DBSCAN

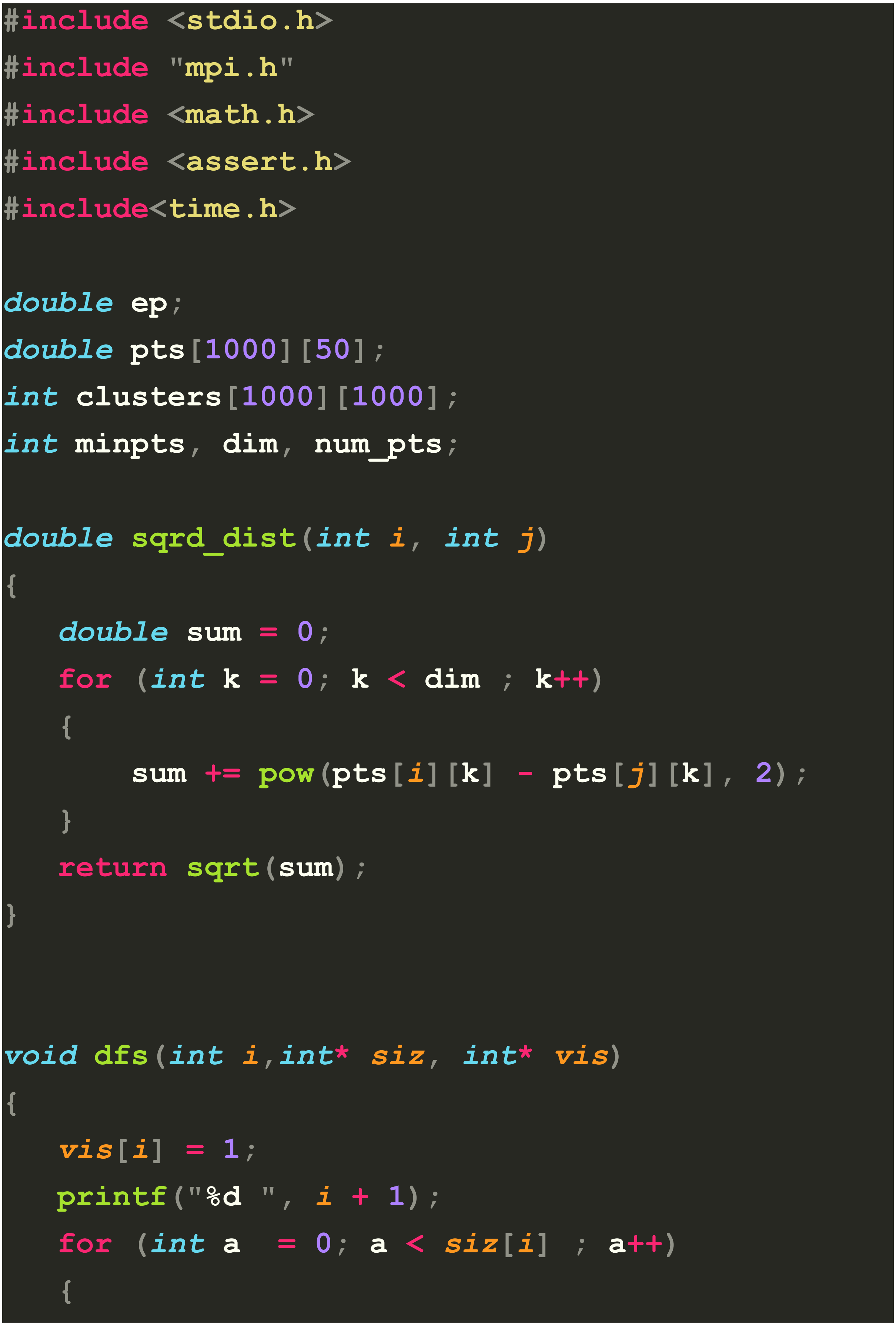
MPI Parallelization

Ashish Choudhary
CED18I061

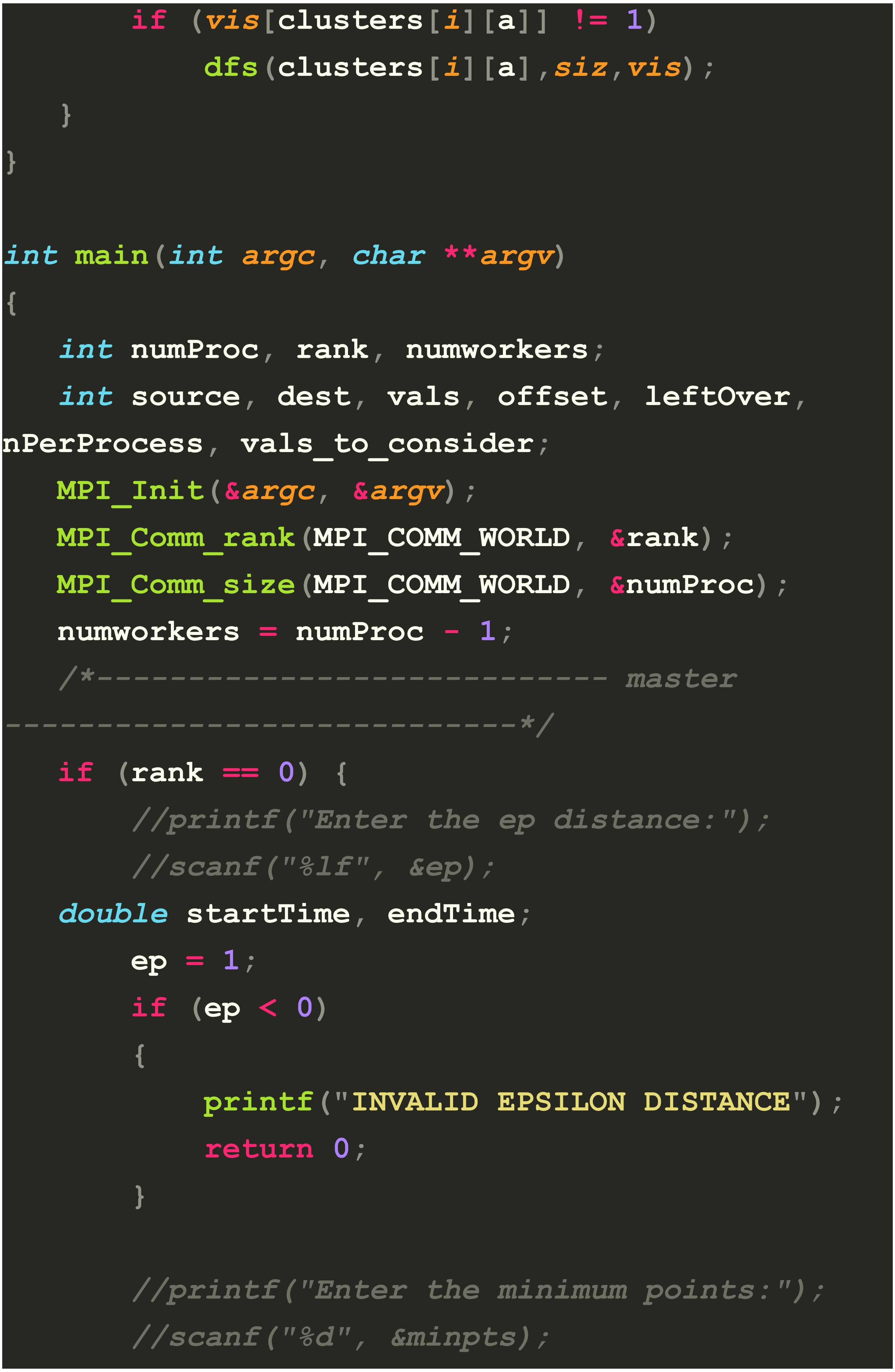
Hardware Configuration:

```
CPU NAME : Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
        Number of Sockets: : 1
        Cores per Socket : 4
        Threads per core : 2
        L1 cache size  : 256 KiB
        L2 Cache size : 1 MiB
        L3 Cache size(Shared): 8 MiB
         RAM : 16 GiB
```

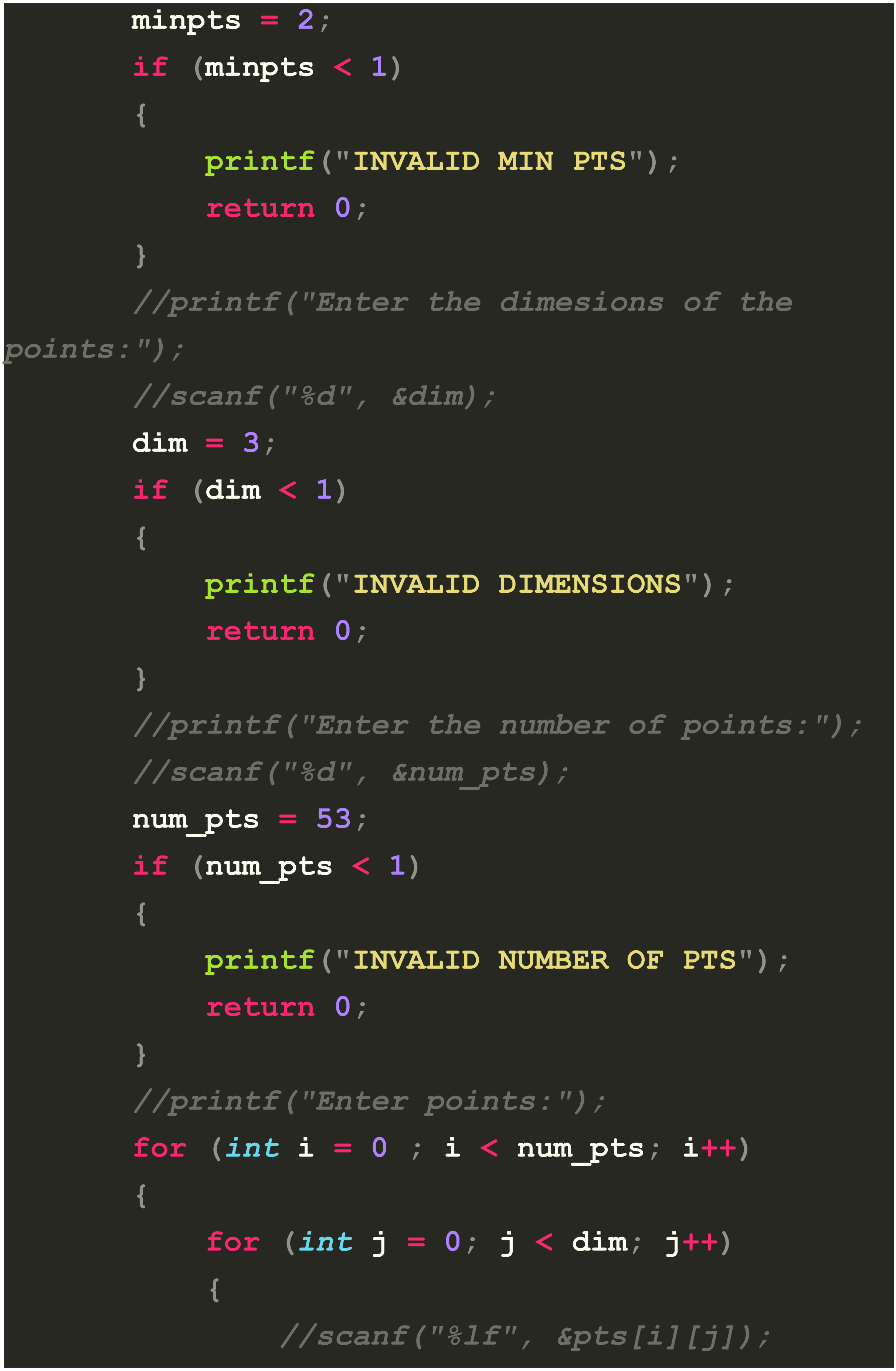
Parallel Code:



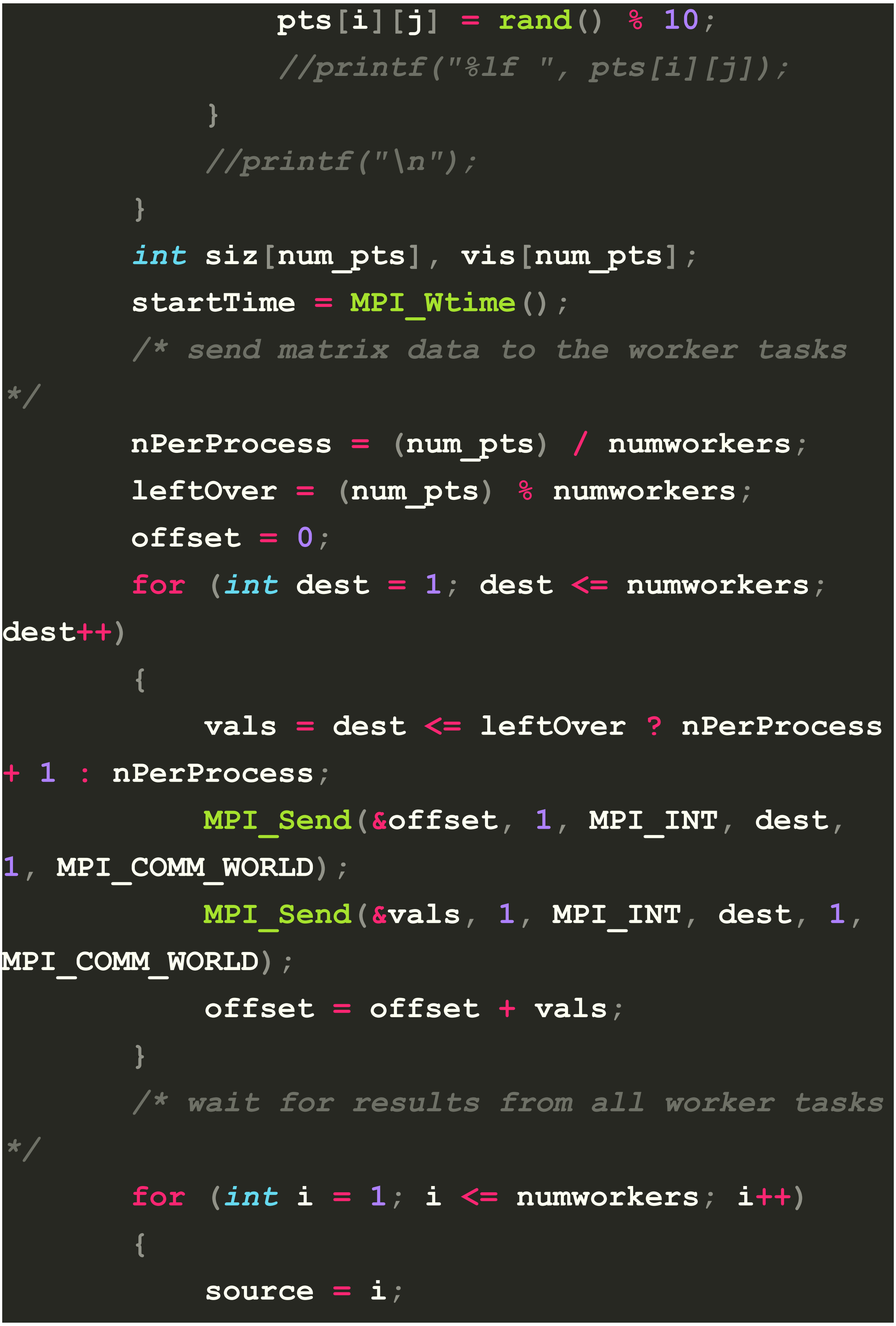
```
#include <stdio.h>
#include "mpi.h"
#include <math.h>
#include <assert.h>
#include<time.h>

double ep;
double pts[1000][50];
int clusters[1000][1000];
int minpts, dim, num_pts;

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);
}


void dfs(int i,int* siz, int* vis)
{
   vis[i] = 1;
   printf("%d ", i + 1);
   for (int a  = 0; a < siz[i] ; a++)
   {
```



```
        if (vis[clusters[i][a]] != 1)
            dfs(clusters[i][a],siz,vis);
    }
}

int main(int argc, char **argv)
{
    int numProc, rank, numworkers;
    int source, dest, vals, offset, leftOver,
nPerProcess, vals_to_consider;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProc);
    numworkers = numProc - 1;
    /*---------------------------- master
----------------------------*/
    if (rank == 0) {
        //printf("Enter the ep distance:");
        //scanf("%lf", &ep);
    double startTime, endTime;
        ep = 1;
        if (ep < 0)
        {
            printf("INVALID EPSILON DISTANCE");
            return 0;
        }

        //printf("Enter the minimum points:");
        //scanf("%d", &minpts);
```

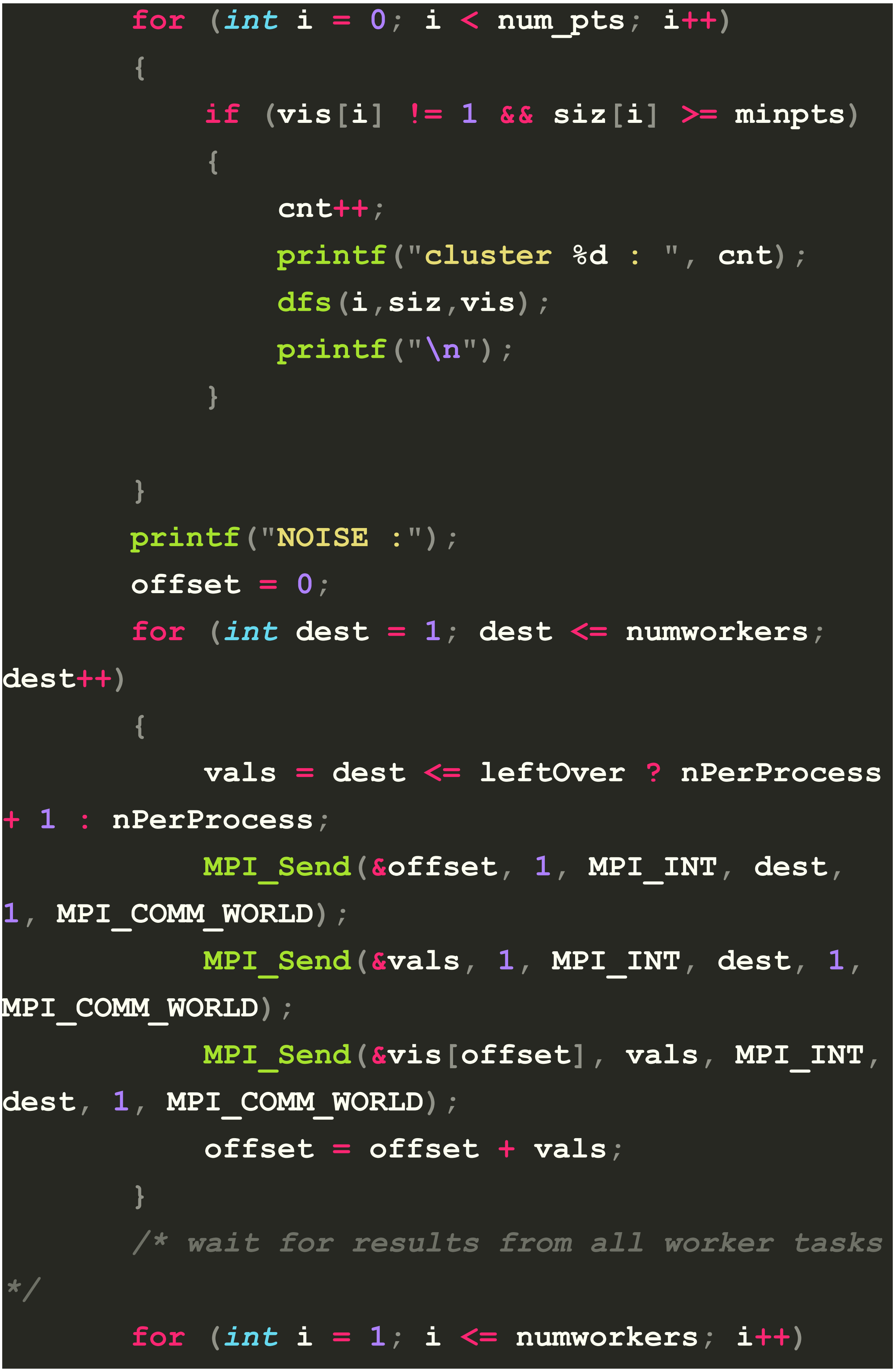


```
        minpts = 2;
        if (minpts < 1)
        {
            printf("INVALID MIN PTS");
            return 0;
        }
        //printf("Enter the dimesions of the
points:");
        //scanf("%d", &dim);
        dim = 3;
        if (dim < 1)
        {
            printf("INVALID DIMENSIONS");
            return 0;
        }
        //printf("Enter the number of points:");
        //scanf("%d", &num_pts);
        num_pts = 53;
        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]);
```

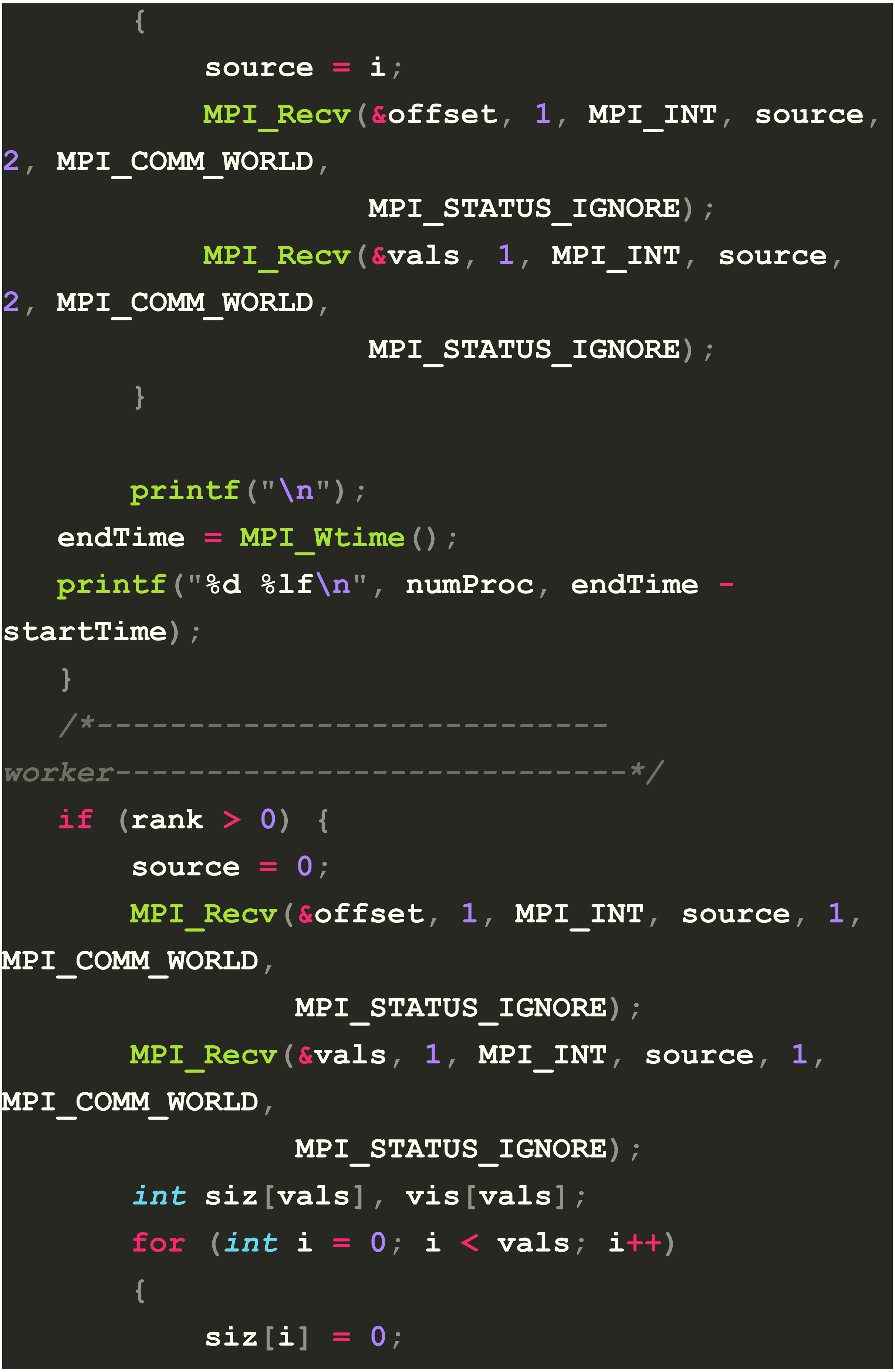


```
                pts[i][j] = rand() % 10;
                //printf("%lf ", pts[i][j]);
            }
            //printf("\n");
        }
        int siz[num_pts], vis[num_pts];
        startTime = MPI_Wtime();
        /* send matrix data to the worker tasks
*/
        nPerProcess = (num_pts) / numworkers;
        leftOver = (num_pts) % numworkers;
        offset = 0;
        for (int dest = 1; dest <= numworkers;
dest++)
        {
            vals = dest <= leftOver ? nPerProcess
+ 1 : nPerProcess;
            MPI_Send(&offset, 1, MPI_INT, dest,
1, MPI_COMM_WORLD);
            MPI_Send(&vals, 1, MPI_INT, dest, 1,
MPI_COMM_WORLD);
            offset = offset + vals;
        }
        /* wait for results from all worker tasks
*/
        for (int i = 1; i <= numworkers; i++)
        {
            source = i;
```

```
           MPI_Recv(&offset, 1, MPI_INT, source,
2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&vals, 1, MPI_INT, source,
2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&siz[offset], vals, MPI_INT,
source, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&vis[offset], vals, MPI_INT,
source, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
       }
       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]++;
               }
           }
       }
       int cnt = 0;
```

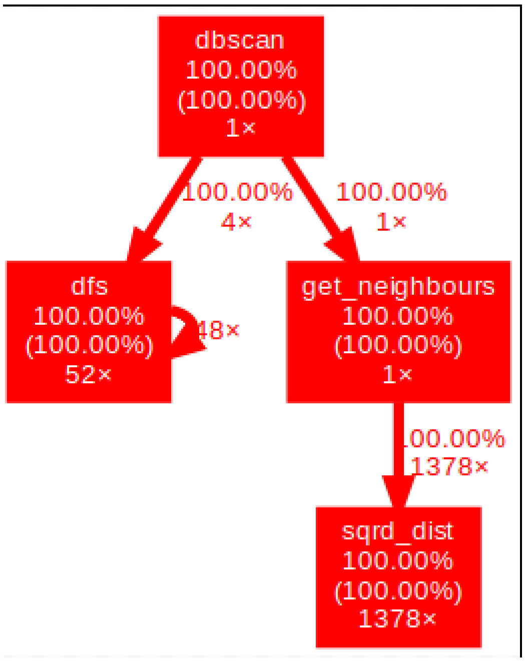


```
        for (int i = 0; i < num_pts; i++)
        {
            if (vis[i] != 1 && siz[i] >= minpts)
            {
                cnt++;
                printf("cluster %d : ", cnt);
                dfs(i,siz,vis);
                printf("\n");
            }

        }
        printf("NOISE :");
        offset = 0;
        for (int dest = 1; dest <= numworkers;
dest++)
        {
            vals = dest <= leftOver ? nPerProcess
+ 1 : nPerProcess;
            MPI_Send(&offset, 1, MPI_INT, dest,
1, MPI_COMM_WORLD);
            MPI_Send(&vals, 1, MPI_INT, dest, 1,
MPI_COMM_WORLD);
            MPI_Send(&vis[offset], vals, MPI_INT,
dest, 1, MPI_COMM_WORLD);
            offset = offset + vals;
        }
        /* wait for results from all worker tasks
*/
        for (int i = 1; i <= numworkers; i++)
```



```
       {
           source = i;
           MPI_Recv(&offset, 1, MPI_INT, source,
2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&vals, 1, MPI_INT, source,
2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
       }

       printf("\n");
   endTime = MPI_Wtime();
   printf("%d %lf\n", numProc, endTime -
startTime);
   }
   /*----------------------------
worker----------------------------*/
   if (rank > 0) {
       source = 0;
       MPI_Recv(&offset, 1, MPI_INT, source, 1,
MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
       MPI_Recv(&vals, 1, MPI_INT, source, 1,
MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
       int siz[vals], vis[vals];
       for (int i = 0; i < vals; i++)
       {
           siz[i] = 0;
           vis[i] = 0;
       }

       MPI_Send(&offset, 1, MPI_INT, 0, 2,
MPI_COMM_WORLD);
       MPI_Send(&vals, 1, MPI_INT, 0, 2,
MPI_COMM_WORLD);
       MPI_Send(&siz, vals, MPI_INT, 0, 2,
MPI_COMM_WORLD);
       MPI_Send(&vis, vals, MPI_INT, 0, 2,
MPI_COMM_WORLD);


       MPI_Recv(&offset, 1, MPI_INT, source, 1,
MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
       MPI_Recv(&vals, 1, MPI_INT, source, 1,
MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
       int vis_2[vals];
       MPI_Recv(&vis_2, vals, MPI_INT, source,
1, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
       for (int i = 0; i < vals; i++)
       {
           if(vis_2[i] == 0)
               printf("%d ", i + offset + 1);
       }
       MPI_Send(&offset, 1, MPI_INT, 0, 2,
MPI_COMM_WORLD);
       MPI_Send(&vals, 1, MPI_INT, 0, 2,
MPI_COMM_WORLD);
   }
   MPI_Finalize();
}
```

CRITICAL PART AND METHODOLOGY

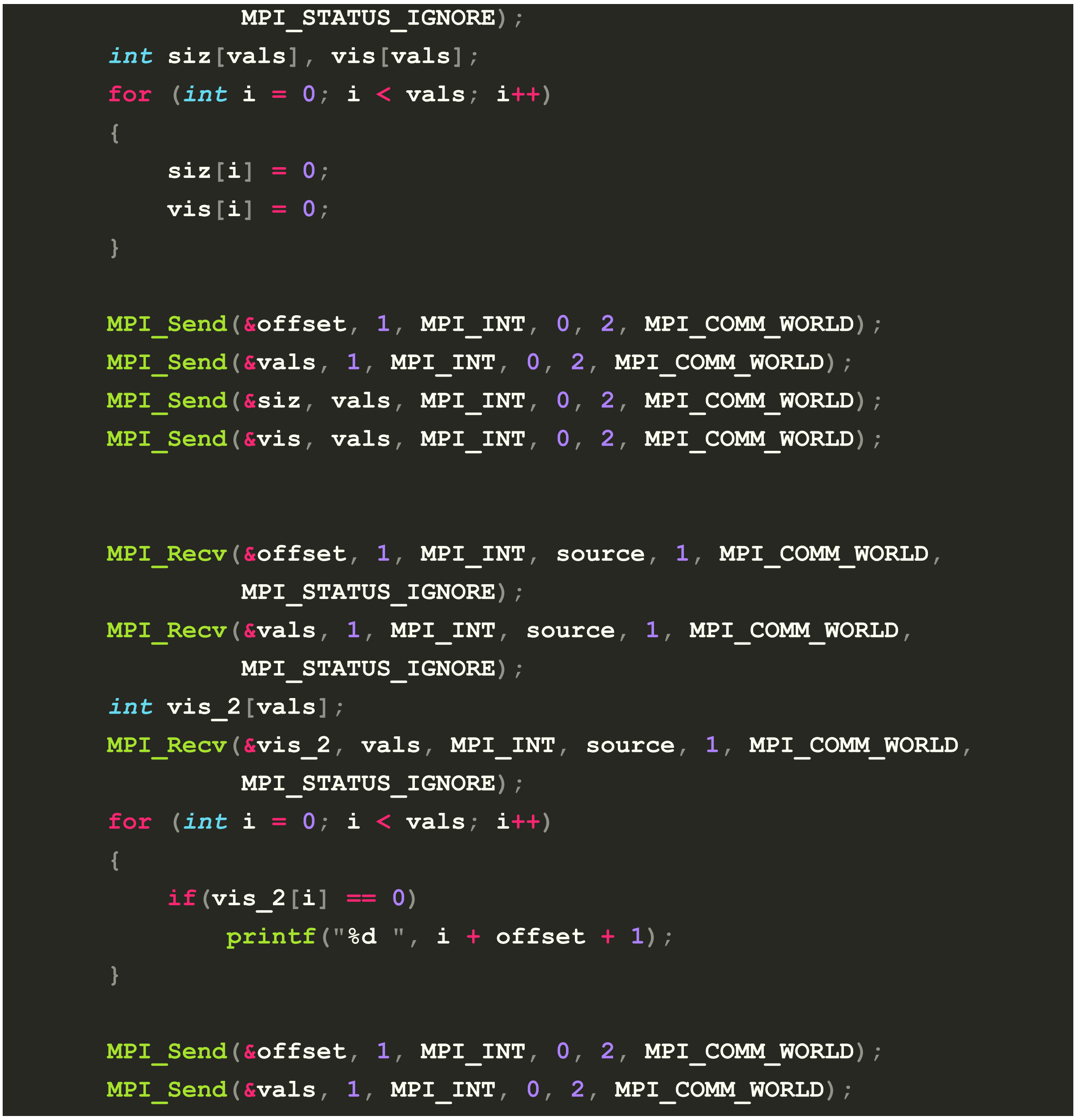


sqrt_dist is the most invoked function. It is invoked inside the get-neighbours. Initial attempt was made to parallelize the sqrt_dist function itself.

Significant improvement was observed by parallelizing the noise function. Furthermore, some more parallelization was tried on the noise function and initialization was done.

LOAD BALANCING AND SYNCHRONIZATION:

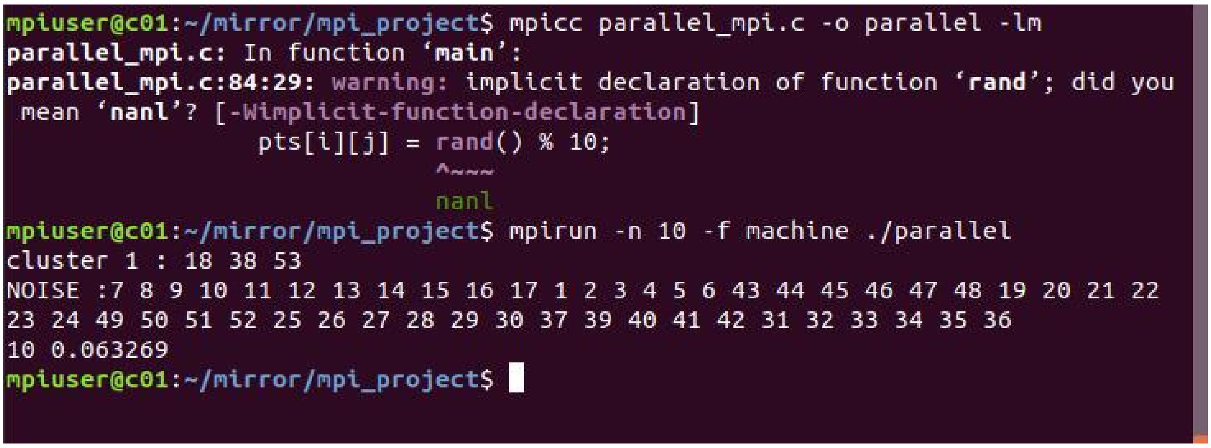
```
       nPerProcess = (num_pts) / numworkers;
       leftOver = (num_pts) % numworkers;
       offset = 0;
       for (int dest = 1; dest <= numworkers; dest++)
       {
           vals = dest <= leftOver ? nPerProcess + 1 : nPerProcess;
           MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
           MPI_Send(&vals, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
           offset = offset + vals;
       }
       /* wait for results from all worker tasks */
       for (int i = 1; i <= numworkers; i++)
       {
           source = i;
           MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&vals, 1, MPI_INT, source, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&siz[offset], vals, MPI_INT, source, 2,
MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&vis[offset], vals, MPI_INT, source, 2,
MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
       }
       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]++;
               }
           }
       }
       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,siz,vis);
               printf("\n");
           }

       }
       printf("NOISE :");
       offset = 0;
       for (int dest = 1; dest <= numworkers; dest++)
       {
           vals = dest <= leftOver ? nPerProcess + 1 : nPerProcess;
           MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
           MPI_Send(&vals, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
           MPI_Send(&vis[offset], vals, MPI_INT, dest, 1,
MPI_COMM_WORLD);
           offset = offset + vals;
       }
       /* wait for results from all worker tasks */
       for (int i = 1; i <= numworkers; i++)
       {
           source = i;
           MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
           MPI_Recv(&vals, 1, MPI_INT, source, 2, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
       }

       printf("\n");
   endTime = MPI_Wtime();
   printf("%d %lf\n", numProc, endTime - startTime);
   }
   /*---------------------------- worker----------------------------*/
   if (rank > 0) {
       source = 0;
       MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
       MPI_Recv(&vals, 1, MPI_INT, source, 1, MPI_COMM_WORLD,
```



```
             MPI_STATUS_IGNORE);
    int siz[vals], vis[vals];
    for (int i = 0; i < vals; i++)
    {
        siz[i] = 0;
        vis[i] = 0;
    }

    MPI_Send(&offset, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&vals, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&siz, vals, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&vis, vals, MPI_INT, 0, 2, MPI_COMM_WORLD);


    MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    MPI_Recv(&vals, 1, MPI_INT, source, 1, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    int vis_2[vals];
    MPI_Recv(&vis_2, vals, MPI_INT, source, 1, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    for (int i = 0; i < vals; i++)
    {
        if(vis_2[i] == 0)
            printf("%d ", i + offset + 1);
    }

    MPI_Send(&offset, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&vals, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
```

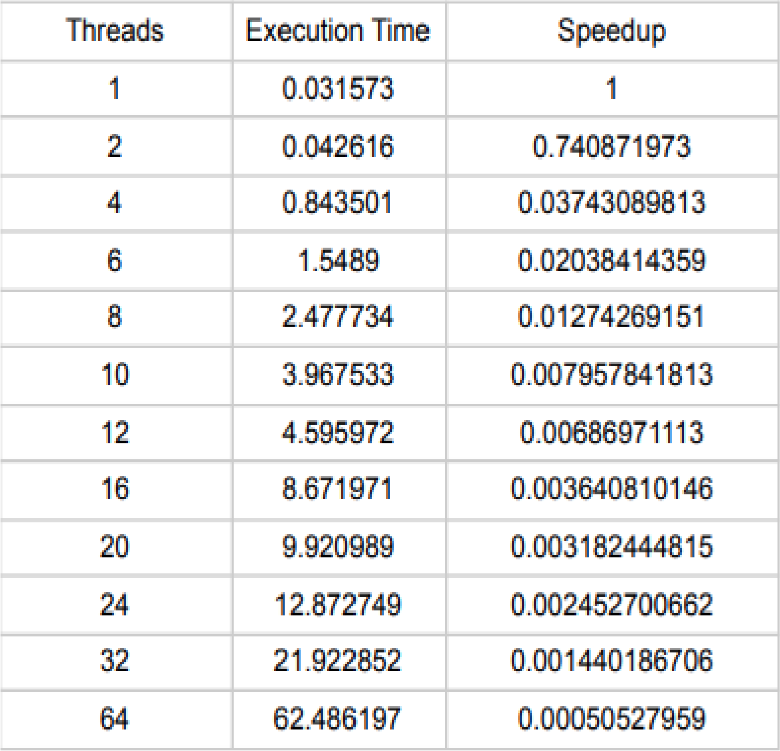
Machine_file :

Output:

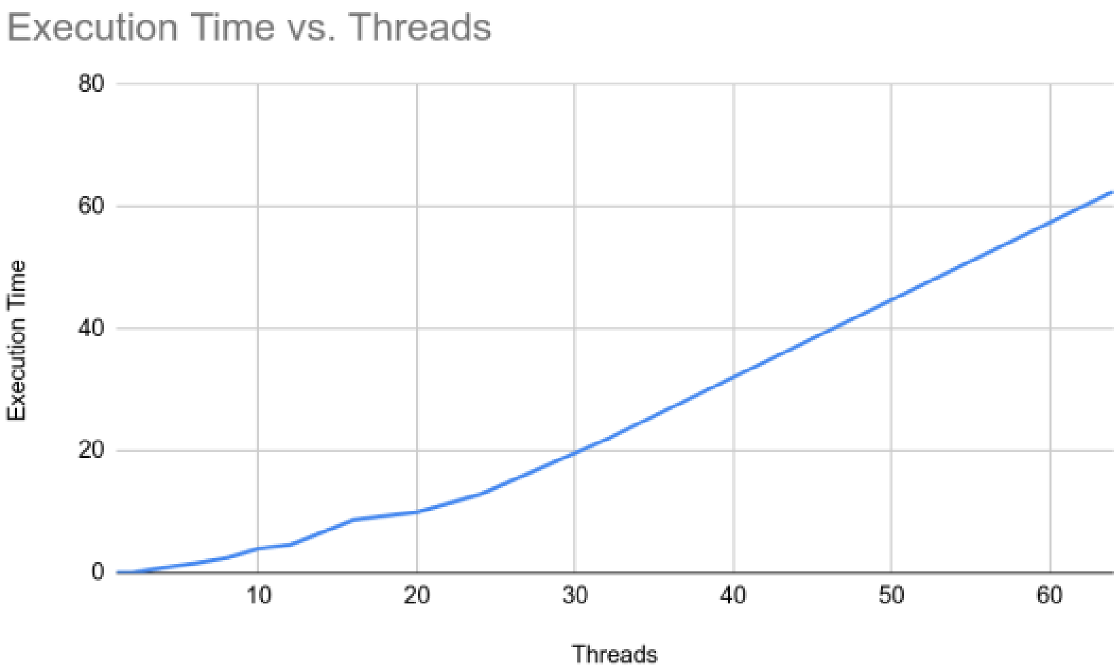


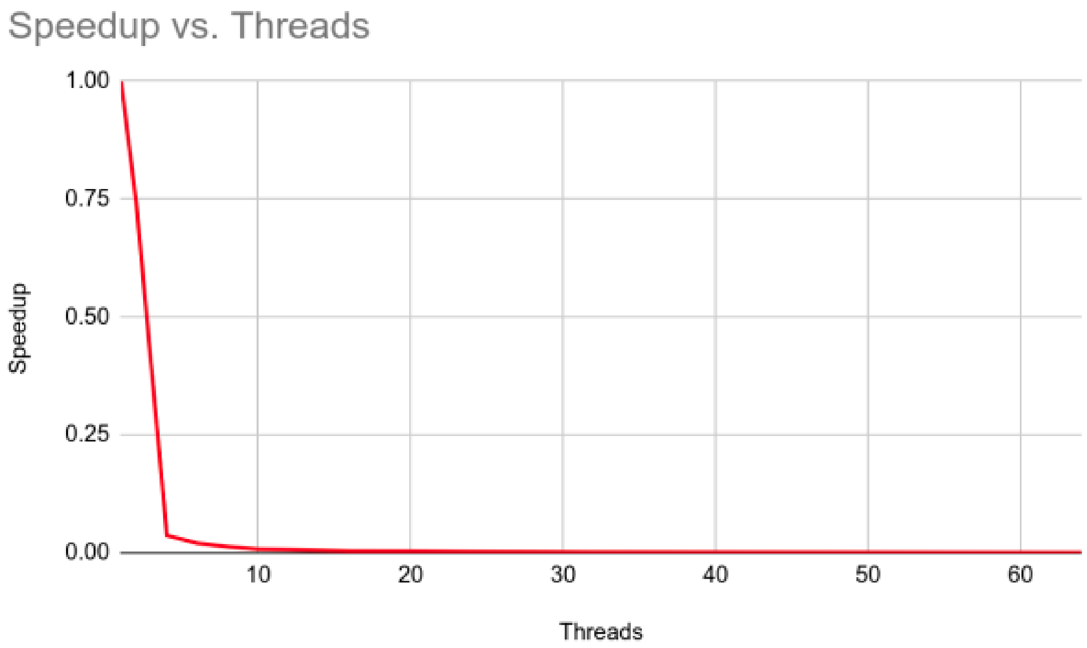
```
mpiuser@c01:~/mirror/mpi_project$ mpicc parallel_mpi.c -o parallel -lm
parallel_mpi.c: In function ‘main’:
parallel_mpi.c:84:29: warning: implicit declaration of function ‘rand’; did you
 mean ‘nanl’? [-Wimplicit-function-declaration]
                pts[i][j] = rand() % 10;
                            ^~~~
                            nanl
mpiuser@c01:~/mirror/mpi_project$ mpirun -n 10 -f machine ./parallel
cluster 1 : 18 38 53
NOISE :7 8 9 10 11 12 13 14 15 16 17 1 2 3 4 5 6 43 44 45 46 47 48 19 20 21 22
23 24 49 50 51 52 25 26 27 28 29 30 37 39 40 41 42 31 32 33 34 35 36
10 0.063269
mpiuser@c01:~/mirror/mpi_project$ 
```

Observations:



Threads	Execution Time	Speedup
1	0.031573	1
2	0.042616	0.740871973
4	0.843501	0.03743089813
6	1.5489	0.02038414359
8	2.477734	0.01274269151
10	3.967533	0.007957841813
12	4.595972	0.00686971113
16	8.671971	0.003640810146
20	9.920989	0.003182444815
24	12.872749	0.002452700662
32	21.922852	0.001440186706
64	62.486197	0.00050527959


Execution Time vs. Threads
Execution Time
80
60
40
20
0
10
20
30
40
50
60
Threads


Speedup vs. Threads
Speedup
1.00
0.75
0.50
0.25
0.00
10
20
30
40
50
60
Threads

Inference:

- Negligible speedup on higher thread count can be attributed to communication cost (MPI_Send/ MPI_Recv) and context switching between processors.