HIGH PERFORMANCE COMPUTING MPI CLUSTER

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:

```
1  Compute initial values $\{z_0, \ldots, z_{n-1}\}$ using Equation 6.
2  for $k = 1 \ldots k_{\max}$
3    Let $\Delta z_{\max} = 0$.
4    for $j = 0 \ldots n-1$
5      Compute the product $Q_j = \prod_{\substack{i=0 \\ i \neq j}}^{n-1} (z_j - z_i)$ (Equation 3).
6      Set $\Delta z_j = -f(z_j)/Q_j$.
7      if $|\Delta z_j| > \Delta z_{\max}$
8        Set $\Delta z_{\max} = |\Delta z_j|$.
9    for $j = 0 \ldots n-1$
10     Update $z_j = z_j + \Delta z_j$.
11   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 j^{th} root where j is the iterating variable/root and fz represents the result that is obtained by substituting the previous iteration’s calculated j^{th} 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 in MPI?:

➢ 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 .

MPI Code Changes:

```
void durand_kerner(int n,int argc, char *argv[]) {

    //----------------------------------MPI-------------------------------
    int no_tasks, taskid, no_workers, source, workers, mtype,
no_elements_iter, no_elements, no_elements_left, index, i, j, k,rc, ack;
    double start,end;

    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &no_tasks);
    if (no_tasks < 2)
    {
        printf("Available Processors =%d\n",no_tasks);
        printf("Program is Terminated since there are less than 2 
threads\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
        exit(1);
    }

    char pro_name[MPI_MAX_PROCESSOR_NAME];
    int length;
    MPI_Get_processor_name(pro_name,&length);

    printf("Processor name = %s, rank %d out of %d 
processors\n",pro_name,taskid,no_tasks);
    no_workers = no_tasks - 1;

//-----------------------------------MASTER-------------------------------

    if(taskid==0)
    {   double x,y;
        while(scanf("%lf %lf",&x,&y) == 2)  {
        cList[n] = (x + y*I);
        n++;
    }
        cList[n]=(1 + 0*I);
        start=MPI_Wtime();
        float st,total;

        R = 1 + max_cof(cList,n);

        calc_theta(n);
        int k;
        index=0;

        for(int i=1;i<=no_workers;i++)
        {
            MPI_Send(&max_iter, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
            MPI_Send(&n, 1, MPI_INT, i,2, MPI_COMM_WORLD);
            MPI_Send(&cList, n, MPI_DOUBLE_COMPLEX, i,3, MPI_COMM_WORLD);
        }


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

                deltaZMax = 0;

            no_elements = n / no_workers;
            no_elements_left = n % no_workers;

            index = 0;
            mtype = FROM_MASTER;

            for (workers = 1; workers <= no_workers; workers++)
            {

                no_elements_iter = (workers == no_workers) ? no_elements +
no_elements_left : no_elements;

                MPI_Send(&index, 1, MPI_INT, workers, 4, MPI_COMM_WORLD);
                MPI_Send(&no_elements_iter, 1, MPI_INT, workers, 5,
MPI_COMM_WORLD);
                MPI_Send(&z, n, MPI_DOUBLE_COMPLEX, workers, 6,
MPI_COMM_WORLD);


                index += no_elements_iter;
            }

            mtype = FROM_WORKER;
            for (i = 1; i <= no_workers; i++)
            {
                source = i;

                MPI_Recv(&index, 1, MPI_INT, source, 7, MPI_COMM_WORLD,
&status);
                MPI_Recv(&no_elements_iter, 1, MPI_INT, source, 8,
MPI_COMM_WORLD, &status);
                MPI_Recv(&deltaZ[index], no_elements_iter,
MPI_DOUBLE_COMPLEX, source,9, MPI_COMM_WORLD, &status);

            }

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

                if(cabs(deltaZ[j]) > deltaZMax) {
                    deltaZMax = cabs(deltaZ[j]);
                }
            }
                                ack=1;
                if(deltaZMax <= epsilon) {
                    ack=0;
                }
                printf(" Zmax=%f %d;\n",deltaZMax,k);

            mtype = FROM_MASTER;

            for (workers = 1; workers <= no_workers; workers++)
            {

                MPI_Send(&ack, 1, MPI_INT, workers, 10, MPI_COMM_WORLD);
            }
            if(ack==0)
            break;

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

    }
    //----------------------WORKERS-------------------------------
    if(taskid>0)
    {

        mtype = FROM_MASTER;

        MPI_Recv(&max_iter, 1, MPI_INT, MASTER, 1, MPI_COMM_WORLD,
&status);

        MPI_Recv(&n, 1, MPI_INT, MASTER, 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&cList, n, MPI_DOUBLE_COMPLEX, MASTER, 3, MPI_COMM_WORLD,
&status);

        for(int l=0;l<max_iter;l++)
        {

            MPI_Recv(&index, 1, MPI_INT, MASTER, 4, MPI_COMM_WORLD,
&status);

            MPI_Recv(&no_elements_iter, 1, MPI_INT, MASTER, 5,
MPI_COMM_WORLD, &status);

            MPI_Recv(&z, n, MPI_DOUBLE_COMPLEX, MASTER, 6, MPI_COMM_WORLD,
&status);
```

```
            for(int j=index;j < index+no_elements_iter;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);

            }

            mtype = FROM_WORKER;

            MPI_Send(&index, 1, MPI_INT, MASTER, 7, MPI_COMM_WORLD);

            MPI_Send(&no_elements_iter, 1, MPI_INT, MASTER, 8,
MPI_COMM_WORLD);

            MPI_Send(&deltaZ[index], no_elements_iter, MPI_DOUBLE_COMPLEX,
MASTER, 9, MPI_COMM_WORLD);

            mtype = FROM_MASTER;
            MPI_Recv(&ack, 1, MPI_INT, MASTER, 10, MPI_COMM_WORLD,
&status);

            if(ack==1)
            {
                continue;
            }
            else
            l=max_iter+1;
        }

    }
    MPI_Finalize();
}
```

How is the Parallelization achieved in MPI?:

➢ In MPI the workers are split based on the number provided by the user.Assuming the no of tasks given by the user is ‘n_tasks’ and the degree of the polynomial given by the user is ‘n’ then the ‘n_tasks-1’ workers calculate the roots and Master collects the results and updates the roots.

➢ Each Worker calculates (‘n_tasks-1’/’n’) root’s deltaZ Value on an average.Once the worker calculates the deltaZ it sends the value to master and master updates z value by adding deltaZ to the corresponding root.

➢ Program Analysis:
  - Master sends the following to the workers before starting the calculation:
    - The no of iterations each worker has to be do before entering into the iteration
    - The no of roots of the polynomial
    - The Complex Array containing the coefficients of the polynomial
  - Once the above are sent,the master then sends the initial root values of the polynomial.**NOTE: All the roots had to be sent to all the workers because the variable QsubJ requires the previous root values .It is not necessary to pass all the coefficients of the polynomial during each iteration because only the roots are updated at the end of iteration**
  - Each worker then calculates its corresponding no of deltaZ values and sends its contribution to the master.Master collects all the deltaZ values and updates it to the root.

- Once the roots are updated by the master,the master sends an acknowledgement to all workers telling them to move on to the next iteration without which there is a possibility of infinite wait or race condition.

Observations on Execution time of the program:

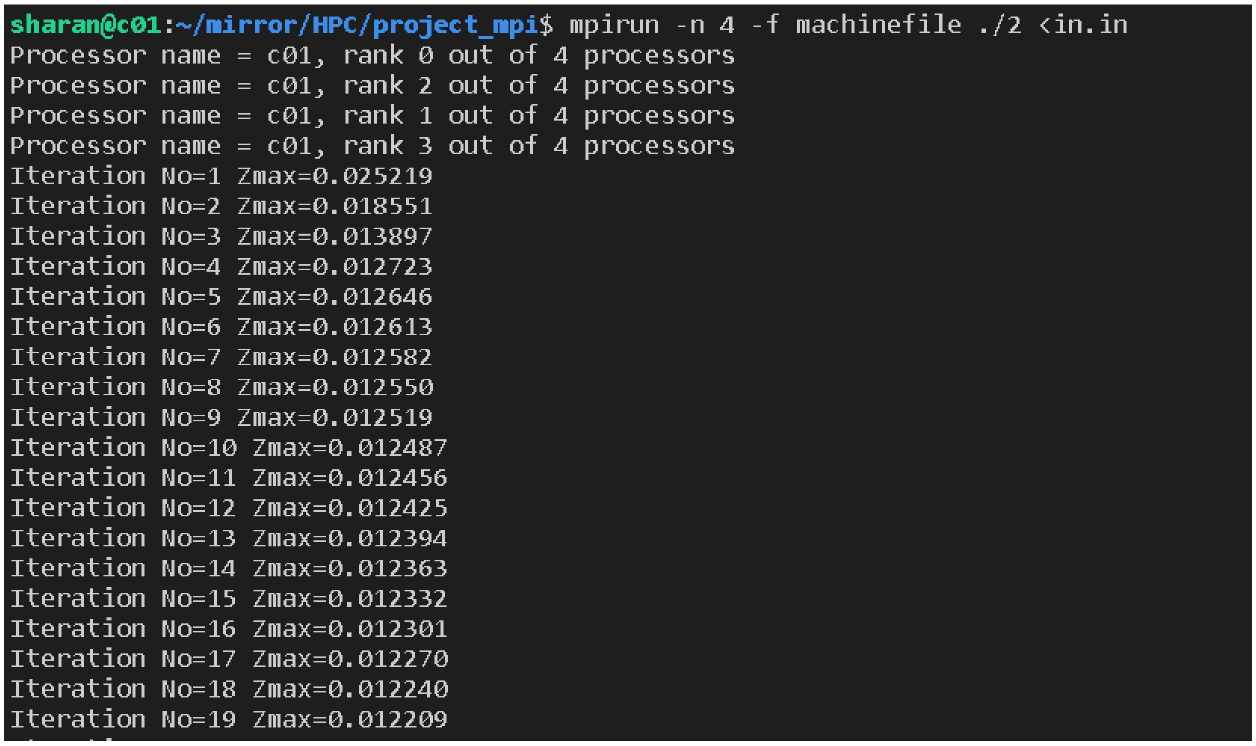
Note :

- ➢ MPI is very stable as compared to OPENMP and hence the no of roots for which the program can run is higher in this document as compared to OPENMP observation report also the output of root value during every iteration is removed owing to the fact that MPI being stable
- ➢ Since the input file is large ,it is attached as a link to the document.

Link:
https://drive.google.com/file/d/1de_Aq780bhfxRD0CpU1o_cZgxFYXt46b/view?usp=sharing

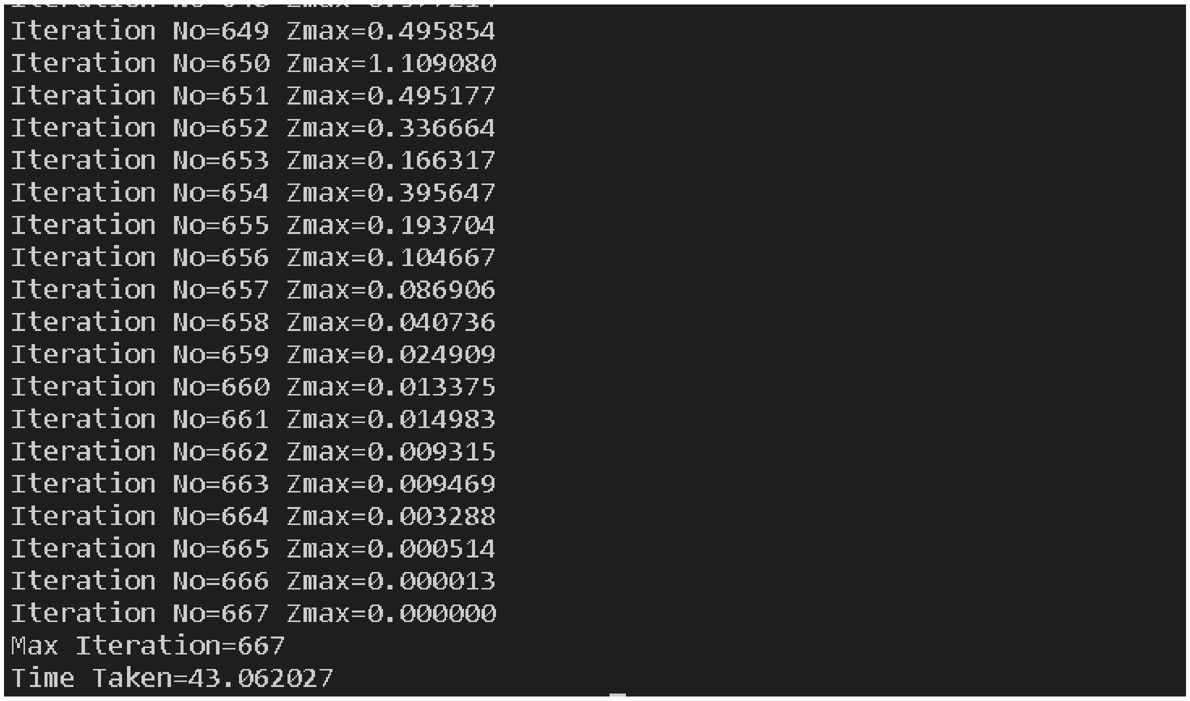
Output::

Sample Terminal Starting for the above input file included:



```
sharan@c01:~/mirror/HPC/project_mpi$ mpirun -n 4 -f machinefile ./2 <in.in
Processor name = c01, rank 0 out of 4 processors
Processor name = c01, rank 2 out of 4 processors
Processor name = c01, rank 1 out of 4 processors
Processor name = c01, rank 3 out of 4 processors
Iteration No=1 Zmax=0.025219
Iteration No=2 Zmax=0.018551
Iteration No=3 Zmax=0.013897
Iteration No=4 Zmax=0.012723
Iteration No=5 Zmax=0.012646
Iteration No=6 Zmax=0.012613
Iteration No=7 Zmax=0.012582
Iteration No=8 Zmax=0.012550
Iteration No=9 Zmax=0.012519
Iteration No=10 Zmax=0.012487
Iteration No=11 Zmax=0.012456
Iteration No=12 Zmax=0.012425
Iteration No=13 Zmax=0.012394
Iteration No=14 Zmax=0.012363
Iteration No=15 Zmax=0.012332
Iteration No=16 Zmax=0.012301
Iteration No=17 Zmax=0.012270
Iteration No=18 Zmax=0.012240
Iteration No=19 Zmax=0.012209
```

Sample Terminal Ending for the above input file included:



```
Iteration No=649 Zmax=0.495854
Iteration No=650 Zmax=1.109080
Iteration No=651 Zmax=0.495177
Iteration No=652 Zmax=0.336664
Iteration No=653 Zmax=0.166317
Iteration No=654 Zmax=0.395647
Iteration No=655 Zmax=0.193704
Iteration No=656 Zmax=0.104667
Iteration No=657 Zmax=0.086906
Iteration No=658 Zmax=0.040736
Iteration No=659 Zmax=0.024909
Iteration No=660 Zmax=0.013375
Iteration No=661 Zmax=0.014983
Iteration No=662 Zmax=0.009315
Iteration No=663 Zmax=0.009469
Iteration No=664 Zmax=0.003288
Iteration No=665 Zmax=0.000514
Iteration No=666 Zmax=0.000013
Iteration No=667 Zmax=0.000000
Max Iteration=667
Time Taken=43.062027
```

Output File of both the serial and the MPI can be found in the below link where each file contains the roots of the polynomial provided in (the above link file) input file:

Serial Output Link:
https://drive.google.com/file/d/1F6pYVqikDIhkbSd6Uz30IVCrBTEVtfH0/view?usp=sharing
MPI Output Link:
https://drive.google.com/file/d/160vtCouU78wRak8yBjGqIB7ZgZIGgonQ/view?usp=sharing

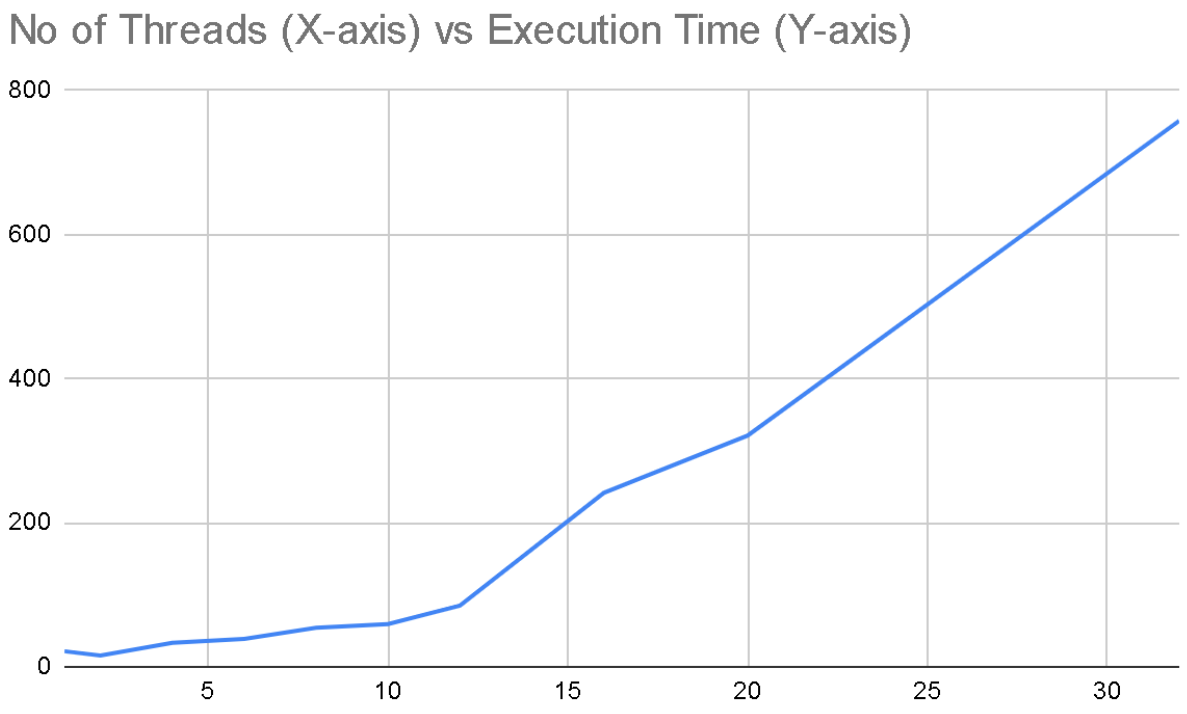
Observation:

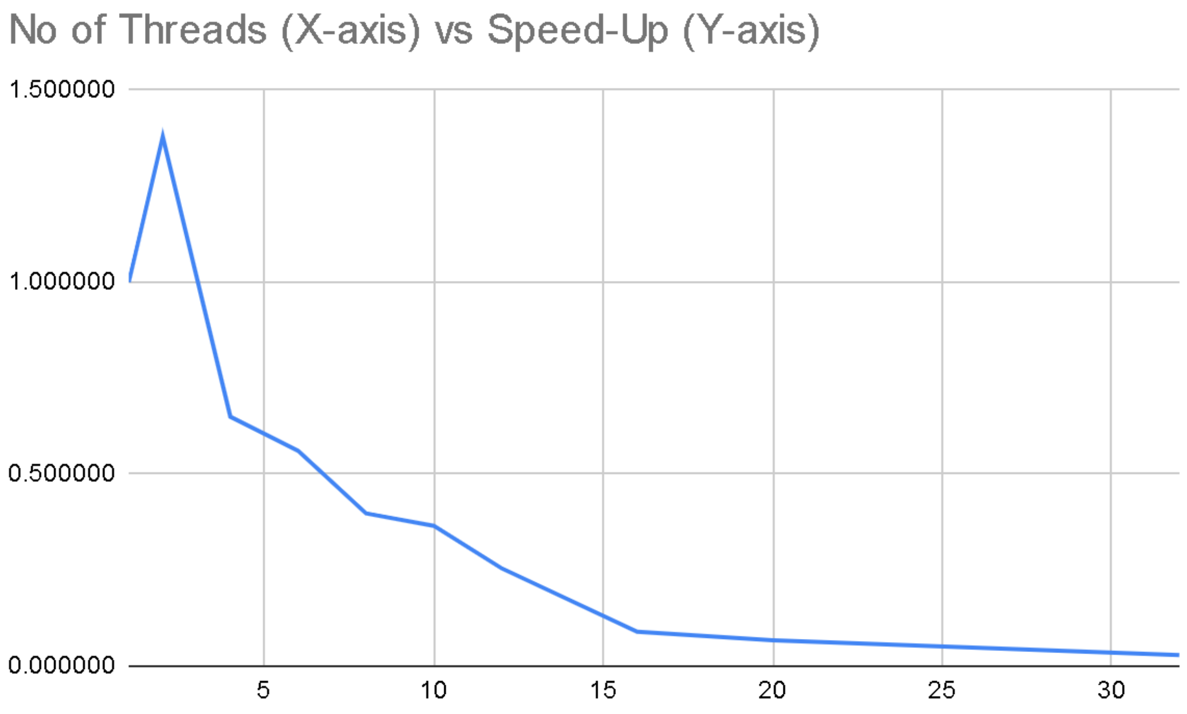
Note:Execution time for MPI will be increasing as the no of the tasks are subjected to increase owing to the fact that the program runs in local computer and not on a real cluster.Hence in this report it is observed that No of Iterations taken by the MPI for all the threads is 667 as opposed to 978 iterations taken by the serial code for the roots to converge which is proof to the fact that Parallel Processing is happening in the code.NOTE: No of threads here is assumed as no of tasks.

For the above given input:

Execution Time vs Number of Threads:

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	21.657925	1.000000	
2	15.709623	1.378641	0.549296
4	33.354614	0.649323	-0.720087
6	38.624592	0.560729	-0.940072
8	54.436192	0.397859	-1.729661
10	59.264652	0.365444	-1.929329
12	84.96621	0.254900	-3.188837
16	241.532501	0.089669	-10.828964
20	321.00766	0.067469	-14.549177
32	757.729187	0.028583	-35.082562


No of Threads (X-axis) vs Execution Time (Y-axis)
800
600
400
200
0
5
10
15
20
25
30


No of Threads (X-axis) vs Speed-Up (Y-axis)
1.500000
1.000000
0.500000
0.000000
5
10
15
20
25
30

Inference:

- ➢ From the above inference ,the max speedup of the program is 1.378 for no of tasks =2 and parallelization fraction is 0.549 for the same.
- ➢ Since the MPI runs on the local computer ,the execution time increases with more no of tasks because of the Communication overhead between the tasks.
- ➢ On comparing with the OpenMP ,the code is stable even for calculating polynomials of degree upto 400 after which the programs throws in segmentation fault as the data type double complex is used ,the memory support is reduced.
- ➢ It is also observed that the no of iteration taken by MPI code is ⅓ less than the iterations taken by the serial code for the final answer ,this shows the potential of Parallel Programming ,if only the MPI code is run on an actual MPI Cluster ,the difference can be truly appreciated.

Complete MPI Parallel Code:

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <mpi.h>
#include <stdlib.h>

#define M_PI 3.14159265358979323846
#define coff_size 501

#define MASTER 0

#define FROM_MASTER 1

#define FROM_WORKER 2

double R=0;
double complex z[coff_size];
double complex cList[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();
void calc_theta();
double max_cof();
void hello();
void printz();
void update_z();
void printfile(double complex cList[],int n,int k,float st);


int main(int argc, char *argv[]) {
    double complex z;

    int n=0;

    durand_kerner(n,argc, argv);

    return 0;
}
//-----------------------------------Function
Definition--------------------------------
void hello()
{
    printf("Hello=");
}

void durand_kerner(int n,int argc, char *argv[]) {

    //-----------------------------------MPI--------------------------------
    int no_tasks, taskid, no_workers, source, workers, mtype,
no_elements_iter, no_elements, no_elements_left, index, i, j, k,rc, ack;
    double start,end;

    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &no_tasks);
    if (no_tasks < 2)
    {
        printf("Available Processors =%d\n",no_tasks);
        printf("Program is Terminated since there are less than 2
threads\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
        exit(1);
    }
    char pro_name[MPI_MAX_PROCESSOR_NAME];
    int length;
    MPI_Get_processor_name(pro_name,&length);

    printf("Processor name = %s, rank %d out of %d
processors\n",pro_name,taskid,no_tasks);
    no_workers = no_tasks - 1;

//----------------------------------MASTER-------------------------------

    if(taskid==0)
    {   double x,y;
        while(scanf("%lf %lf",&x,&y) == 2)  {
        cList[n] = (x + y*I);
        n++;
    }
        cList[n]=(1 + 0*I);
        start=MPI_Wtime();
        float st,total;

        R = 1 + max_cof(cList,n);

        calc_theta(n);

        int k;
        index=0;

        for(int i=1;i<=no_workers;i++)
        {
            MPI_Send(&max_iter, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
            MPI_Send(&n, 1, MPI_INT, i,2, MPI_COMM_WORLD);
            MPI_Send(&cList, n, MPI_DOUBLE_COMPLEX, i,3, MPI_COMM_WORLD);
        }


        for(k=1;k <= max_iter;k++)
        {
                deltaZMax = 0;

            no_elements = n / no_workers;
            no_elements_left = n % no_workers;

            index = 0;
            mtype = FROM_MASTER;

            for (workers = 1; workers <= no_workers; workers++)
            {

                no_elements_iter = (workers == no_workers) ? no_elements +
no_elements_left : no_elements;

                MPI_Send(&index, 1, MPI_INT, workers, 4, MPI_COMM_WORLD);
                MPI_Send(&no_elements_iter, 1, MPI_INT, workers, 5,
MPI_COMM_WORLD);
                MPI_Send(&z, n, MPI_DOUBLE_COMPLEX, workers, 6,
MPI_COMM_WORLD);


                index += no_elements_iter;

            }

            mtype = FROM_WORKER;
            for (i = 1; i <= no_workers; i++)
            {
                source = i;

                MPI_Recv(&index, 1, MPI_INT, source, 7, MPI_COMM_WORLD,
&status);
                MPI_Recv(&no_elements_iter, 1, MPI_INT, source, 8,
MPI_COMM_WORLD, &status);
                MPI_Recv(&deltaZ[index], no_elements_iter,
MPI_DOUBLE_COMPLEX, source,9, MPI_COMM_WORLD, &status);
```

```
        }

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

            if(cabs(deltaZ[j]) > deltaZMax) {
                deltaZMax = cabs(deltaZ[j]);
            }
        }
                            ack=1;
            if(deltaZMax <= epsilon) {
                ack=0;
            }
            printf("Iteration No=%d Zmax=%f\n",k,deltaZMax);

        mtype = FROM_MASTER;

        for (workers = 1; workers <= no_workers; workers++)
        {

            MPI_Send(&ack, 1, MPI_INT, workers, 10, MPI_COMM_WORLD);

        }
        if(ack==0)
        break;

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

}
//----------------------WORKERS-------------------------------
if(taskid>0)
```

```
    {

        mtype = FROM_MASTER;

        MPI_Recv(&max_iter, 1, MPI_INT, MASTER, 1, MPI_COMM_WORLD,
&status);

        MPI_Recv(&n, 1, MPI_INT, MASTER, 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&cList, n, MPI_DOUBLE_COMPLEX, MASTER, 3, MPI_COMM_WORLD,
&status);

        for(int l=0;l<max_iter;l++)
        {

            MPI_Recv(&index, 1, MPI_INT, MASTER, 4, MPI_COMM_WORLD,
&status);

            MPI_Recv(&no_elements_iter, 1, MPI_INT, MASTER, 5,
MPI_COMM_WORLD, &status);

            MPI_Recv(&z, n, MPI_DOUBLE_COMPLEX, MASTER, 6, MPI_COMM_WORLD,
&status);

            for(int j=index;j < index+no_elements_iter;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);

            }

            mtype = FROM_WORKER;

            MPI_Send(&index, 1, MPI_INT, MASTER, 7, MPI_COMM_WORLD);

            MPI_Send(&no_elements_iter, 1, MPI_INT, MASTER, 8,
MPI_COMM_WORLD);

            MPI_Send(&deltaZ[index], no_elements_iter, MPI_DOUBLE_COMPLEX,
MASTER, 9, MPI_COMM_WORLD);

            mtype = FROM_MASTER;
            MPI_Recv(&ack, 1, MPI_INT, MASTER, 10, MPI_COMM_WORLD,
&status);

            if(ack==1)
            {
                continue;
            }
            else
            l=max_iter+1;
        }

    }
    MPI_Finalize();
}


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 printfile(double complex cList[],int n,int k,float st)
{
        FILE *fp;
        fp = fopen("project_roots_mpi.txt", "w");
        fprintf(fp,"Durand Kerner Serial 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);
}
```