Jacobi X 6144 Y 6144 Px 4 Py 4 Iter 256 CompTime 6.850163 MessTime 2.372009 Total 9.190001 midpoint 0.000
Jacobi X 6144 Y 6144 Px 8 Py 4 Iter 256 CompTime 6.443347 MessTime 3.461622 Total 8.937407 midpoint 0.000
Jacobi X 6144 Y 6144 Px 8 Py 8 Iter 256 CompTime 4.797882 MessTime 5.542366 Total 7.580173 midpoint 0.000
```

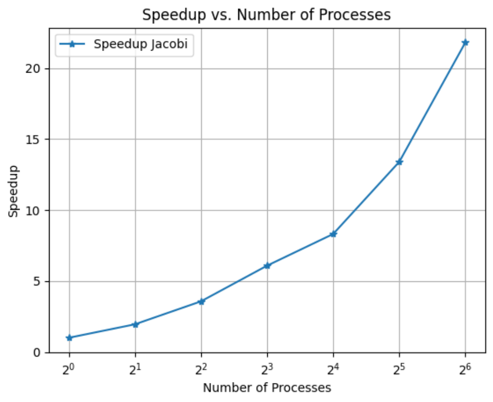
Figure 23: Output Data Jacobi for various grid sizes.

For the 2048 × 2048 grid, the total runtime scales satisfactorily and compute time scales very well, while communication time remains roughly constant (see 24a). Notably, at 64 threads the time is more than halved compared to 32 threads (25a). This may be because the small grid fits well across processor caches. Beyond compute time, the rest (overhead, setup, inter-process communication) is relatively stable due to the smaller data volume.

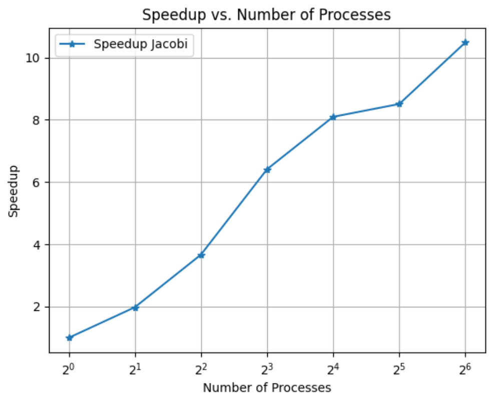
For 4096×4096, compute time scales perfectly up to 8 processes (Fig. 24b), and continues to scale (though non-ideally) up to 64 processes. Non-ideal scaling likely relates to cache capacity being insufficient for the larger problem. This also appears in the rising communication cost with more processes (25b), since more processes imply higher communication costs when data no longer fit in cache.

Similarly, for 6144×6144, the hardware/setup does not favor this grid size; increasing process count makes communication significantly more difficult and costly.

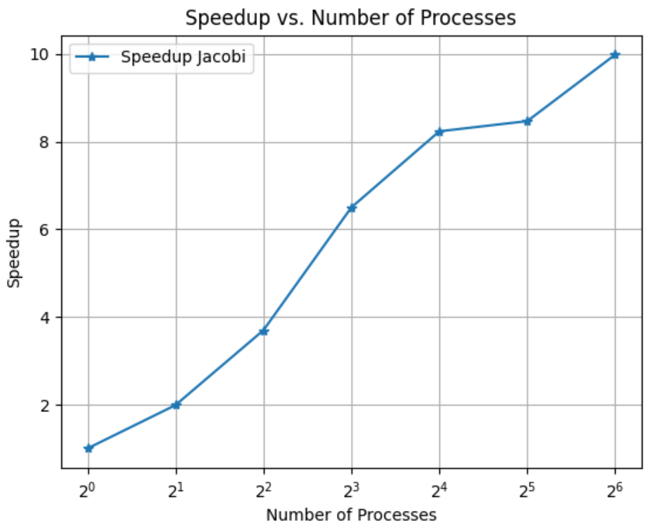
Although behaviors for 4096^2 and 6144^2 look similar, the absolute communication time does scale up, from about half a second at the smallest to nearly six seconds at the largest problem.



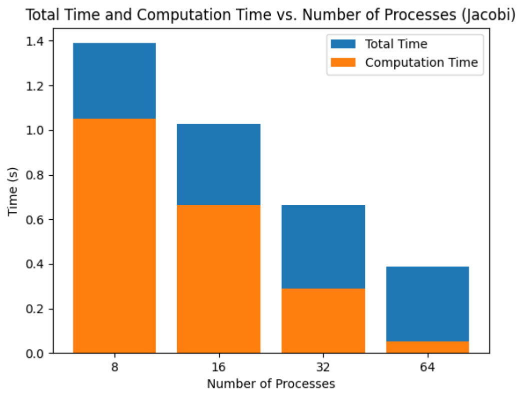
(a) speedup 2048×2048



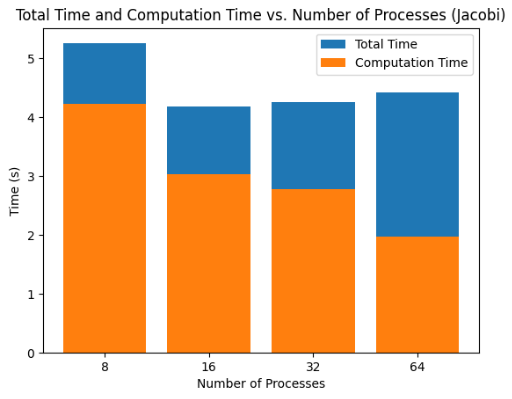
(b) speedup 4096×4096



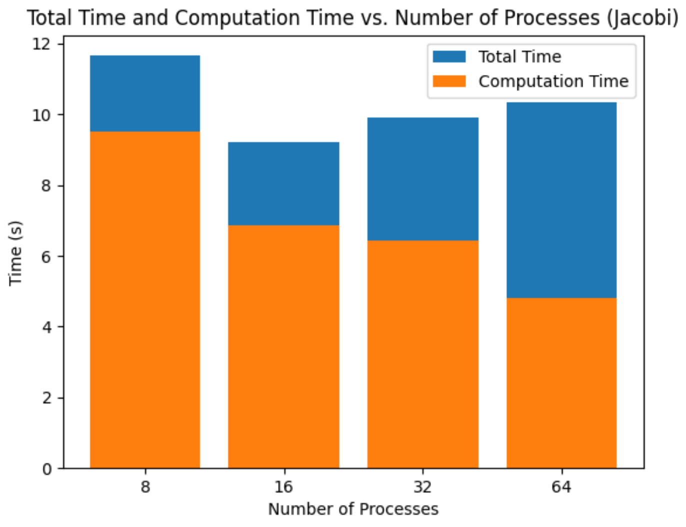
(c) speedup 6144×6144



(a) Runtime breakdown 2048 × 2048



(b) Runtime breakdown 4096 × 4096



(c) Runtime breakdown 6144 × 6144

17 Appendix

Listing 14: Kmeans MPI

```c
...
do {
    // before each loop, set cluster data to 0
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++)
            rank_newClusters[i*numCoords + j] = 0.0;
        rank_newClusterSize[i] = 0;
    }
    rank_delta = 0.0;
    for (i=0; i<numObjs; i++) {
        // find the array index of nearest cluster center
        index=find_nearest_cluster(numClusters,numCoords,&objects[i*numCoords],clusters);
        // if membership changes, increase rank_delta by 1
        if (membership[i] != index)
            rank_delta += 1.0;
        // assign the membership to object i
        membership[i] = index;
        // update new cluster centers : sum of objects located within
        rank_newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            rank_newClusters[index*numCoords + j] += objects[i*numCoords + j];
    }
    /*
     * TODO: Perform reduction of cluster data (rank_newClusters, rank_newClusterSize) from
    local arrays to shared.
     */
    MPI_Allreduce(rank_newClusters, newClusters, numClusters*numCoords, MPI_DOUBLE, MPI_SUM,
     MPI_COMM_WORLD);
    MPI_Allreduce(rank_newClusterSize, newClusterSize, numClusters, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD);
    // 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
    ];
            }
        }
    }
    /*
     * TODO: Perform reduction from rank_delta variable to delta variable, that will be used
     for convergence check.
     */
    MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    // Get fraction of objects whose membership changed during this loop. This is used as a
    convergence criterion.
    delta /= numObjs;
    loop++;
} while (delta > threshold && loop < loop_threshold);
...
```

Listing 15: Jacobi MPI setup

```c
...
//----Ensure that u_current and u_previous are both initialized----//
init2d(u_current,local[0]+2,local[1]+2);
init2d(u_previous,local[0]+2,local[1]+2);
MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block, &(u_previous[1][1]), 1,
     local_block, 0, MPI_COMM_WORLD);
```

```
MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block, &(u_current[1][1]),  1,
     local_block, 0, MPI_COMM_WORLD);

if (rank==0) free2d(U);
MPI_Datatype row;
MPI_Type_contiguous(local[1], MPI_DOUBLE, &dummy);
MPI_Type_create_resized(dummy, 0, sizeof(double), &row);
MPI_Type_commit(&row);
MPI_Datatype column;
MPI_Type_vector(local[0], 1, local[1]+2, MPI_DOUBLE, &dummy);
MPI_Type_create_resized(dummy, 0, sizeof(double), &column);
MPI_Type_commit(&column);

//----Find the 4 neighbors with which a process exchanges messages----//
int north, south, east, west;

// the processes "in the middle" of the grid have 4 neighbors
// the processes on the edges have 3 neighbors
// the processes on the corners have 2 neighbors

MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);
...
```

Listing 16: Jacobi MPI ranges

```
...
//---Define the iteration ranges per process-----//
int i_min,i_max,j_min,j_max;
/*Three types of ranges:
-internal processes
-boundary processes
-boundary processes and padded global array
*/
// these are the ranges that will be used later for the double for loop
i_min = (north == MPI_PROC_NULL) ? 2 : 1;
j_min = (west == MPI_PROC_NULL) ? 2 : 1;
i_max = (south == MPI_PROC_NULL) ? local[0] - (global_padded[0] - global[0] + 1): local[0];
j_max = (east == MPI_PROC_NULL) ? local[1]- (global_padded[1] - global[1] + 1): local[1];
// because of the ghost cells, a 'normal' 'internal' process would start from its
// first row and column and end just before its last row and column
// if the process is on some edge, then the boundaries are shifted by one row (+) or column
    (-)
...
```

Listing 17: Jacobi MPI core

```
//----Computational core----//
gettimeofday(&tts, NULL);
#ifdef TEST_CONV
for (t=0;t<T && !global_converged;t++) {
#endif
#ifndef TEST_CONV
#undef T
#define T 256
for (t=0;t<T;t++) {
#endif
    swap = u_previous;
    u_previous = u_current;
    u_current = swap;
    /*Compute and Communicate*/
    int count = 0;
    MPI_Request req[8];
    MPI_Status stat[8];
    gettimeofday(&tms,NULL);
    if (north != MPI_PROC_NULL)
    {
        MPI_Isend(&u_previous[1][1], 1, row, north, 0, MPI_COMM_WORLD, &req[count++]);
        MPI_Irecv(&u_previous[0][1], 1, row, north, 0, MPI_COMM_WORLD, &req[count++]);
    }
    if (south != MPI_PROC_NULL)
    {
        MPI_Isend(&u_previous[local[0]][1], 1, row,south, 0, MPI_COMM_WORLD, &req[count++]);
        MPI_Irecv(&u_previous[local[0]+1][1],1,row,south, 0, MPI_COMM_WORLD, &req[count++]);
    }
    if (east != MPI_PROC_NULL)
    {
        MPI_Isend(&u_previous[1][local[1]], 1,column,east,0,MPI_COMM_WORLD, &req[count++]);
        MPI_Irecv(&u_previous[1][local[1]+1],1,column,east,0,MPI_COMM_WORLD, &req[count++]);
    }
    if (west != MPI_PROC_NULL)
    {
        MPI_Isend(&u_previous[1][1], 1, column, west, 0, MPI_COMM_WORLD, &req[count++]);
        MPI_Irecv(&u_previous[1][0], 1, column, west, 0, MPI_COMM_WORLD, &req[count++]);
    }
    // Isend / Irecv do not block.
    MPI_Waitall(count, req, stat);
    gettimeofday(&tmf,NULL);
    tmsg += (tmf.tv_sec-tms.tv_sec)+(tmf.tv_usec-tms.tv_usec)*0.000001;
    gettimeofday(&tcs,NULL);
    for (i = i_min; i <= i_max; i++) {
        for (j = j_min; j <= j_max; j++) {
            u_current[i][j] = 0.25 * (u_previous[i-1][j] + u_previous[i+1][j]
                                   +  u_previous[i][j-1] + u_previous[i][j+1]);
        }
    }
    gettimeofday(&tcf,NULL);
    tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
    /*Add appropriate timers for computation*/
#ifdef TEST_CONV
    if (t%C==0) {
    /*Test convergence*/
        converged = converge(u_current, u_previous, i_min, i_max, j_min, j_max);
        MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    }
#endif
}
```