Assignment-3

OPENMP

Roll Number:CED18I061
Name : Ashish Choudhary

Improvements since profiling:

In the profiling experiment, we realized that there are a lot of optimizations that we can do in the code before parallelizing it. The following optimizations are done:

1. Get_neighbours function: It is called by dbscan to get the distance of each node with the other and classify them as neighbours or not.There can be n*n pairs and for k dim to get the distance it would take n*n*k computations. If we parallelize it with p threads time complexity would become O(n*n*k/p).This is the heaviest function in the whole code.

2. DFS Function and finding noise value: We can parallelize dfs and noise finding function the time complexity of this would become O(n/p) for p threads.

Just by doing these two updates, the following speed up was achieved:

1. Naive-Implementation:: Execution Time = 66.6776 for 50000 3 dimensional points

2. Optimized- Implementation: Execution time = 32.3772 for 50000 3 dimensional points

Great! We reduced the execution time by half by just making small & simple changes

Parallelizing the execution with OPEN-MP

The following 4 for loops were parallezed:

1. Get_neighbours function:

```
void get_neighbours()
{
   #pragma omp for
   for(int i = 0; i < num_pts-1; i++)
   {
       for(int j = i+1; j < num_pts; j++)
       {
           if(i == j)
           continue;
           if(sqrd_dist(i,j) <= ep)
           {
               clusters[i][siz[i]] = j;
               clusters[j][siz[j]] = i;
               siz[i]++;
               siz[j]++;
           }
       }
   }
}
```

2. DFS function:

```
void dfs(int i)
{
   vis[i] = 1;
   printf("%d ",i+1);
   #pragma omp for
   for(int a  = 0; a < siz[i] ; a++)
   {
       if(vis[clusters[i][a]] != 1)
           dfs(clusters[i][a]);
   }
}
```

3. Noise calculation:

```
   printf("NOISE :");
   #pragma omp for
   for(int i = 0; i < num_pts; i++)
   {
       if(vis[i] != 1)
           printf("%d ",i+1);
   }
   printf("\n");
```

4.Cross Linking Code:

```
int threads[] = {1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128};
int i = 0, j = 0, k = 0;
for (int thread = 0; thread < 12; thread++)
{
    omp_set_num_threads(threads[thread]);
    float startTime = omp_get_wtime();
    #pragma omp for
    for(int i = 0; i < num_pts-1; i++)
    {
        for(int j = i+1; j < num_pts; j++)
        {
            clusters[i][siz[i]] = 0;
            clusters[j][siz[j]] = 0;
        }
    }
    #pragma omp for
    for(int i = 0; i < num_pts; i++)
    {
        siz[i] = 0;
        vis[i] = 0;
    }
    dbscan();
    float endTime = omp_get_wtime();
    float execTime = endTime - startTime;
    printf("\n threads: %d -----> time taken : %f\n\n", threads[thread], execTime);
}
```

Parallel Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#include <omp.h>

double ep;
double pts[1000][50];
int clusters[1000][1000];
int siz[1000];
int minpts,dim,num_pts;
int vis[100000];
double sqrd_dist(int i,int j)
{
   double sum = 0;
   for(int k = 0; k < dim ; k++)
   {
       sum += pow(pts[i][k] - pts[j][k],2);
   }
   return sqrt(sum);
}


int is_core_node(int i)
{
   if(siz[i] >= minpts-1 )
```

```
        return 1;
    return 0;
}


void get_neighbours()
{
    #pragma omp for
    for(int i = 0; i < num_pts-1; i++)
    {
        for(int j = i+1; j < num_pts; j++)
        {
            if(i == j)
            continue;
            if(sqrd_dist(i,j) <= ep)
            {
                clusters[i][siz[i]] = j;
                clusters[j][siz[j]] = i;
                siz[i]++;
                siz[j]++;
            }
        }
    }
}
void dfs(int i)
{
    vis[i] = 1;
    printf("%d ",i+1);
```

```
    #pragma omp for
    for(int a  = 0; a < siz[i] ; a++)
    {
        if(vis[clusters[i][a]] != 1)
            dfs(clusters[i][a]);
    }
}
void dbscan()
{
    get_neighbours();

    int cnt = 0;
    for(int i = 0; i < num_pts; i++)
    {
        if(vis[i] != 1 && siz[i] >= minpts)
        {
            cnt++;
            printf("cluster %d : ",cnt);
            dfs(i);
            printf("\n");
        }

    }
    printf("NOISE :");
    #pragma omp for
    for(int i = 0; i < num_pts; i++)
    {
        if(vis[i] != 1)
```

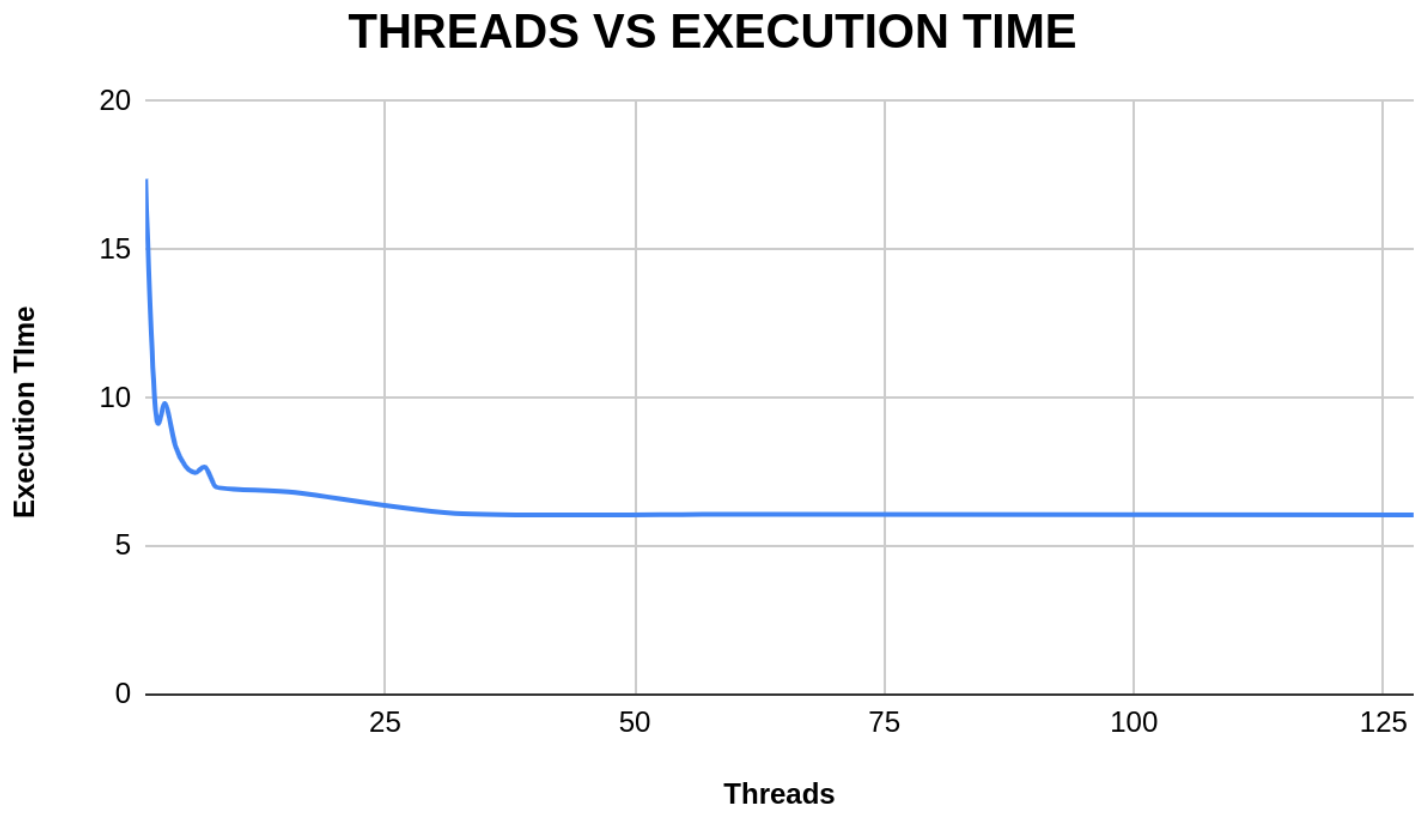
```
            printf("%d ",i+1);

    }
    printf("\n");
}



int main()
{
    printf("Enter the ep distance:");
    scanf("%lf",&ep);
    if(ep < 0)
    {
        printf("INVALID EPSILON DISTANCE");
        return 0;
    }
    printf("Enter the minimum points:");
    scanf("%d",&minpts);
    if(minpts < 1)
    {
        printf("INVALID MIN PTS");
        return 0;
    }
    printf("Enter the dimesions of the points:");
    scanf("%d",&dim);
    if(dim < 1)
    {
```

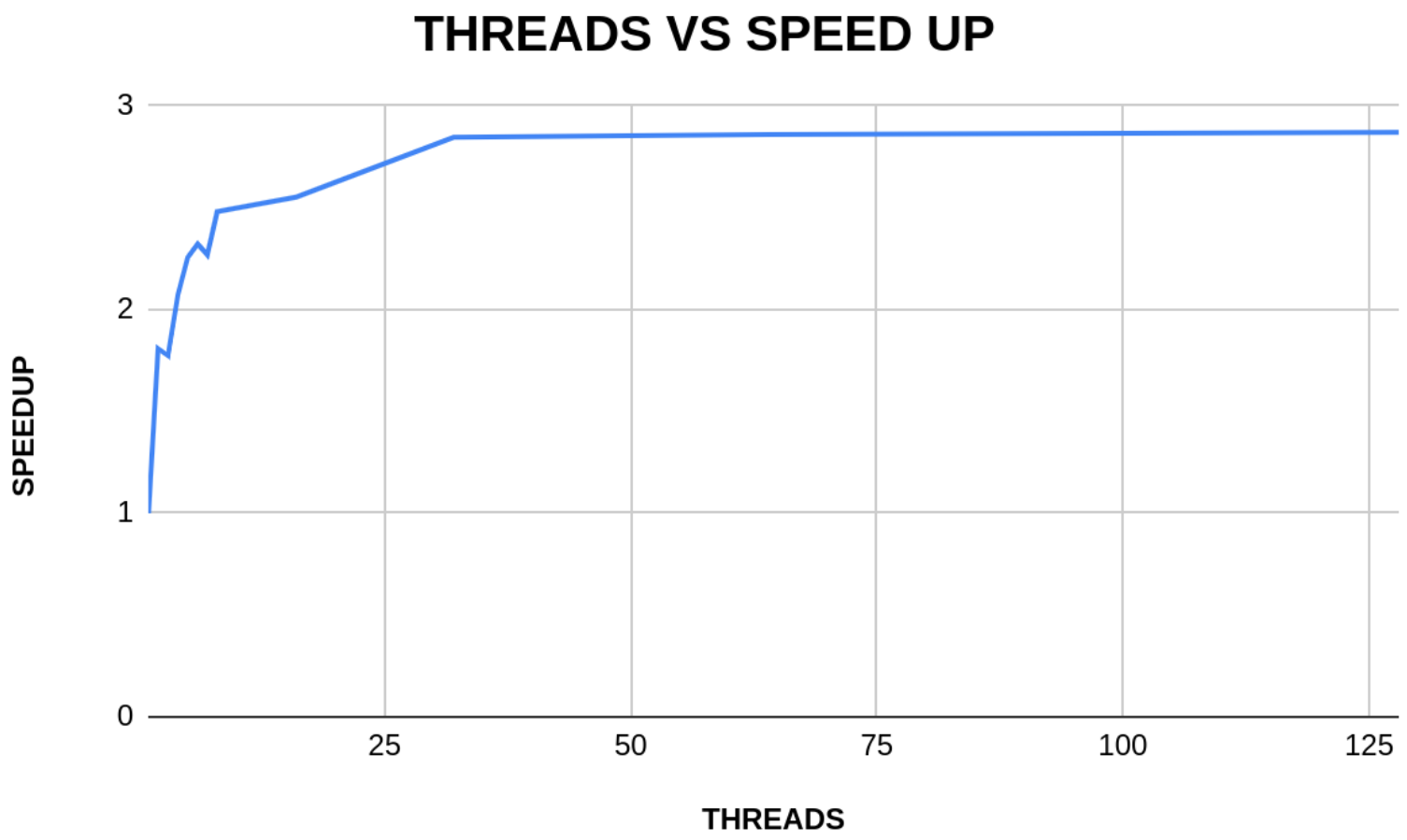
```
    printf("INVALID DIMENSIONS");
    return 0;
}
printf("Enter the number of points:");
scanf("%d",&num_pts);
if(num_pts < 1)
{
    printf("INVALID NUMBER OF PTS");
    return 0;
}
printf("Enter points:");
for(int i = 0 ; i < num_pts; i++)
{
    for(int j = 0; j < dim; j++)
    {
        scanf("%lf",&pts[i][j]);
    }
}

int threads[] = {1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128};
int i = 0, j = 0, k = 0;
for (int thread = 0; thread < 12; thread++)
{
    omp_set_num_threads(threads[thread]);
    float startTime = omp_get_wtime();
    #pragma omp for
    for(int i = 0; i < num_pts-1; i++)
    {
```

```
            for(int j = i+1; j < num_pts; j++)
            {
                clusters[i][siz[i]] = 0;
                clusters[j][siz[j]] = 0;
            }
        }
        #pragma omp for
        for(int i = 0; i < num_pts; i++)
        {
            siz[i] = 0;
            vis[i] = 0;
        }
        dbscan();
        float endTime = omp_get_wtime();
        float execTime = endTime - startTime;
        printf("\n threads: %d -----> time taken : %f\n\n", threads[thread], execTime);
    }
    return 0;
}
```

Execution Time Vs Threads & Speed-Up Vs Threads:

The following graphs were obtained after the above optimizations:




THREADS VS SPEED UP
SPEEDUP
3
2
1
0
25
50
75
100
125
THREADS

Conclusion/Inferences:

We first reduced the execution time by half by doing small optimizations in the code. Further on parallelizing it, the best performance was 8 times faster than the naive implementation. Also, it can be observed from the graph that the performance doesn’t improve after 16 threads.