DS 221 - Introduction to Scalable Systems
Parallelization of KMeans Clustering using OpenMP

Shriram R.
M Tech (CDS)
06-02-01-10-51-18-1-15763

November 27, 2018

1 Introduction

KMeans Clustering algorithm has been parallelized using OpenMP and the speedup for different thread counts and schedule configurations were observed through experiments. The following sections cover the methodology, experimental setup and results in detail.

2 Methodology

The algorithm runs for a fixed number of iterations (currently 100) for convergence of centroids. The following steps are followed in each iteration:

1. A parallel region with given no. of threads is started with the points, centroid, label and temp array in shared mode. The temp array is used to store updated centroid info for each thread:

    (a) The set of points is distributed among the threads using either static or dynamic chunks.
    (b) Each thread will assign its points to their nearest centroid by computing the euclidean distance and updates the corresponding entry in the label array
    (c) Each thread also updates the temp array with its local cluster size and centroid values

2. Global centroid information is updated by aggregating the thread level centroids computed in the previous step and is used for the next iteration

The above methodology is illustrated in the code snippet in the appendix.

3 Experimental Setup

Experiments were run on a single compute node having 8 core AMD Opteron 3380 processor and 32GB RAM in the Turing compute cluster. Batch job was created using a shell script and executed using PBS. For each thread count, two experiments, one with static and other with dynamic chunk sizes were performed.

The time taken to run the KMeans clustering was measured using C library functions and was printed to stdout along with specified output. Each experiment was repeated 20 times consecutively and then the average was computed for the plots and analysis.

In each experiment, both sequential and parallel versions of code were executed so as to remove bias from system load. The time measured is only for the algorithm and excludes any input read and output writes in the program.

4   Results

The experimental results are tabulated and plotted below,

Schedule	Threads	Sequential (Avg.) (s)	Parallel (Avg.) (s)	Speedup
Static	4	0.3358	0.1943	1.73
Static	8	0.3361	0.1565	2.15
Static	16	0.3349	0.1932	1.73
Dynamic	4	0.3450	0.1990	1.73
Dynamic	8	0.3337	0.1485	2.25
Dynamic	16	0.3435	0.1472	2.33

20 Clusters
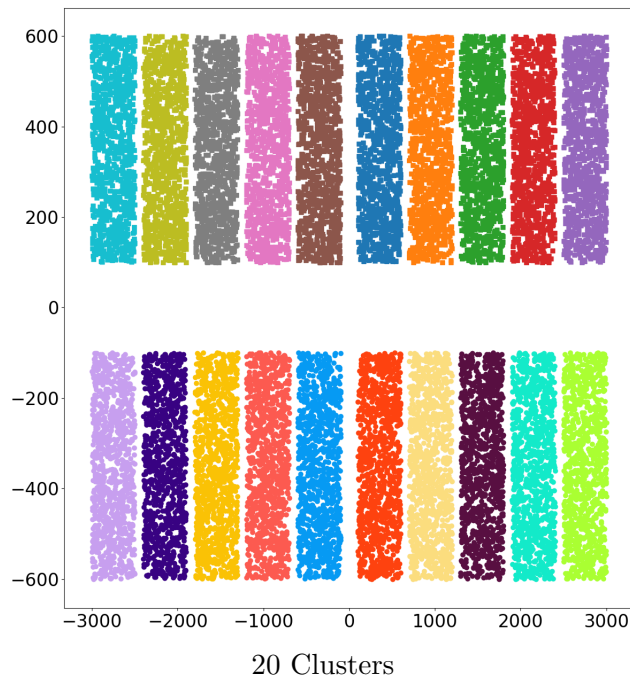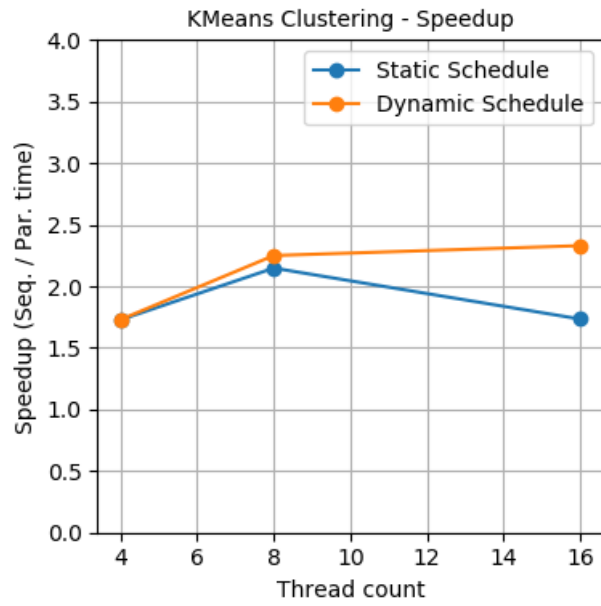
5 Observations

It can be observed that the speedup increases with number of threads upto 8 and then decreases in the case of static scheduling. This is because a single node can run only maximum of 8 threads in parallel. Any additional increase in the thread count would result in more context switches thereby reducing the speedup.

In the dynamic scheduling, the speedup increases even for 16 threads since the iterations from waiting threads can be distributed to the threads in execution thereby the computation does not have to wait for CPU to be available. It can be observed that dynamic scheduling was more efficient than static.

The speedup is less than the thread count as the program contains both serial and parallel regions. In addition, there is a overhead of creating and managing threads in each iteration which reduces the speedup further.

6 References

1. https://en.cppreference.com/w/c

2. DS 221 Course lecture notes

3. https://computing.llnl.gov/tutorials/openMP/

4. http://www.arc.ox.ac.uk/content/pbs-job-scheduler

5. https://www.dartmouth.edu/ rc/classes/intro_openmp/print_pages.shtml

7 Appendix - Code

```
for (int i = 0; i < ITER; i++)
{
    // Parallel region
    #pragma omp parallel num_threads(THREADS) shared(pts, label, centroid, temp)
    {
        float min_dist, dist;
        int tid, min_label;
        tid = omp_get_thread_num();

        // Reset temp
        for (int k = 0; k < CLUSTERS; k++)
        {
            temp[tid][k][0] = 0;
            temp[tid][k][1] = 0;
            temp[tid][k][2] = 0;
        }
```

```
        // Assign Labels for all points
        #pragma omp for schedule(static, chunk)
        for (int j = 0; j < SIZE; j++)
        {
            min_dist = FLT_MAX;
            for (int k = 0; k < CLUSTERS; k++)
            {
                dist = (pts[j][0] - centroid[k][0]) * (pts[j][0] - centroid[k][0]) +
                (pts[j][1] - centroid[k][1]) * (pts[j][1] - centroid[k][1]);
                if (dist < min_dist)
                {
                    min_dist = dist;
                    min_label = k;
                }
            }
            label[j] = min_label;
            temp[tid][min_label][0] += pts[j][0];
            temp[tid][min_label][1] += pts[j][1];
            temp[tid][min_label][2] += 1;
        }
    }
    // End of parallel region

    // Collect temp
    for (int j = 1; j < THREADS; j++)
    {
        for (int k = 0; k < CLUSTERS; k++)
        {
            temp[0][k][0] += temp[j][k][0];
            temp[0][k][1] += temp[j][k][1];
            temp[0][k][2] += temp[j][k][2];
        }
    }

    // Compute and update centroid
    for (int k = 0; k < CLUSTERS; k++)
    {
        if (temp[0][k][2] == 0)
            continue;
        centroid[k][0] = temp[0][k][0] / temp[0][k][2];
        centroid[k][1] = temp[0][k][1] / temp[0][k][2];
    }
}
```

————End of Report————