HIGH PERFORMANCE COMPUTING OPENMP

Implementation of Durand Kerner Method to solve Polynomial Equations

Durand Kerner

$$r_{n+1} = r_n - \frac{p(r_n)}{(r_n - s_n)(r_n - t_n)}$$

$$s_{n+1} = s_n - \frac{p(s_n)}{(s_n - r_n)(s_n - t_n)}$$

$$t_{n+1} = t_n - \frac{p(t_n)}{(t_n - r_n)(t_n - s_n)}$$

-Report By:
Sharan SK
CED18I049

Serial Code:

```c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include<omp.h>

#define M_PI 3.14159265358979323846
#define coff_size 500

double R=0;
double complex z[coff_size];
double complex deltaZ[coff_size];
double deltaZMax;
double epsilon = 1e-6;
double complex QsubJ,fz;
int max_iter = 1000;


//----------------------Function Prototypes-------------------------------
void durand_kerner(); //Prototypes
void calc_theta();
double max_cof();
void printz();
void update_z();
void update_fz();


int main() {

   double complex cList[coff_size];  //List of coefficients
   double complex z;
   double x,y; //x for real and y for imaginary parts of the coefficient
   int n=0; //n is number degree of polynomial


//------Read Coefficients-----------------------------------------------
```

```
   printf("Enter coefficients and enter any char other than number when
done:\n");
   while(scanf("%lf %lf",&x,&y) == 2)  { //Read coefficients from stdin
       cList[n] = (x + y*I);
       n++;
   }
   x = 1;  //Cn = 1, because the equation has to be normalized
   y = 0;
   z = (x + y*I);
   cList[n] = z; //Store in cList[]


   durand_kerner(cList,n);



}
//-----------------------------------Function
Definition-------------------------------


void durand_kerner(double complex cList[],int n) {
   float st;
   st=omp_get_wtime();
   R = 1 + max_cof(cList,n);  //End Equation 5

   calc_theta(n);
   int k;
   for(k=1;k <= max_iter;k++) {

       //printz(cList,n,k);

       deltaZMax = 0;

       update_fz(cList,n);
       update_z(n);

       if(deltaZMax <= epsilon) {
           break;
       }

   }
   st=omp_get_wtime()-st;
   printf("%d\n",k);
   printz(cList,n);
   printf("Time Taken=%f\n",st);

}

void calc_theta(int n) {
   for(int j=0;j < n;j++) {
       z[j] = ( cos( j*((2*M_PI)/n) ) + (I*sin( j*((2*M_PI)/n) )) )*R;
   }

}

double max_cof(double complex cList[],int n)
{
   double r;
   for(int j=0;j < n;j++) {
       if(cabs(cList[j]) > R) {
           r = cabs(cList[j]);
       }
   }

   return r;
}

void printz(double complex cList[],int n)
{
       printf("Final Output:(Note: if the roots repeat then there exist
less than n-1 roots for the equation)\n");
       for(int i=0;i < n;i++) {
                   printf("z[%d] = %0.10f +
%0.10f*I\n",i,creal(z[i]),cimag(z[i]));
```

```
                fflush(stdout);
            }
}

void update_z(int n)
{
   for(int j=0;j < n;j++) {
           z[j] = z[j] + deltaZ[j];
       }
}

void update_fz(double complex cList[],int n)
{
   for(int j=0;j < n;j++) {

           QsubJ = 1;
           for(int i=0;i < n;i++) {
               if(i != j) {
                   QsubJ = (z[j]-z[i])*QsubJ;
               }
           }
           fz = 1;
           for(int k = n-1;k >= 0;k--) {
               fz = fz*z[j] + cList[k];

           }

           deltaZ[j] = (-fz/QsubJ);

           if(cabs(deltaZ[j]) > deltaZMax) {
               deltaZMax = cabs(deltaZ[j]);
           }
       }
}
```

Pseudo Code of the Algorithm:

```
Compute initial values $\{z_0, \ldots, z_{n-1}\}$ using Equation 6.
for $k = 1 \ldots k_{\max}$
  Let $\Delta z_{\max} = 0$.
  for $j = 0 \ldots n-1$
    Compute the product $Q_j = \prod_{\substack{i=0 \\ i \neq j}}^{n-1} (z_j - z_i)$ (Equation 3).
    Set $\Delta z_j = -f(z_j)/Q_j$.
    if $|\Delta z_j| > \Delta z_{\max}$
      Set $\Delta z_{\max} = |\Delta z_j|$.
  for $j = 0 \ldots n-1$
    Update $z_j = z_j + \Delta z_j$.
  if $\Delta z_{\max} \leq \epsilon$ quit.
```

Serial Code Explanation/Observations:

- GIven the Max No of iterations,the program will run for either that many no of iterations or until the maximum absolute value of deltaZ becomes less e^{-5}.
- From the profiling report ,it was observed that the update_z function has the lines that run most of the times.Now these lines are inside a for loop which runs upto ‘n’ which is the degree of the polynomial equation/input provided.
- Every root is stored inside complex array called ‘z’ which is updated by deltaZ value which is calculated by -fz/QsubJ,where QsubJ is the difference between every root other than the jth root where j is the iterating variable/root and fz represents the result that is obtained by substituting the previous iteration’s calculated jth root value.

- The initial values of the root are calculated using the formula below:

$$z_j^{(0)} = (\cos\theta_j + i\sin\theta_j)\cdot R, \quad j = 0, \ldots, n-1,$$

$$\theta_j = j\frac{2\pi}{n}.$$

Where ‘j’ represents the jth root of the equation and (0) represents that this value is initial value ,i.e of 0th iteration.

How is program can be parallelized?:

- From the Above pseudo code ,we can see that the every iteration’s root value calculation depends on the previous root values meaning $Z_j^{(k)}$,which is jth root of the equation calculated in Kth iteration depends on $Z_j^{(k-1)}$ i.e on the previous iteration values .Hence the outer loop cannot be parallelized.
- It is also observed that the inner for loop i.e update_fz function depends on the previous Z values but calculates the root independently given this value ,which means if the threads wait until deltaZ is calculated and Z values of Kth iteration are updated then for every Kth iteration ,the roots can be parallely calculated at the same time .

Openmp Code Changes:

```
void durand_kerner(double complex cList[],int n,int threads) {
   float st;

   R = 1 + max_cof(cList,n);   //End Equation 5
   int a,i,j,k,l,m;
```

```
for(a=0;a < n;a++)
    z[a] = ( cos( a*((2*M_PI)/n) ) + (I*sin( a*((2*M_PI)/n) )) )*R;

st=omp_get_wtime();
for(k=1;k <= max_iter;k++) {

    deltaZMax = 0;

    //st=omp_get_wtime();
#pragma omp parallel private (i,j,l) shared (z,deltaZ,n,cList)
        {


#pragma omp for
for( j=0;j < n;j++) {

        double complex QsubJ;
        QsubJ = 1;
        //fz=0;
        #pragma omp parallel for reduction(*:QsubJ)
        for(i=0;i < n;i++) {
            if(i != j) {
                QsubJ*= (z[j]-z[i]);
            }
        }

        //fz=0;
        fz_temp = 1;

        te=z[j];

        for(l = n-1;l >= 0;l--) {
            t=fz_temp*te;
            fz_temp =t+ cList[l];
            //fz = fz*z[j] + cList[l];
        }
```

```
            fz=fz_temp;

            deltaZ[j] = (-fz/QsubJ);

        z[j]+=deltaZ[j];
        printf("Iter=%d %d \n",j,k);
        printf("z[%d] = %0.10f + %0.10f*I\n",j,creal(z[j]),cimag(z[j]));
            //printf("z[%d] = %0.10f + %0.10f*I ,fz= %0.10f +
%0.10f*I",j,creal(z[j]),cimag(z[j]),creal(fz),cimag(fz));
            //printf("z[%d] = %0.10f + %0.10f*I ,QsubJ= %0.10f +
%0.10f*I",j,creal(z[j]),cimag(z[j]),creal(QsubJ),cimag(QsubJ));
    }
    #pragma omp barrier
    }
    printf("\n");


        for(m=0;m<n;m++)
        {
            if(cabs(deltaZ[m]) > deltaZMax) {
                deltaZMax = cabs(deltaZ[m]);
        }
        }
        //printf("Zmax=%f %d; ",deltaZMax,k);

        if(deltaZMax <= epsilon) {
            break;
        }


    }

    st=omp_get_wtime()-st;
    printf("Max Iteration=%d\n",k);
    printfile(cList,n,k,st,threads);
    printf("Time Taken=%f\n",st);
}
```

- ➢ Above is the Durand-Kerner function that calculated the roots.It can be observed the parallelization region is mentioned using “#pragma omp parallel private (i,j,l) shared (z,deltaZ,n,cList)” loop which is inside the first outer for loop that runs upto max no of iterations.
- ➢ QsubJ calculates the product of difference between every other root and j th root which can be parallelized using reduction concept in openmp ,and the reduction operator here is ‘ * ‘ (Multiplication).
- ➢ fz calculation cannot be parallelized as multiple operators are involved and cant be reduced .
- ➢ In order to ensure that the threads are completed before going on to the next iteration to prevent race condition,’#pragma omp barrier’ is used ,which waits for all the threads to complete.
- ➢ **Concepts Used:**
    - ○ Reduction using OpenMp
    - ○ Thread Wait using OpenMP
    - ○ Parallelized Code to support OpenMP

Observations on Execution time of the program:

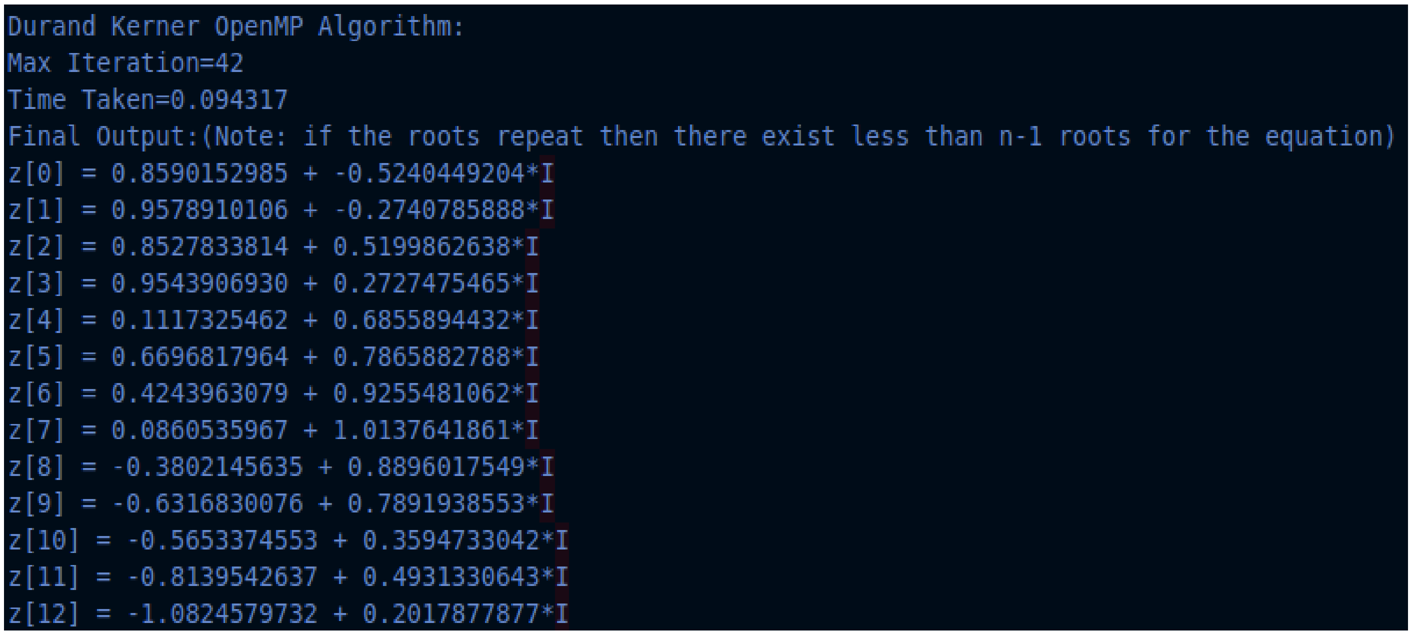
Note :

- ➢ Each line in the given input takes the coefficients x and y such that each coefficient is of the form (x+ y*I)
- ➢ For the below input there 23 lines,so the highest degree of the equation is 24 ,since it is assumed that the equation is scaled such that the leading coefficient is 1 ,making the degree equal to 1 + no of lines in the input
- ➢ The no of the roots will be less than or equal to (degree -1) ,for the below input ,there will be at most 23 roots.If some roots are repeated then it has below 23 roots.
- ➢ To Execute the program ,execute the following code
    1) gcc -fopenmp <filename>-lm -o <object filename>
    2) ./<object filename>
    3) Once coefficients are entered in the below order ,enter any letter to stop giving inputs like ‘e’ in the below case.

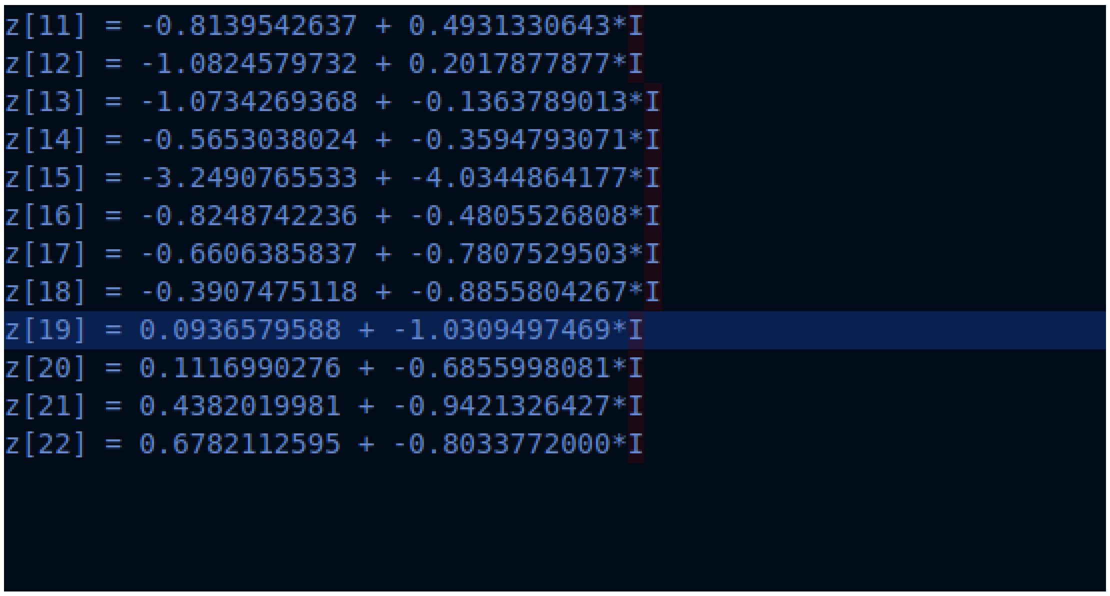
Given input:

```
1 1
2 2
3 3
4 4
5 5
1 1
2 2
3 3
4 4
2 2
3 3
4 4
4 4
2 2
3 3
4 4
2 2
3 3
4 4
4 4
2 2
3 3
4 4
e
```

Output:(for the above equation):



```
Durand Kerner OpenMP Algorithm:
Max Iteration=42
Time Taken=0.094317
Final Output:(Note: if the roots repeat then there exist less than n-1 roots for the equation)
z[0] = 0.8590152985 + -0.5240449204*I
z[1] = 0.9578910106 + -0.2740785888*I
z[2] = 0.8527833814 + 0.5199862638*I
z[3] = 0.9543906930 + 0.2727475465*I
z[4] = 0.1117325462 + 0.6855894432*I
z[5] = 0.6696817964 + 0.7865882788*I
z[6] = 0.4243963079 + 0.9255481062*I
z[7] = 0.0860535967 + 1.0137641861*I
z[8] = -0.3802145635 + 0.8896017549*I
z[9] = -0.6316830076 + 0.7891938553*I
z[10] = -0.5653374553 + 0.3594733042*I
z[11] = -0.8139542637 + 0.4931330643*I
z[12] = -1.0824579732 + 0.2017877877*I
```



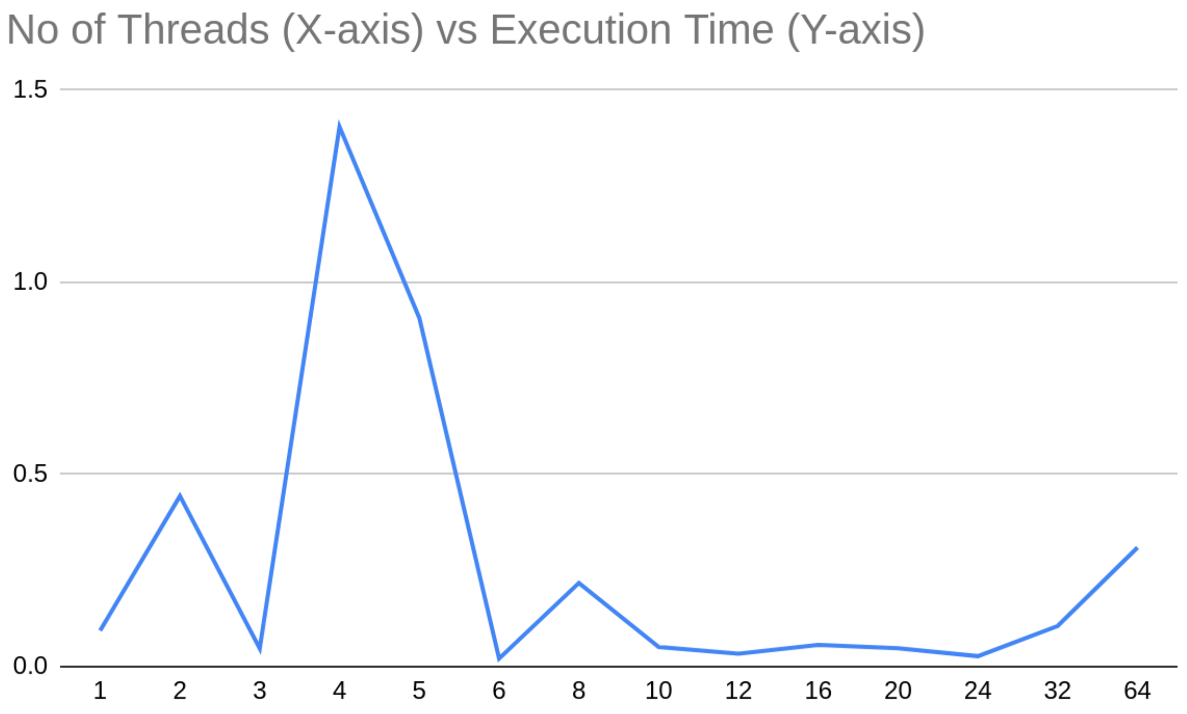
```
z[11] = -0.8139542637 + 0.4931330643*I
z[12] = -1.0824579732 + 0.2017877877*I
z[13] = -1.0734269368 + -0.1363789013*I
z[14] = -0.5653038024 + -0.3594793071*I
z[15] = -3.2490765533 + -4.0344864177*I
z[16] = -0.8248742236 + -0.4805526808*I
z[17] = -0.6606385837 + -0.7807529503*I
z[18] = -0.3907475118 + -0.8855804267*I
z[19] = 0.0936579588 + -1.0309497469*I
z[20] = 0.1116990276 + -0.6855998081*I
z[21] = 0.4382019981 + -0.9421326427*I
z[22] = 0.6782112595 + -0.8033772000*I
```

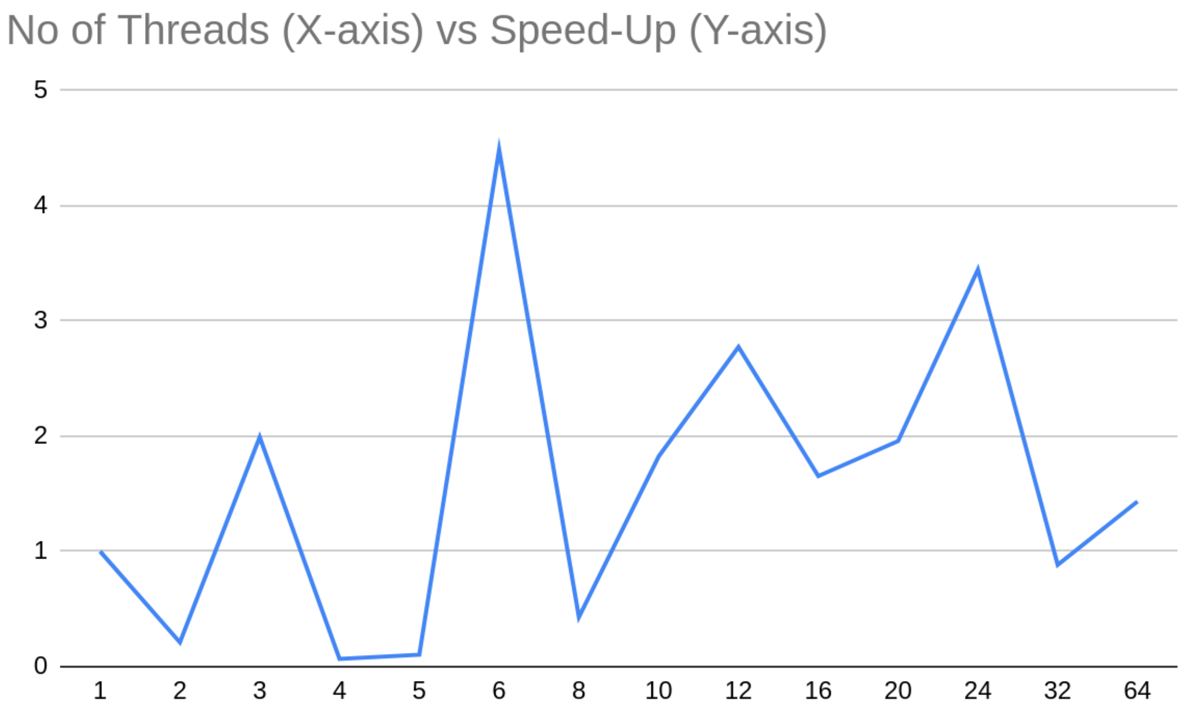
Observation of execution time:

Note:Execution time is calculated with respect to the function update_fz as it is the function that is parallelized in the code.

For the above given input:

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.093551	1	
2	0.443847	0.21	-7.49
3	0.046975	1.99	0.75
4	1.40479	0.07	-18.69
5	0.907321	0.10	-10.87
6	0.02085	4.49	0.93
8	0.217709	0.43	-1.52
10	0.051246	1.83	0.50
12	0.033722	2.77	0.70
16	0.056567	1.65	0.42
20	0.047775	1.96	0.52
24	0.027148	3.45	0.74
32	0.105784	0.88	-0.13
64	0.309696	1.43	0.31


No of Threads (X-axis) vs Execution Time (Y-axis)
1.5
1.0
0.5
0.0
1
2
3
4
5
6
8
10
12
16
20
24
32
64


No of Threads (X-axis) vs Speed-Up (Y-axis)
5
4
3
2
1
0
1
2
3
4
5
6
8
10
12
16
20
24
32
64

Inference:

- From the above inference ,the max speedup of the program is 4.49 for no of threads =6 and parallelization fraction is 0.93 for the same.
- The Program involves complex numbers and increasing degree of the equation causes the program to be unstable thereby making the program to project roots are “nan” (Not a number)
- When multiplying complex number ,c compiler’s inbuilt function __muldc3 is invoked when runs most of the time in the program ,as every time roots are calculated they are multiplications involving two complex numbers which can’t be prevented.

Complete OpenMP Parallel Code:

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include<omp.h>
#include<unistd.h>

#define M_PI 3.14159265358979323846
#define coff_size 500

double R=0;
double complex z[coff_size];
double complex deltaZ[coff_size];
double deltaZMax;
double epsilon = 1e-6;
double complex fz,t,te,fz_temp;
int max_iter = 500;


//---------------------Function Prototypes-----------------------------
void durand_kerner(); //Prototypes
void calc_theta();
double max_cof();
```

```
void printz();
void update_z();
void update_fz();
void printfile(double complex cList[],int n,int k,float st,int threads);


int main() {

   double complex cList[coff_size];  //List of coefficients
   double complex z;
   double x,y; //x for real and y for imaginary parts of the coefficient
   int n=0; //n is number degree of polynomial


//------Read Coefficients--------------------------------------------------
   printf("Enter coefficients and enter any char other than number when
done:\n");
   while(scanf("%lf %lf",&x,&y) == 2)  { //Read coefficients from stdin
       cList[n] = (x + y*I);
       n++;
   }
   x = 1;  //Cn = 1, because the equation has to be normalized
   y = 0;
   z = (x + y*I);
   cList[n] = z; //Store in cList[]
   int thread[]={1, 2,3, 4,5, 6, 8, 10, 12, 16, 20, 24, 32, 64};
   //for(int i=0;i<4;i++){
   //omp_set_num_threads(thread[i]);
   omp_set_num_threads(8);
   durand_kerner(cList,n,8);
   //durand_kerner(cList,n,thread[i]);
   //}


}
//-----------------------------------Function
Definition--------------------------------
```

```
void durand_kerner(double complex cList[],int n,int threads) {
   float st;

   R = 1 + max_cof(cList,n);  //End Equation 5
   int a,i,j,k,l,m;



   for(a=0;a < n;a++)
       z[a] = ( cos( a*((2*M_PI)/n) ) + (I*sin( a*((2*M_PI)/n) )) )*R;

   st=omp_get_wtime();
   for(k=1;k <= max_iter;k++) {

       deltaZMax = 0;

       //st=omp_get_wtime();
   #pragma omp parallel private (i,j,l) shared (z,deltaZ,n,cList)
           {


   #pragma omp for
   for( j=0;j < n;j++) {

           double complex QsubJ;
           QsubJ = 1;
           //fz=0;
           #pragma omp parallel for reduction(*:QsubJ)
           for(i=0;i < n;i++) {
               if(i != j) {
                   QsubJ*= (z[j]-z[i]);
               }
           }

           //fz=0;
```

```
            fz_temp = 1;

            te=z[j];

            for(l = n-1;l >= 0;l--) {
                t=fz_temp*te;
                fz_temp =t+ cList[l];
                //fz = fz*z[j] + cList[l];
            }

            fz=fz_temp;

            deltaZ[j] = (-fz/QsubJ);

        z[j]+=deltaZ[j];
        printf("Iter=%d %d \n",j,k);
        printf("z[%d] = %0.10f + %0.10f*I\n",j,creal(z[j]),cimag(z[j]));
            //printf("z[%d] = %0.10f + %0.10f*I ,fz= %0.10f +
%0.10f*I",j,creal(z[j]),cimag(z[j]),creal(fz),cimag(fz));
            //printf("z[%d] = %0.10f + %0.10f*I ,QsubJ= %0.10f +
%0.10f*I",j,creal(z[j]),cimag(z[j]),creal(QsubJ),cimag(QsubJ));
    }
    #pragma omp barrier
    }
    printf("\n");


        for(m=0;m<n;m++)
        {
            if(cabs(deltaZ[m]) > deltaZMax) {
                deltaZMax = cabs(deltaZ[m]);
        }
        }
        //printf("Zmax=%f %d; ",deltaZMax,k);

        if(deltaZMax <= epsilon) {
            break;
        }

    }

    st=omp_get_wtime()-st;
    printf("Max Iteration=%d\n",k);
    printfile(cList,n,k,st,threads);
    printf("Time Taken=%f\n",st);

}



double max_cof(double complex cList[],int n)
{
    double r;
    for(int j=0;j < n;j++) {
        if(cabs(cList[j]) > R) {
            r = cabs(cList[j]);
        }
    }

    return r;
}

void printz(double complex cList[],int n)
{

        printf("Final Output:(Note: if the roots repeat then there exist
less than n-1 roots for the equation)\n");
        for(int i=0;i < n;i++) {
                        printf("z[%d] = %0.10f +
%0.10f*I\n",i,creal(z[i]),cimag(z[i]));
                    fflush(stdout);
            }
}
void printfile(double complex cList[],int n,int k,float st,int threads)
{
        FILE *fp;
        fp = fopen("openmp_project_roots.txt", "w");
        fprintf(fp,"Durand Kerner OpenMP Algorithm:\n");
        fprintf(fp,"Max Iteration=%d\n",k);
        fprintf(fp,"Time Taken=%f\n",st);
        fprintf(fp,"Final Output:(Note: if the roots repeat then there
exist less than n-1 roots for the equation)\n");
        for(int i=0;i < n;i++) {
                        fprintf(fp,"z[%d] = %0.10f +
%0.10f*I\n",i,creal(z[i]),cimag(z[i]));
                    fflush(stdout);
            }
        fclose(fp);
        fp=fopen("openmp_time.txt", "a");
        fprintf(fp,"Thread=%d\tTime Taken=%f\n",threads,st);
        fclose(fp);
}
```