Introduction to Parallel Computing

Fall Semester 2024

Assignment # 1

Po Yu Lai
email:lai00177@umn.edu

1 Parallelization Method

1.1 serial code

1. Select the initial K medoids.
   (a) For reproducibility, use points 0, 1, ..., K-1.
2. Assign each data point to the closest medroid (measured via Euclidean distance).
3. While not converged:
   (a) In each cluster, make the point that minimizes the average Euclidean distance to all other points in the cluster the (new) medoid.
   (b) Assign all data points to the closest medroid as measured via Euclidean distance.

1.2 openmp

The first part of the parallel strategy is using a matrix to store the distance between each node. Even though it will need large computation in the beginning, it can save much time for the later part, especially for the higher dimension dataset. With the method, it can further be speedup with openmp, assigning $N_{nodes}/N_{threads}$ points to each thread and finding out all the distance between assigned points. The biggest drawback is the memory issue, it will crash on larger dataset. In this case, please modify my USE_MATRIX macro from 1 to 0 in the code. After changing it, my code will compute distance every time instead of storing it in a matrix.

The second part is to parallel the process of assigning nodes to closet medoids. This simply distribute nodes to each threads and look up which medroid is the closest. After that, generate cluster list for each medroid. To prevent racing situation, it first allow each thread write in their private list, and then update with critical/mutex to the global list.

The last part is to parallel the process of picking the points with smallest summation of distance between nodes inside the same cluster. The logic is similar to the other part. the only difference is that we only divided the number of nodes in cluster instead of the whole nodes. During the process of comparing and updating the node with smallest distance summation, there will also have a racing situation. Here using the same technique as second part by updating to its own private variable inside each thread and then update to global with critical/mutex.

1.3 pthread

Most of parallel strategies for pthread are similar to openmp. The only difference is to calculating the distance matrix. Pthread don't have similar function as dynamical scheduling in openmp. This will be an issue for calculating the distance matrix, since distance matrix is symmetric and we only need to calculate the upper triangle part. The later index will have less computation. By improving this issue, it is a mapping to make index jumping back and forth. The relation is shown as follow:

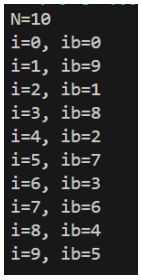

Figure 1: *the relation between original index and transferred index*

```
for(i=offset; i<offset+num; i++){
    ib=(i%2)?((i%2)*(N_points-1)-(i/2)):((i%2)*(N_points-1)+(i/2));
    for(j=ib; j<N_points; j++){
                distance_m[ib][j-ib]=0;
                for(k=0; k<Dim; k++){
                distance_m[ib][j-ib]+=(x[ib+k*N_points]-x[j+k*N_points
                    ])*(x[ib+k*N_points]-x[j+k*N_points]);
        }
        distance_m[ib][j-ib]=sqrtf(distance_m[ib][j-ib]);
    }
}
```

Fig:1 is the example for index start from 0 to 9. Ideally by this method, it can provide some load balancing.

2 Timing Results

# of clusters	1 thread	2 threads	4 threads	8 threads	16 threads	speed-up
256	32351.3419	14771.2892	7378.9433	4491.4708	2117.0062	15.28
512	32241.0617	14673.4176	7342.0537	4369.6009	2107.3767	15.30
1024	32286.9456	14676.4134	7339.6279	4084.8170	2133.4255	15.13

Table 1: *Timing results for OpenMP. Speed-up is defined as the time taken by 1 thread divided by the time-taken for 16 threads with a given number of clusters, unit:sec*

# of clusters	1 thread	2 threads	4 threads	8 threads	16 threads	speed-up
256	32346.3989	17871.9492	8750.4270	4461.5280	2267.8769	14.26
512	32441.1647	14691.5036	7344.7350	4380.6029	2231.2843	14.53
1024	32316.5694	14680.9303	7339.1276	4688.6868	2248.4854	14.37

Table 2: *Timing results for Pthread. Speed-up is defined as the time taken by 1 thread divided by the time-taken for 16 threads with a given number of cluster, unit:sec*