COL380: Lab I Report

Arneish Prateek [2014CH10786]

February 10, 2019

Design Decisions and Parallelization Strategies

POSIX threads-based parallelization in C

- For pthreads-based parallelization of the *k-means* algorithm, a **data-parallelism** approach was adopted. The *N* points were equally split among the *num_threads* number of threads spawned using pthread_create (in case of an imperfect data split among threads, the remainder points are allotted to the last thread)
- **Key Points in the Parallelization Algorithm:**
    - ➢ **Initialization:** The first *K* data points are chosen as the initial centroids
    - ➢ **Data-parallelism:** Each thread assigned *N/num_threads* number of points
    - ➢ **Thread function:** Each thread runs a loop (with a *max_iter* value of 100), in every iteration of which it computes the closest cluster centroid for every point assigned to it, and then assigns the point to that cluster. After every point is assigned to a cluster, the global (shared) cluster centroids are updated with the values computed for their coordinates from the points that were assigned to that cluster. This is again an instance of *data-parallelism*.
    - ➢ **Stopping-condition:** The *L2-norm* is computed for every cluster centroid coordinates by comparing against corresponding values in the previous iteration. The norms are then summed and compared against the *threshold* value, chosen to be *1e-6*. All threads break from the iterations loop as soon as the *delta* value goes below the *threshold*.
- **Mutual-exclusion and Critical section:** The first *thread-synchronization* strategy which ensures that updates to the shared cluster *centroid* array are undertaken by each thread only after acquiring a mutex or a spinlock (if compilation is done with the *-DUSE_SPINLOCK* flag). This is essential to avoid data race due to different threads attempting to modify the shared array.
- **Barriers:** After every iteration, barriers are used for thread synchronization so that all threads view the same updated value of the *centroids* array. Two instances of barriers are used. The first one for synchronization of the *centroids* update, and the second one following the first to synchronize the value of *delta,* again a shared variable among all threads.

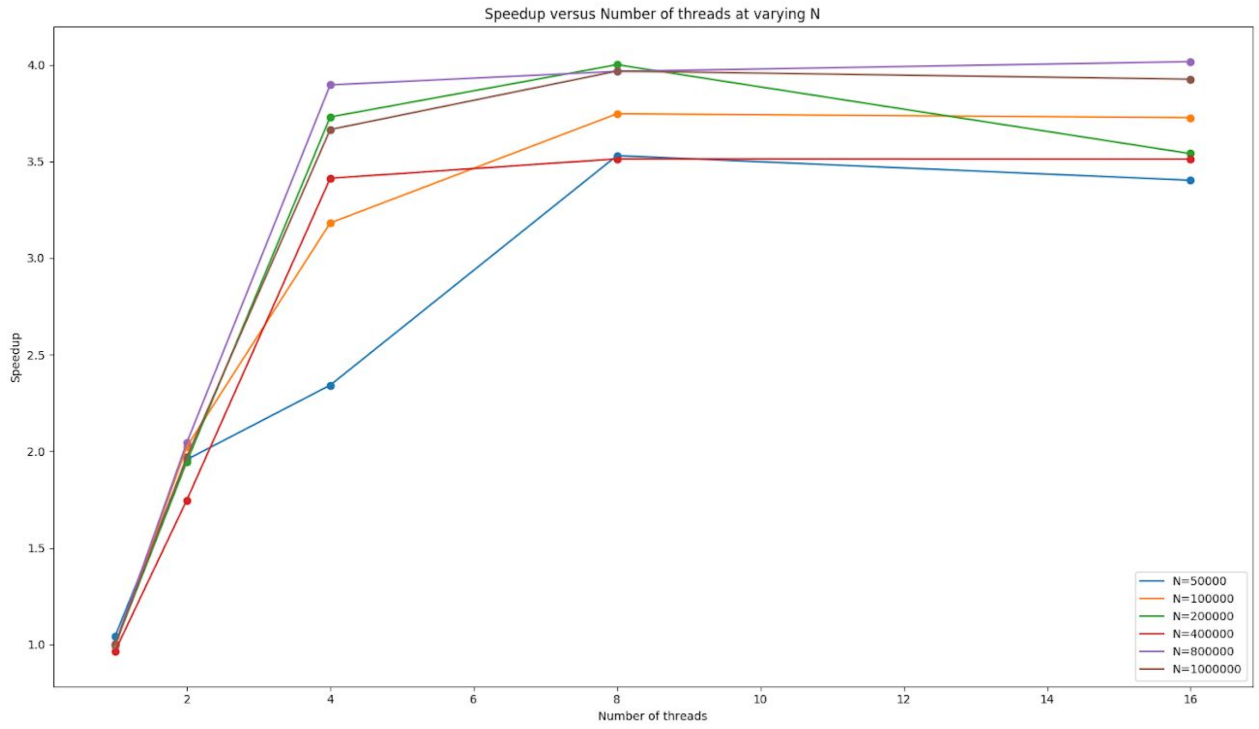
- **False-sharing:** To minimize false sharing, shared variable updates for the *centroids* array are minimized. Moreover, local variables are used in wherever feasible (for instance in *delta* value computations).
- **Load-balancing:** The nature of parallelization probably does not require an explicit load-balancing strategy since its an **SPMD**-based parallelization scheme. One obvious source of unbalanced load would be the case of a slightly larger number of points (bounded by *num_threads*) processed by the last thread.

OpenMP parallelization in C

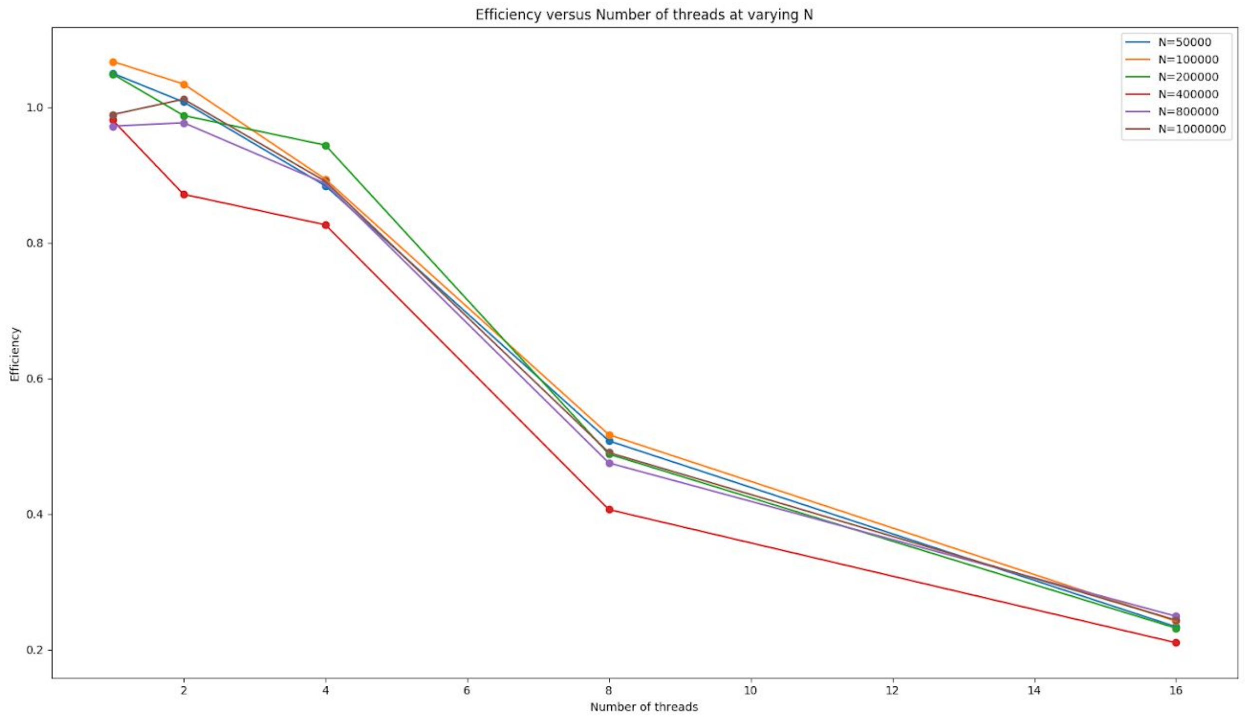
Identical **data-parallelism** strategy adopted for OpenMP-based parallelization. Thread synchronization is undertaken using *pragma omp critical* construct for mutual exclusion and *pragma omp barrier* for progress synchronization.

Performance Analysis

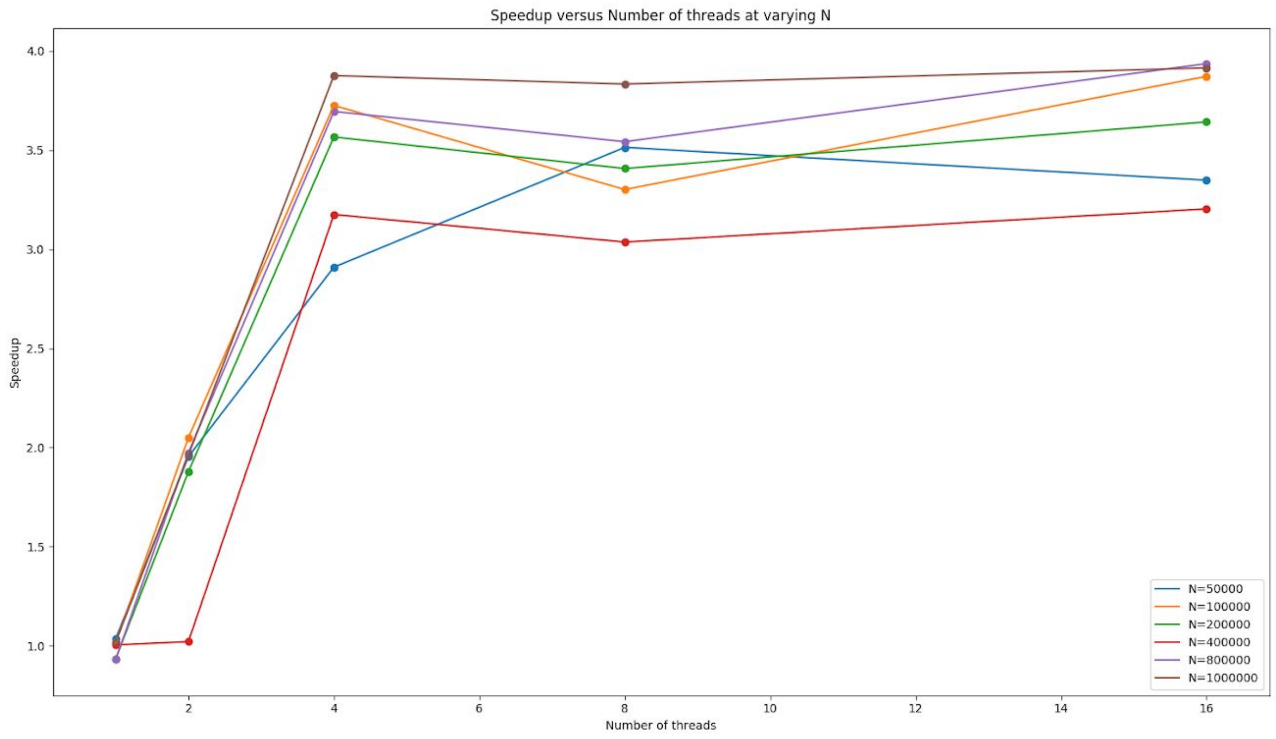
A. Speedup with *Pthreads*



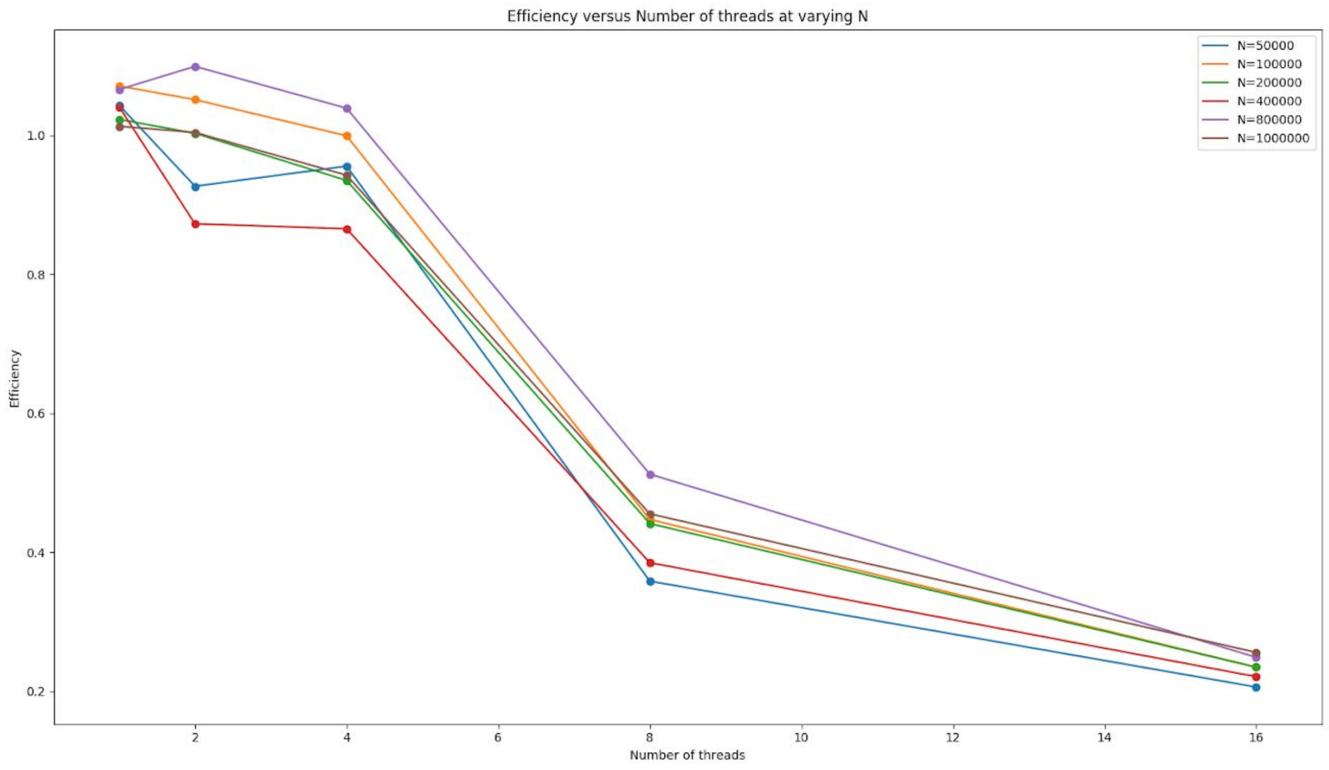
B. Efficiency with *Pthreads*



C. Speedup with *OpenMP*



D. Efficiency with *OpenMP*



Discussions

- A quadcore system (4 cores, 2 threads per core) was used for performing the above experiments. The datasets are generated by only varying the *total_points* parameter. Plots *A* and *B* demonstrate how speedup and efficiency changes for a Pthreads-based implementation, while *C* and *D* demonstrate the same for OpenMP-based implementation.
- *Pthreads:* The **speedup curve** obtained is as expected from the **Amdahl's law.** The speedup increases (with diminishing marginal returns) with the increase in *num_threads* but saturates at 8 threads. In fact for a dataset with 200,000 points, the speedup actually decreases on using 16 threads due to increased **overheads**. Among the speedup curves for different-sized datasets, the **best speedup** at *num_threads = 8* with an approximate value of 4 is observed for datasets with sizes **200.000, 800.000 and 1 million**. The **larger-sized datasets** are expected to have **higher speedups** at a given *num_threads*. The worst speedup (approximate value of 3.5) is recorded for the smallest dataset - again, as expected. The **efficiency curve** obtained also decreases with an increase in the number of threads, as expected. In the particular instance run, the efficiency was found lowest for 400,000-sized dataset at 8 threads, with the best value for 100,000-sized. The efficiency at *num_threads = 1* goes above 1.0 in certain runs due to superlinear speedups perhaps due to out-of-control system latencies during particular executions.

- *OpenMP:* Similarly for OpenMP-based parallelization, the **best speedups** are obtained for the **larger-sized threads** (sizes 800,000 and 1 million). The worst performance is for the dataset-sized 400,000. Again, the speedup curves saturate at *num_threads = 8.* Again, **superlinear speedups** are observed in some runs at single-threaded run due to out-of-control system execution properties and/or latencies. The **efficiency** curves at 8 threads again demonstrate **best efficiency at the larger-sized datasets** (800,000 and 1 million) and the **worst-performance at smaller-sized ones** (50000). The efficiency curves decrease for every instance run with an increase in the number of threads. This is again due to the diminishing returns property on including an extra thread - a consequence of the **Amdahl's Law.**
- Among OpenMP and Pthreads-based implementations, the relative speedups value are very similar. Usually OpenMP-based implementations performed slightly better than Pthreads-based ones at a higher number of threads.