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 23, 2018

1 Introduction

KMeans Clustering algorithm has been parallelized using OpenMP and the speedup for different thread count 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 no. 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 respective nearest centroid by computing euclidean distance and update the corresponding entry in label array
   - (c) The threads also update the temp array with its local cluster size and centroid value
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 appendix A.

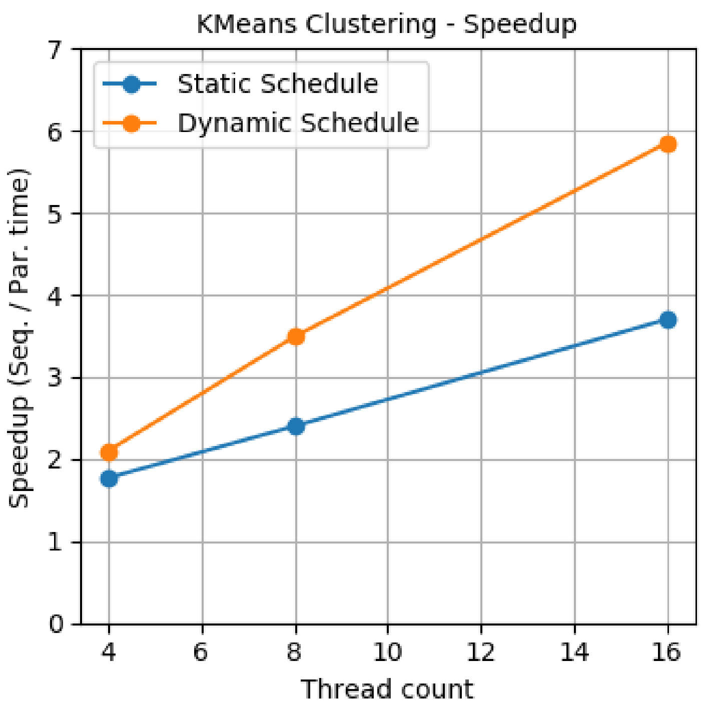
3 Experimental Setup

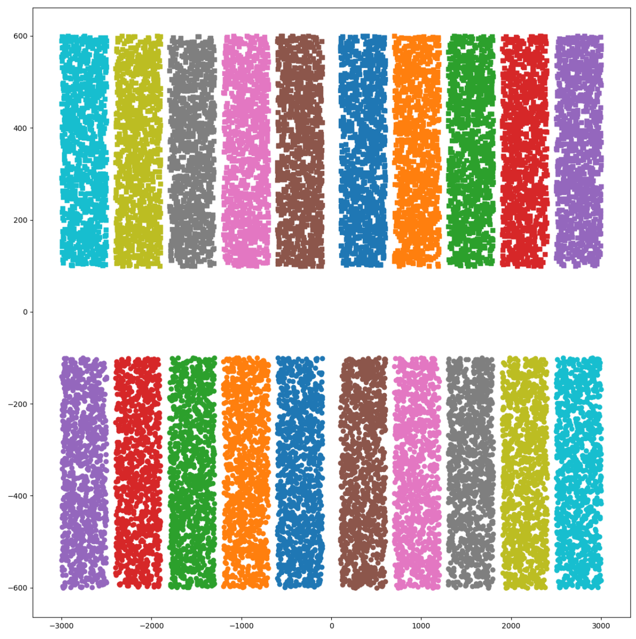
The Turing compute cluster having 24 nodes with 8 cores each was used to run the experiments. Batch job was created and executed using PBS. For each thread count, two experiments (i.e) one with static and dynamic chunk sizes were performed.

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

4 Results

Schedule	Threads	Sequential (Avg.) (s)	Parallel (Avg.) (s)	Speedup
Static	4	0.36895	0.20875	1.77
Static	8	0.3715	0.15455	2.40
Static	16	0.38465	0.10405	3.70
Dynamic	4	0.37980	0.10150	2.10
Dynamic	8	0.36990	0.10600	3.49
Dynamic	16	0.37090	0.06345	5.85





20 Clusters

5 Observations

It can be observed that the speedup increases with no. of threads. Also, the increase is not strictly linear as seen in the plot. Dynamic scheduling was more efficient than static due to its excellent cache and balance properties. There is a considerable amount of overhead in creating the threads for each iteration and so the speedup for P threads is quites less than P.

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 A

```
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];
}
}
```