HIGH PERFORMANCE COMPUTING REPORT

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

Problem Statement:

Parallelize Durand-Kerner Algorithm based on solving polynomial equations Using Parallel Computing APIs like OpenMP,MPI and CUDA.

What are polynomial equations ?

An equation formed with variables, exponents, and coefficients together with operations and an equal sign is called a polynomial equation.It has different exponents. The higher one gives the degree of the equation.Usually, the polynomial equation is expressed in the form of $a_n(x^n)$.

- a is the coefficient
- x is the variable
- n is the exponent

On putting the values of a and n, we get a polynomial function of degree n.
Example:

$$F(x) = \mathrm{a_n}\left(\mathrm{x^n}\right) = a_n x^n + \ldots + a_0 = 0$$

Note:

- ➢ A polynomial equation does not contain a non-integer exponent.
- ➢ A polynomial equation supports algebraic operations only variables. For Example (2/x+3)=0 is not a polynomial.

What is the Durand-Kerner Algorithm and how does it work? :

Durand–Kerner method, is a root-finding algorithm for solving polynomial equations. In other words, the method can be used to solve numerically the equation *f*(*x*) = 0, where *f* is a given polynomial, which can be taken to be scaled so that the leading coefficient is 1.

Let us consider the equation of degree four and can be generalized to other higher degree.Let the polynomial f be defined by

$$f(x) = x^4 + ax^3 + bx^2 + cx + d$$

for all x,where :
a,b,c,d are the coefficients.

Let the (complex) numbers *P*, *Q*, *R*, *S* be the roots of this polynomial *f*.Then

$$f(x) = (x - P)(x - Q)(x - R)(x - S)$$

for all *x*. One can isolate the value *P* from this equation

$$P = x - \frac{f(x)}{(x-Q)(x-R)(x-S)}.$$

So using fixed-point iteration and mathematical induction.Let P be x1 and x be x_0 which Implies

$$x_1 := x_0 - \frac{f(x_0)}{(x_0-Q)(x_0-R)(x_0-S)},$$

it is strongly stable in that every initial point $x_0 \neq Q, R, S$ delivers after one iteration the root $P = x_1$.

Furthermore, if one replaces the zeros Q, R and S by approximations $q \approx Q$, $r \approx R$, $s \approx S$, such that q, r, s are not equal to P, then P is still a fixed point of the perturbed fixed-point iteration

$$x_{k+1} := x_k - \frac{f(x_k)}{(x_k-q)(x_k-r)(x_k-s)},$$

since

$$P - \frac{f(P)}{(P-q)(P-r)(P-s)} = P - 0 = P.$$

Note that the denominator is still different from zero. This fixed-point iteration is a contraction mapping for x around P.

The clue to the method now is to combine the fixed-point iteration for P with similar iterations for Q, R, S into a simultaneous iteration for all roots.

Initialize p, q, r, s:

$p_0 := (0.4 + 0.9i)^0$,

$q_0 := (0.4 + 0.9i)^1$,

$r_0 := (0.4 + 0.9i)^2$,

$s_0 := (0.4 + 0.9i)^3$.

There is nothing special about choosing $0.4 + 0.9i$ except that it is neither a real number nor a root of unity.Make the substitutions for $n = 1, 2, 3, \dots$ i.e,

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{(p_{n-1}-q_{n-1})(p_{n-1}-r_{n-1})(p_{n-1}-s_{n-1})},$$

$$q_n = q_{n-1} - \frac{f(q_{n-1})}{(q_{n-1} - p_n)(q_{n-1} - r_{n-1})(q_{n-1} - s_{n-1})},$$

$$r_n = r_{n-1} - \frac{f(r_{n-1})}{(r_{n-1} - p_n)(r_{n-1} - q_n)(r_{n-1} - s_{n-1})},$$

$$s_n = s_{n-1} - \frac{f(s_{n-1})}{(s_{n-1} - p_n)(s_{n-1} - q_n)(s_{n-1} - r_n)}.$$

Re-iterate until the numbers *p*, *q*, *r*, *s* essentially stop changing relative to the desired precision. They then have the values *P*, *Q*, *R*, *S* in some order and in the chosen precision. So the problem is solved.

Note that complex number arithmetic must be used, and that the roots are found simultaneously rather than one at a time.

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.
```

Why Should the algorithm be parallelized / Observation?

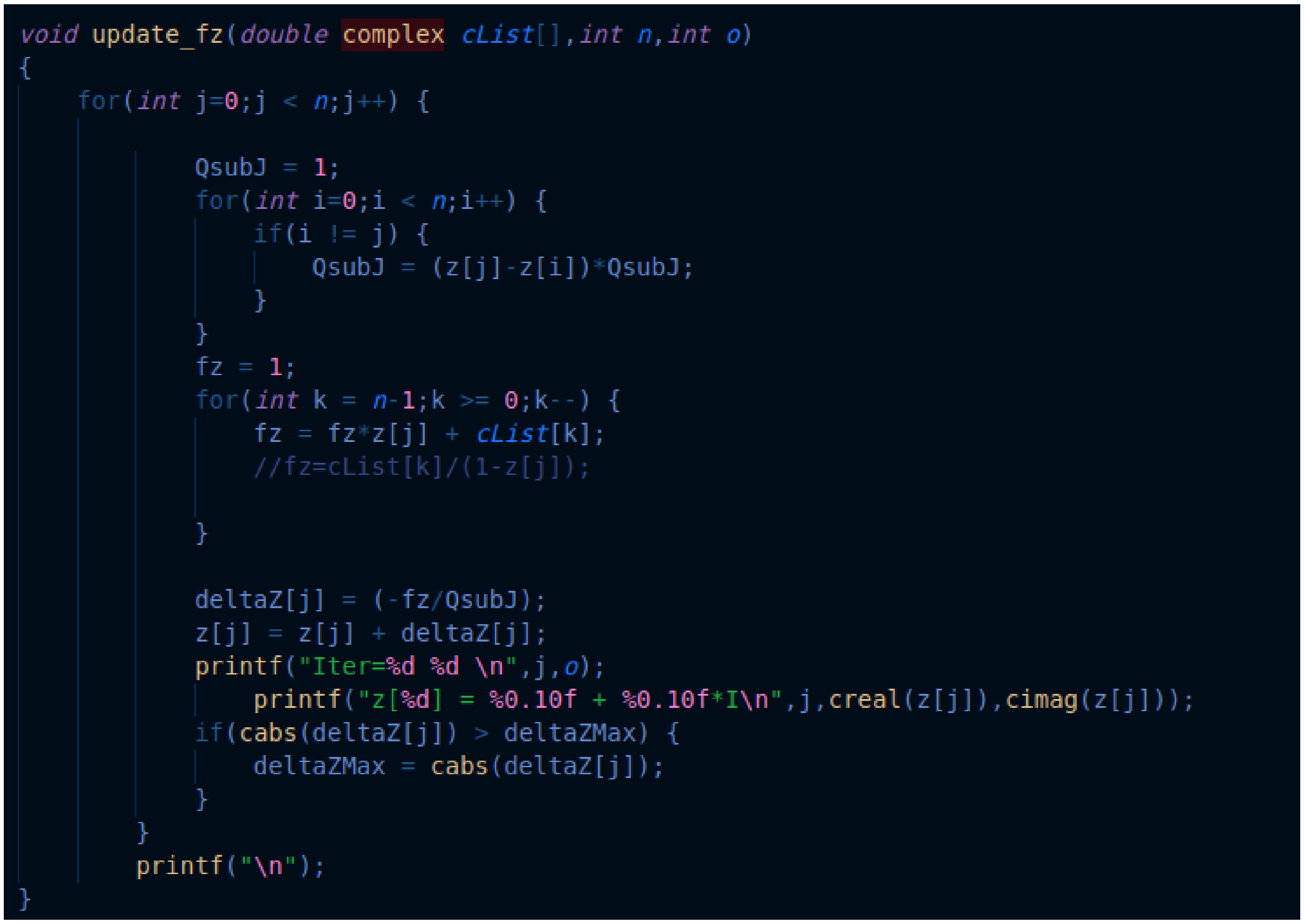
- ➢ The algorithm calculates the value of the function for all the n roots for kmax time.
- ➢ The initial value calculation is independent of the previous value .
- ➢ The above two parts of the algorithm can be parallelized.
- ➢ The Algorithm can become computationally intensive if the coefficients are complex numbers and the polynomial degree is high.

Applications/API Used:

- OpenMP
- MPI
- CUDA

Code Balancing:

Code Balancing is estimated for the critical section of this code which is updating the roots at each iteration and continues iterating until either roots converge or max no of iterations are reached.Assuming that there are ‘n’ roots,Let’s proceed with Code Balancing:



```
void update_fz(double complex cList[],int n,int o)
{
    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];
                //fz=cList[k]/(1-z[j]);

            }

            deltaZ[j] = (-fz/QsubJ);
            z[j] = z[j] + deltaZ[j];
            printf("Iter=%d %d \n",j,o);
                printf("z[%d] = %0.10f + %0.10f*I\n",j,creal(z[j]),cimag(z[j]));
            if(cabs(deltaZ[j]) > deltaZMax) {
                deltaZMax = cabs(deltaZ[j]);
            }
        }
        printf("\n");
}
```

No of words :

= 8*n(QsubJ : 4 for real part of the number and 4 for imaginary part of the number) + 8*n(fz)+6(deltaZ)+6(z[j]) + 4(Read Absolute Value of deltaZ) + 4 (assuming deltaZmax is updated)

=16*n+20

No of FLOPS per iteration:

= [2+6](QsubJ)*n + [6{For Complex no Multiplication}+2{For Complex no Additon}](fz)*n + 6(deltaZ) + 2 (z[j])

=16*n+8

Code Balance :

= Words / FLOPS

= 1 (for sufficiently Large values of n)

No of flops supported by the system:

=no of cores x clock frequency x double-precision calculations

=4 x 2.5 x 4

= 40 Gflops/sec

Machine Balance:

= 2.5 / 40

= 0.0625