```
////////////////////////////////////////////////
//  Assignment #2
//  Jered Stevens
//  Parallel Programming CMPS 4563 - Colmenarez
//  Date:   03/28/2024
//  ********************************************
//  Compile by typing the following or copying and pasting into
terminal:
//  mpicc -o Stevens16_1024.exe Stevens_MPIVer_16_1024.c
//
//  Run by typing or copy and pasting the following:
//  sbatch StevensParallel16_1024Script
//
//  This program computes the fft of an array of
//  complex numbers, taking the information from
//  the time domain to the frequency domain.
//
//  The size of the array must be a power of 2 to
//  function properly. The array size in this program
//  2^10
//
//  The program performs the fft calculation 3 times
//  in parallel and records the average amount of time
//  taken to complete. Then it shows the results of
//  the first 11 elements of the output array along with
//  the average time taken.
////////////////////////////////////////////////

#include <time.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define PI 3.141592653589793
#define N 1024 // Size of FFT, must be power of 2

/****************************************
 * Struct Name: Complex()
 *
 * Elements: double real, double imaginary
 *
 * Description: Represents a complex number with a
 *  real part and imaginary part.
 ****************************************/
typedef struct
{
    double real;
    double imaginary;
```

```
} Complex;

/*****************************************
 * Function Name: addComp()
 *
 * Parameters: Complex lhs, Complex rhs
 *
 * Description: Adds two complex numbers. Each parameter represents a
 * complex number with a real and imaginary part.
 *
 * Returns a Complex structure representing the sum of the two input
complex numbers.
 ******************************************/
Complex addComp(Complex lhs, Complex rhs)
{
    Complex result;
    result.real = lhs.real + rhs.real;
    result.imaginary = lhs.imaginary + rhs.imaginary;
    return result;
}

/*****************************************
 * Function Name: subComp()
 *
 * Parameters: Complex lhs, Complex rhs
 *
 * Description: Subtracts the second complex number (rhs)
 *  from the first complex number (lhs).
 *
 * Each parameter is a complex number. The function returns a new
 *  Complex struct representing the difference.
 ******************************************/
Complex subComp(Complex lhs, Complex rhs)
{
    Complex result;
    result.real = lhs.real - rhs.real;
    result.imaginary = lhs.imaginary - rhs.imaginary;
    return result;
}

/******************************************
 * Function Name: multComp()
 *
 * Parameters: Complex lhs, Complex rhs
 *
 * Description: Multiplies two complex numbers. Parameters lhs and
 *  rhs are the complex numbers to be multiplied.
 *
 * Returns a Complex struct representing the product of the
 *  two input complex numbers.
```

```
 ****************************************/
Complex multComp(Complex lhs, Complex rhs)
{
    Complex result;
    result.real = (lhs.real * rhs.real) - (lhs.imaginary *
rhs.imaginary);
    result.imaginary = lhs.real * rhs.imaginary + lhs.imaginary *
rhs.real;
    return result;
}

/****************************************
 * Function Name: fft()
 *
 * Parameters: Complex* data, Complex* output
 *
 * Description: Computes the Fast Fourier Transform (FFT) of an
 *  array of complex numbers. 'data' is a pointer to the input array
 *  of complex numbers, and 'output' is a pointer to the array where
 *  the FFT result will be stored.
 *
 * This function does not return a value but fills the 'output'
 *  array with the FFT result.
 ****************************************/
void fft(Complex *data, Complex *output, int localStart, int localEnd)
{
    for (int i = localStart; i < localEnd; i++)
    {
        Complex twiddle = {0, 0};
        Complex evenSum = {0, 0};
        Complex evenDataX = {0, 0};
        Complex eNumber = {0, 0};
        Complex oddSum = {0, 0};
        Complex oddDataX = {0, 0};
        Complex temp = {0, 0};

        // Caculate Twiddle Factor
        twiddle.real = cos((2 * PI * i) / N);
        twiddle.imaginary = -sin((2 * PI * i) / N);

        for (int j = 0; j < (N / 2); j++)
        {
            // Calculate sum of even elements in data
            evenDataX.real = data[2 * j].real;
            evenDataX.imaginary = data[2 * j].imaginary;
            eNumber.real = cos(((2 * PI) / (N / 2)) * i * j);
            eNumber.imaginary = -sin(((2 * PI) / (N / 2)) * i * j);
            temp = multComp(evenDataX, eNumber);
            evenSum = addComp(temp, evenSum);
```

```
            // Calculate sum of odd elements in data
            oddDataX.real = data[(2 * j) + 1].real;
            oddDataX.imaginary = data[(2 * j) + 1].imaginary;
            oddSum = addComp(multComp(oddDataX, eNumber), oddSum);
        }

        output[i].real = evenSum.real;
        output[i + (N / 2)].real = evenSum.real;
        output[i].imaginary = evenSum.imaginary;
        output[i + (N / 2)].imaginary = evenSum.imaginary;
        output[i] = addComp(multComp(twiddle, oddSum), output[i]);
        output[i + (N / 2)] = subComp(multComp(twiddle, oddSum),
output[i + (N / 2)]);
    }
}

int main()
{
     // Array for storing input data
    Complex data[N] = {{3.6, 2.6}, {2.9, 6.3}, {5.6, 4}, {4.8, 9.1},
{3.3, 0.4}, {5.9, 4.8}, {5, 2.6}, {4.3, 4.1}};

    // Array for results
    Complex output[N] = {{0, 0}};

    // MPI Variable
    int commSz;
    int myRank;
    int elementsPerProcess;
    int localStart;
    int localEnd;

    // Variables for timing
    struct timeval start, end;
    long stopwatchArray[3];
    long stopwatch;
    long averageTime;

    // Start up MPI
    MPI_Init(NULL, NULL);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &commSz);

    // Get my rank among all the processes
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    // # of elements per core -> (Samples/2) / number of cores
    elementsPerProcess = (N / commSz / 2);
    localStart = myRank * elementsPerProcess;
```

```
    localEnd = localStart + elementsPerProcess - 1;

    // Run 3 times
    for (int executions = 0; executions < 3; executions++)
    {
        // Reset all output values to 0
        for (int reset = 0; reset < N; reset++){
            output[reset].real = 0;
            output[reset].imaginary = 0;
        }

        // Start timer
        if (myRank == 0)
        {
            gettimeofday(&start, NULL);
        }

        // Do the thing
        fft(data, output, localStart, localEnd);

        // Make sure everyone is done doing the thing
        MPI_Barrier(MPI_COMM_WORLD);

        // Stop the timer
        if (myRank == 0)
        {
            gettimeofday(&end, NULL);
        }

        // Set execution time to a variable
        if(myRank == 0) {

             stopwatch = ((end.tv_sec*1000000 + end.tv_usec) -
(start.tv_sec*1000000 + start.tv_usec));

            // Add stopwatch to list of times for average
            stopwatchArray[executions] = stopwatch;
        }
    }

    // Print the results
     if (myRank == 0)
        {
            printf("TOTAL PROCESSED SAMPLES: %d\n", N);

printf("======================================================\n");
            for (int i = 0; i <= 10; i++)
            {

                printf("XR[%d]: %f      XI[%d]: %f\n", i,
```

```
output[i].real, i, output[i].imaginary);

printf("======================================================\n");
            }

            averageTime = (stopwatchArray[0] + stopwatchArray[1] +
stopwatchArray[2]) / 3;
            printf("Average execution time is: %ld usec",
averageTime);
        }
    MPI_Finalize();

    return 0;
}
```