Министерство образования и науки РФ
Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий - ИКНТ
Иссуственный Интеллект и Машинное Обучение

Тема:

ЧИСЛЕННО-ОПРЕДЕЛЕННЫЙ ИНТЕГРАЛ ПО ФОРМУЛЕ НЬЮТОНА-КОТЕСА

(NUMERICAL DEFINITE INTEGRAL BY THE NEWTON-COTES FORMULA)

Проект по дисциплине «ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ ДЛЯ СУПЕРКОМПЬЮТЕР»

Выполнил:
студент гр. 3540201/20301 Эспинола Ривера Хольгер Э.
подпись, дата

Проверил:
к.т.н., доцент Лукашин А.А
подпись, дата

Санкт-Петербург
2023

1. FORMULATION OF THE PROBLEM

1. Select a task and work out the implementation of an algorithm that allows parallelization into several threads / processes.
2. Develop tests to check the correctness of the algorithm (input data, output data, code for comparing results). To prepare test sets, you can use mathematical packages, for example, MATLAB (available in the Supercomputational center class and on the SCC itself).
3. Implement algorithms using selected technologies.
4. Conduct a study on the effect of using multi-core / multi-threading / multi-processing on the SCC, varying nodes from 1 to 4 (for MPI) and varying the number of processes / threads.
5. Prepare an electronic report.

APPLICATION TASK:

Definite integrals by the Newton-Cotes formula.

2. MATHEMATICAL DEFINITIONS

Definition of Rieman's Integral

The Rieman's integral is defined by the next formula:

$I = \int_a^b f(x)dx$, where using the fundamental theorem of calculus for F(x) antiderivative, we get:

Definition 1.

$$I = \int_a^b f(x)dx = F(b) - F(a)$$

Using the geometrical interpretation of Rieman's integral, we have the generic definition of the definitive Rieman's integral, like a sum of the infinitesimal areas of rectangles:

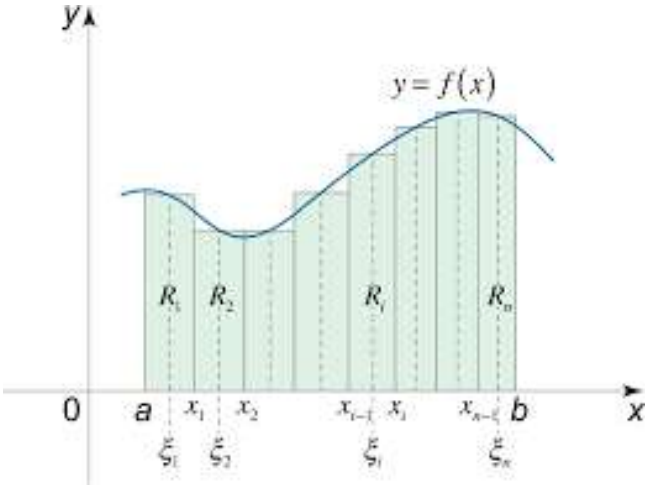


Image 1. Geometrical representation of Rieman's integral

Definition 2.

$$I = \lim_{n\to\infty} \sum_{i=1}^{n} f(\xi_i) \cdot \left(x_i - x_{i-1}\right)$$

The problem of Numerical Methods is to calculate an approximate version of the analytical Rieman's integral. Thus, we formulate this integral like a sum of finite rectangle areas.

Definition 3.

$$I \approx \sum_{i=1}^{n} f(\xi_i) \cdot (x_i - x_{i-1})$$

To calculate this approximate integral, we define a finite number of intervals, which divide the original interval [a, b] in minor intervals, using the formula:

$$h = \frac{b-a}{n}$$

Where:

- n: number of partitions/subintervals
- a: lower limit of definite integral
- b: upper limit of definite integral
- h: step to next interval

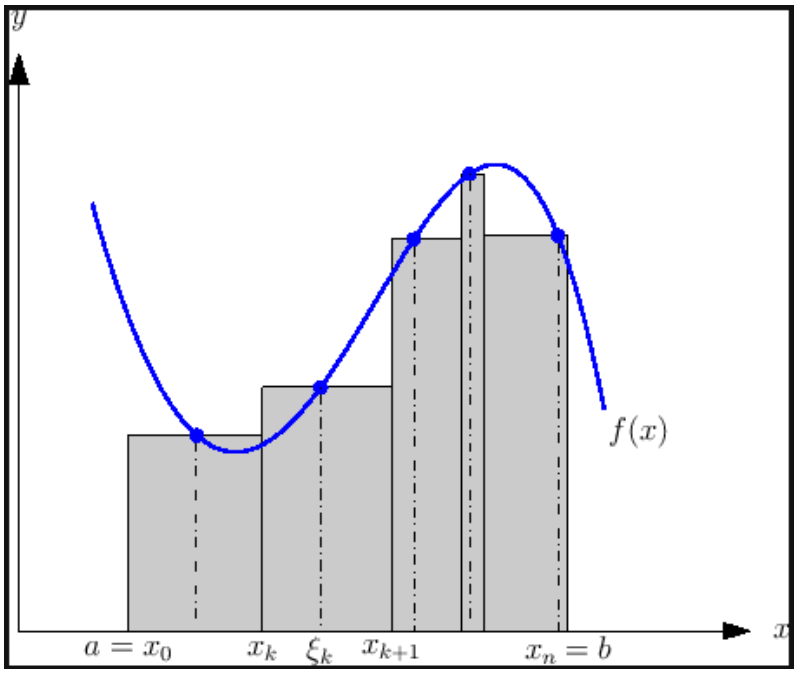


Image 2. Partitions of approximate Rieman's integral

Generic Newton-Cotes Integration Algorithm

The main idea to use the method of Newton-Cotes to calculate the numerical approximate definitive integral, is divide the original interval [a, b] in several subintervals, where in each subinterval, we will calculate the approximate partial-integral. The partial integral for each subinterval is given by the formula:

Definition 4.

$$I_{part} = \int_{x_0}^{x_k} f(x)dx = A_k \cdot h \cdot \sum_{i=0}^{k} C_i^{(k)} \cdot f(x_i) + \underset{error}{\varepsilon}$$

Where:

I_part: is the result of calculating the partial-integral in subinterval $[x_0, x_k]$.

f(x): is the function to need integrate

k: degree of polynomial quadrature

A_k: coefficient for the k-th quadrature rule

$C_i^{(k)}$: vector of Newton-Cotes parameters

x_i: specific point which belongs subinterval $[x_0, x_k]$.

$f(x_i)$: image of the point x_i in the function f

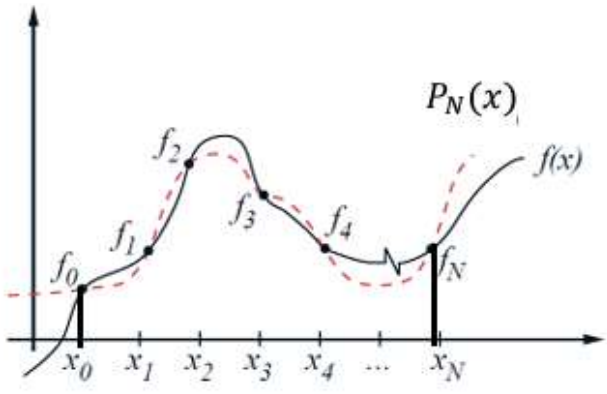


Image 3. Quadrature of Newton-Cotes

The coefficients A_k and $C_i^{(k)}$ can be calculated using Lagrange interpolating polynomials, where k is the polynomial degree of Newton-Cotes quadrature.

In the project, we consider quadrature from k = 1 (linear polynomial) to k = 10 (polynomial with 10-th degree). The values of the coefficients A_k and $C_i^{(k)}$ for the Newton-Cotes formulas is given by:

Table 1. List of Newton-Cotes coefficients

k	Ak	$C_0^{(k)}$	$C_1^{(k)}$	$C_2^{(k)}$	$C_3^{(k)}$	$C_4^{(k)}$	$C_5^{(k)}$
		$C_6^{(k)}$	$C_7^{(k)}$	$C_8^{(k)}$	$C_9^{(k)}$	$C_{10}^{(k)}$	
0	0	0					
1	1/2	1	1				
2	1/3	1	4	1			
3	3/8	1	3	3	1		
4	2/45	7	32	12	32	7	
5	5/288	19	75	50	50	75	19
6	1/140	41	216	27	272	27	216
		41					
7	7/17280	751	3577	1323	2989	2989	1323
		3577	751				
8	4/14175	989	5888	-928	10496	-4540	10496
		-928	5888	989			
9	9/89600	2857	15741	1080	19344	5778	5778
		19344	1080	15741	2857		
10	5/299376	16067	106300	-48525	272400	-260550	427368
		-260550	272400	-48525	106300	16067	

With the coefficients of Newton-Cotes defined above, each case of partial-integrate will be solved using a formula in definition 4 with help of the coefficients.

To calculate the final value of the approximate integral, we define the number of iterations/quadrature (in the future, this will be too the number of processes) and for each iteration/quadrature, we will take this partial-integrals and sum all, to obtain the final result: the approximate integral of function f(x) between [a, b].

Definition 5.

$$I_{total} = \sum_{i=0}^{q} H_i$$

Where:

I_{total}: final result of approximate integral

H_i: result of partial-integral for i-th subinterval

q: number of quadratures/ subintervals

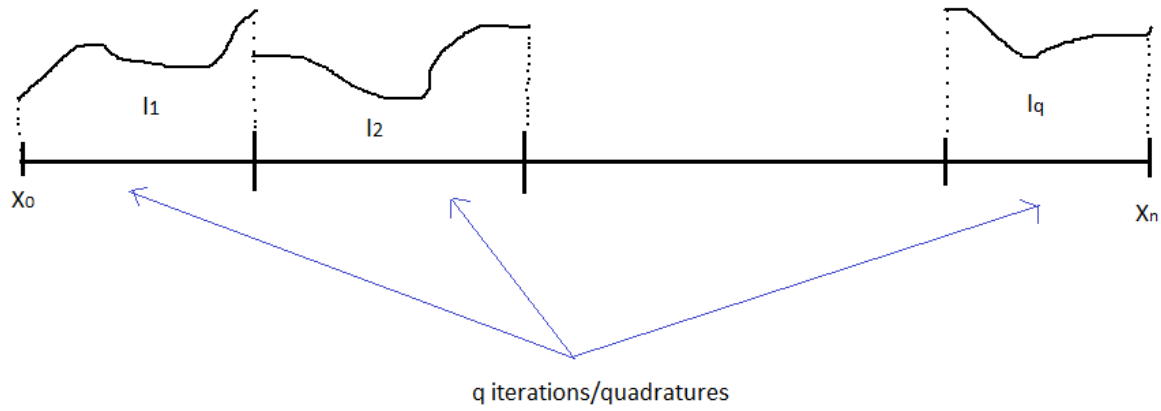


Image 4. Division of original integral in “q” partial integrals

3. METHODOLOGY

1) Dictionary of variables

k: degree of quadrature
[datatype: int]
q: number of quadratures/ number of processes
[datatype: int]
n: number of partitions/intervals (defined like n = k*q)
[datatype: int]
coeff_newton_cotes: structure of newton-cotes coefficients

- pvec_ncotes_k: vector of positions, which represent degree of quadrature
  [datatype: double vector]
- pvec_ncotes_Ak: vector of characteristic coefficients for h
  [datatype: double vector]
- pmatrix_ncotes_Ci: matrix of newton-cotes coefficients
  [datatype: double matrix]

part_int: result of partial integrals for each quadrature [datatype: double]
Formula:
part_int{f(x), lim = [low = x0, upper = xk]} =

Ak * h * sum{i = 0...k} {Ci_k * f(xi)}

a: lower limit of definite integral
[datatype: double]
b: upper limit of definite integral
[datatype: double]
expression: function to integrate
[datatype: char vector]
I_exact: exact value of integral
[datatype: double]

2) Strategy of solution

In the initial step, need to split the initial dataset and massive distribute the slices of data for each process, from master process (process 0) to slave's processes (process 1 until q).

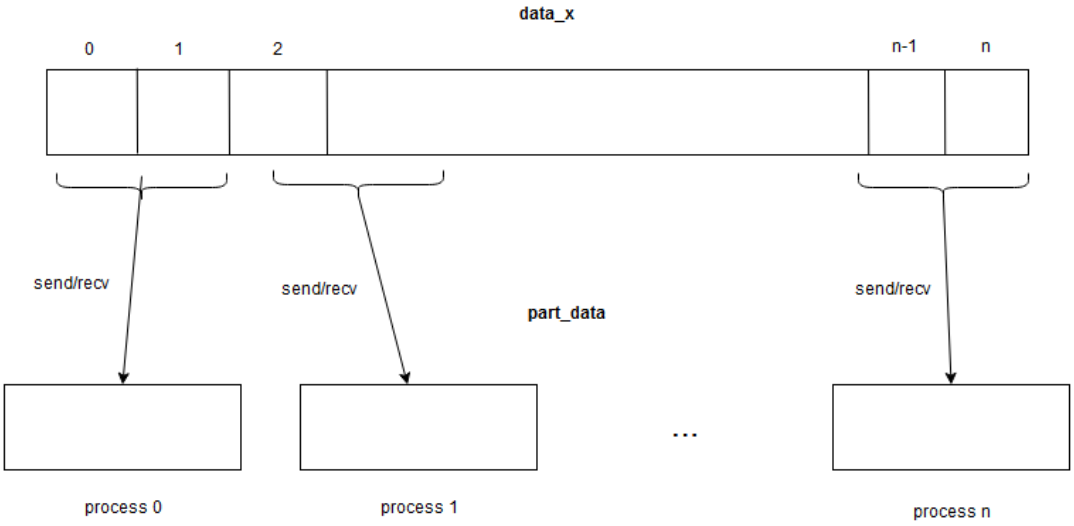


Image 5. Massive distribution of data for each slave process

In the final step for calculate the approximate integral, we do a process of reduce, which sum all the partial-integrals belongs slaves processes and accumulate the results in the master process 0.

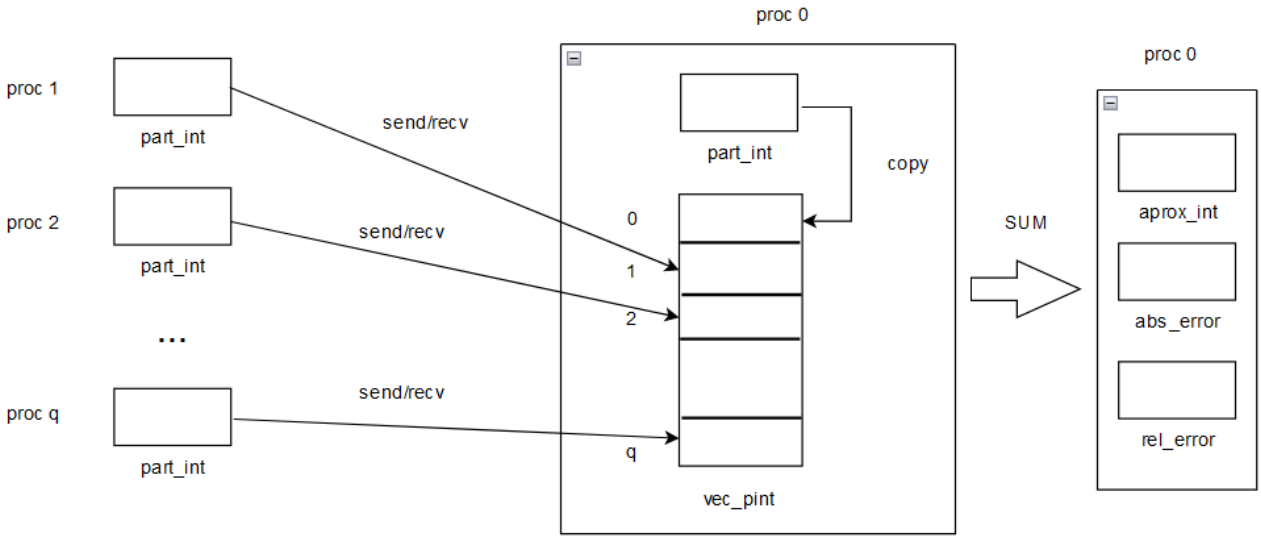


Image 6. Final step of integral calculation sums

3) Code implementation in MPI + Python

File: newton_cotes_final1.py

```
###############################################################################
   #      PARALLEL PROGRAMMING - GENERIC-NUMERICAL SOLUTION FOR NEWTON-COTES
                                INTEGRATION    #
###############################################################################

# SOLUTION DESIGNED IN MPI + PYTHON
# dictionary of variables
# k: degree of quadrature
# q : number of quadratures/ number of processes
# n: number of intervals/partitions (defined like n = k*q)
# coeff_newton_cotes: table of newton-cotes coefficient
#   - Ak: coefficient of h
#   - Ci: vector of weights
# partial integrals of each quadrature:
# I_part{f(x), lim = [low = x0, upper = xk]} = Ak * h * sum{i = 0..k} {Ci_k *
f(xi)}
# a: lower limit of definite integral
# b: upper limit of definite integral
# expresion: function to integrate
# I_exact: exact value of integral

# import packages
from mpi4py import MPI
import numpy as np
import math

# generic parameters
comm = MPI.COMM_WORLD
numproc = comm.Get_size()
myid = comm.Get_rank()

# function to read files
def read_file():
    # read line for line the text file
    file_name = str(input("Name of file: "))
    with open(file_name) as file:
        lines = [line.strip() for line in file]
    # define input variables
    a = float(eval(lines[0]))
    b = float(eval(lines[1]))
    k = int(eval(lines[2]))
    expression = lines[3]
    I_exact = float(eval(lines[4]))
    return a, b, k, expression, I_exact
```

```
########################################
#    MAIN: process of paralelization    #
########################################

##### paralelization process 1: broadcast of input-data #####

comm.Barrier()

# master process 0
if(myid == 0):
    # get the variables
    print("Name of file: ")
    a, b, k, expression, I_exact = read_file()
    q = numproc

    # Dictionary of Newton-Cotes coefficients
    #link: https://mathworld.wolfram.com/Newton-CotesFormulas.html

    # quadratures: k = 1...10 (from linear functions to power 10)
    newton_cotes_coeff = {
        1: {'Ak': 1/2, 'Ci': [1, 1]},
        2: {'Ak': 1/3, 'Ci': [1, 4, 1]},
        3: {'Ak': 3/8, 'Ci': [1, 3, 3, 1]},
        4: {'Ak': 2/45, 'Ci': [7, 32, 12, 32, 7]},
        5: {'Ak': 5/288, 'Ci': [19, 75, 50, 50, 75, 19]},
        6: {'Ak': 1/140, 'Ci': [41, 216, 27, 272, 27, 216, 41]},
        7: {'Ak': 7/17280, 'Ci': [751, 3577, 1323, 2989, 2989, 1323, 3577,
751]},
        8: {'Ak': 4/14175, 'Ci': [989, 5888, -928, 10496, -4540, 10496, -928,
5888, 989]},
        9: {'Ak': 9/89600, 'Ci': [2857, 15741, 1080, 19344, 5778, 5778, 19344,
1080, 15741, 2857]},
        10: {'Ak': 5/299376, 'Ci': [16067, 106300, -48525, 272400, -260550,
427368, -260550, 272400,
                                    -48525, 106300, 16067]}
    }

elif(myid != 0):    # slave processes
    # initialization of variables
    a = b = k = q = expression = I_exact =  None
    newton_cotes_coeff = {}
    print("This is the process ", myid)

comm.Barrier()

# broadcast all arguments from process 0 to every process [1, ... numproc]
a = comm.bcast(a, root = 0)
b = comm.bcast(b, root = 0)
```

```
k = comm.bcast(k, root = 0)
q = comm.bcast(q, root = 0)
expression = comm.bcast(expression, root = 0)
I_exact = comm.bcast(I_exact, root = 0)
newton_cotes_coeff = comm.bcast(newton_cotes_coeff, root = 0)

# print the information that was obtained for each process
print(" ############## ")
print("Process ", myid, " ===> variables: ")
print("a = ", a)
print("b = ", b)
print("k = ", k)
print("q = ", q)
print("expression = ", expression)
print("I = ", I_exact)
print("Newton-cotes coeff = ", newton_cotes_coeff)

comm.Barrier()

###### paralelization process 2: send/recv of original data #######

if(myid == 0):
    # number of intervals
    n = k * q
    # compute step
    h = (b - a) / n
    # generate X
    data_x = np.linspace(start = a, stop = b, num = n + 1)

    # extract subsets of data for each process
    # send SLICE from process 0 ===> to process i
    for i in range(1, q):
        # send h for each process
        comm.send(h, dest = i)
        # split the data
        slice = data_x[i*k:(i+1)*k+1]
        # print the send
        print("Process ", myid, " sends ", (k+1), " values to process ", i, "
===> ", slice)
        # send each slice from process 0 ===> to process i
        comm.send(slice, dest = i)
    # partial-data from process 0
    part_data = data_x[0:(k+1)]
    print("Process ", myid, " have this data: ", part_data)
else:
    # initialize h
    h = None
    # process i <== receive h from process 0
    h = comm.recv(source = 0)
```

```
    # initialize partial-data
    part_data = np.empty(k + 1, dtype = np.float64)
    # process i <=== receive slice from process 0
    part_data = comm.recv(source = 0)
    # print the receiver
    print("Process ", myid, " <=== receive ", (k+1), " values from process 0:
", part_data)

comm.Barrier()

# each process show the corresponding partial-data
print(myid, " process ======> ", part_data)

comm.Barrier()

###############################################
#       Compute the partial-integral          #
###############################################

# internal interpreter for string-math expressions
def f(x):
    f = eval(expression)
    return f

# start clock
start_time = MPI.Wtime()

# initialize y
part_y = np.zeros(k + 1)

# calculate y
for i in range(k + 1):
    part_y[i] = f(part_data[i])

# calculate part-int
part_int = 0

for j in range(k + 1):
    part_int += newton_cotes_coeff[k]['Ci'][j] * part_y[j]
part_int = newton_cotes_coeff[k]['Ak'] * h * part_int

# stop clock
end_time = MPI.Wtime()

# count time
count_time = (end_time - start_time) * 1000

comm.Barrier()
```

```
print("Process ", myid, " has f(x) ===> ", part_y, " with partial-integral =
", part_int)
print("Time to calculate partial-integral = ", count_time, " ms")

comm.Barrier()

#####################################################
# FINAL STEP: CALCULATE THE APROXIMATE INTEGRAL  #
#####################################################

# master-process 0 ==> manage the computation of final-results of integral
if myid == 0:
    final_time = 0
    # define array of partial integral-sums and
    # initialize array with [0, ... 0]
    vec_pint = np.zeros(numproc, np.float64)
    # copy result of part-int in process 0 to vector of partial-integral-
sums[0]
    vec_pint[0] = part_int
    # define array of time-stamp
    vec_times = np.zeros(numproc, np.float64)
    # copy time-processing of process 0
    vec_times[0] = count_time
    # capture partial-integral results and times for each process
    # and save in vector of results in master-process
    for j in range(1, numproc):
        start_t2 = MPI.Wtime()
        vec_pint[j] = comm.recv(source = j)
        vec_times[j] = comm.recv(source = j)
        end_t2 = MPI.Wtime()
        final_time += (end_t2 - start_t2) * 1000
        print("Time to send/recv part-integrals = ", (end_t2 - start_t2) *
1000, " ms")
    start_t3 = MPI.Wtime()
    # finnaly, calculate the value of approximate-integral
    aprox_int = sum(vec_pint)
    # calculate absolute-error
    abs_error = abs(I_exact - aprox_int)
    # calculate relative-error
    rel_error = (abs_error / I_exact) * 100
    end_t3 = MPI.Wtime()
    # calculate final time
    final_time += sum(vec_times)
    final_time += (end_t3 - start_t3) * 1000
    print("Time to calculate approx-integral = ", (end_t3 - start_t3) * 1000,
" ms")
    # PRINT THE FINAL RESULTS
    print("***** FINAL RESULTS ***** from process ", myid)
    print("Integral ", expression, " with limits = ( ", a, ", ", b, ") ===> ")
```

```
    print("Exact-integral = ", I_exact)
    print("Approximate-integral = ", aprox_int)
    print("Absolute-error = ", abs_error)
    print("Relative-error = ", rel_error, "%")
    print("Total-time to compute aproximate integral = ", final_time, " ms")
    print("Finish!")
else:   # slave processes [1..q] ==> sends part_int (partial-results) ===> to
master-process 0
    comm.send(part_int, dest = 0)
    comm.send(count_time, dest = 0)

comm.Barrier()
```

[240 lines of code]

4) Code implementation in MPI + C

File: newton_cotes_vc14.c

Description: This main file have the Newton-Cotes algorithm and the process of parallelization in MPI + C

```
/*
    ================================================================================
PARALLEL PROGRAMMING IN C - GENERIC SOLUTION FOR NEWTON-COTES INTEGRATION    |
    ================================================================================
                                    VERSION 14.0

    Dictionary of variables:

    k: degree of quadrature
    q: number of quadratures/ number of processes
    n: number of partitions/intervals (defined like n = k*q)
    coeff_newton_cotes: structure of newton-cotes coefficients
        - k: position of vector of structure and represent degree of
quadrature
        - Ak: coefficient for h
        - Ci: vector of weights
    part_int: partial integrals for each quadrature
    part_int{f(x), lim = [low = x0, upper = xk]} = Ak * h * sum{i = 0...k}
{Ci_k * f(xi)}
    a: lower limit of definite integral
    b: upper limit of definite integral
    expression: function to integrate
    I_exact: exact value of integral

*/

#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>
#include "tinyexpr.h"
#define MAX 11
#define v 1
#define tag 100

// functions defined by programmer
void read_files(char name_file[40], char lines[5][200]);
```

```c
void input_variables(double *a, double *b, int *k, char expression[200],
                        double *I_exact, char lines[5][200]);
void print_results(double a, double b, int k, char expression[200], double
I_exact);
void define_parameters(double *x, double *y, double **z);
void print_vector(double *x);
void print_matrix(double **x);
double func(double a, char expression[200]);

int main(int argc, char *argv[])
{
    // variables related with MPI
    int myid, numproc;
    // variables related with input data
    double a, b, I_exact;
    int k, q;
    char expression[200];
    // variables related with file manager
    char name_file[40], lines[5][200];
    // variables related with newton-cotes
    double *pvec_ncotes_k, *pvec_ncotes_Ak, **pmatrix_ncotes_Ci;
    // variable to manage communication status
    MPI_Status status;
    // output variables
    double approx_int, abs_error, rel_error;

    // initialize parallel processing
    MPI_Init(&argc, &argv);
    // take the total number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &numproc);
    // take the number of the current process
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    // ******************** STEP 1: broadcast input-data
************************

    // allocate memory for newton-cotes parameters
    pvec_ncotes_k = (double *) calloc(MAX, sizeof(double));
    pvec_ncotes_Ak = (double *) calloc(MAX, sizeof(double));

    pmatrix_ncotes_Ci = (double **) calloc(MAX, sizeof(double *));

    for(int i=0; i<MAX; i++)
    {
        pmatrix_ncotes_Ci[i] = (double *) calloc(MAX, sizeof(double));
    }

    // master process 0 ==> read the input data
    if(myid == 0)
```

```
    {
        printf("Name of file: ");

        // request the name of the file
        scanf("%s", name_file);

        // read the txt file
        read_files(name_file, lines);

        // asign the values to input-variables
        input_variables(&a, &b, &k, expression, &I_exact, lines);
        q = numproc;

        // asign parameters of newton-cotes
        // newton-cotes parameters
        define_parameters(pvec_ncotes_k, pvec_ncotes_Ak, pmatrix_ncotes_Ci);

        // print the results of input-data readed
        if(v > 0)
        {
            printf("\n Master process 0 ");
            // print unidimensional variables
            print_results(a, b, k, expression, I_exact);
            // print newton-cotes variables
            printf("\n Newton-cotes k coefficients: \n");
            print_vector(pvec_ncotes_k);
            printf("\n Newton-cotes Ak coefficients: \n");
            print_vector(pvec_ncotes_Ak);
            printf("\n Newton-cotes Ci coefficients: \n");
            print_matrix(pmatrix_ncotes_Ci);
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    // broadcast all arguments from process 0 to every process [0 until
numproc]
    MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&q, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(expression, 200, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Bcast(&I_exact, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    //
    MPI_Bcast(pvec_ncotes_k, MAX, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(pvec_ncotes_Ak, MAX, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    for(int i=0; i<MAX; i++)
        MPI_Bcast(pmatrix_ncotes_Ci[i], MAX, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
    MPI_Barrier(MPI_COMM_WORLD);

    // print the information that was obtained for each process
    if(v > 0)
    {
        printf("********** Broadcast of input-data ***********");
        printf("\n Process %d ===> variables: \n", myid);
        print_results(a, b, k, expression, I_exact);
        //
        printf("\n Newton-cotes Ak coefficients: \n");
        printf("\n Coeff k: \n");
        print_vector(pvec_ncotes_k);
        printf("\n Coeff Ak: \n");
        print_vector(pvec_ncotes_Ak);
        printf("\n Coeff Ci: \n");
        print_matrix(pmatrix_ncotes_Ci);
    }

    MPI_Barrier(MPI_COMM_WORLD);


    // ********************* STEP 2: send/recv of original data
**********************

    // variables related with computation
    int n;
    double h, *data_x, *part_data, *slice;

    // master process ==> manage data partition
    if(myid == 0)
    {
        // number of intervals
        n = k * q;
        // calculate the steps
        h = (b-a)/n;
        // allocate memory for data_x
        data_x = (double *) calloc(n+1, sizeof(double));

        // generate full dataset data_x
        for(int i=0; i<n+1; i++)
        {
            *(data_x + i) = a + i*h;
        }

        // print dataset
        if(v>0)
        {
            printf("\n **** All data: *** \n");
```

```
            for(int i=0; i<n+1; i++)
            {
                printf("%f \t", *(data_x + i));
            }
            printf("\n");
        }

        // PROCESS OF SENDING ..................

        // extract subsets of data for each process
        // send SLICE from process 0 ==> to ptocess i
        for(int i=1; i<q; i++)
        {
            // send h for each process
            MPI_Send(&h, 1, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);

            // split the data in small pieces
            slice = (double *) calloc(k+1, sizeof(double));

            for(int j=0; j<k+1; j++)
            {
                *(slice + j) = *(data_x + i*k + j);
            }

            // send each slice from process 0 ==> to process i
            MPI_Send(slice, k+1, MPI_DOUBLE, i, tag,
MPI_COMM_WORLD);

            // print the send
            if(v>0)
            {
                printf("\n Process %d sent %d values to process %d  ===> \n",
myid, k+1, i);
                for(int j=0; j<k+1; j++)
                    printf("%f \t", *(slice +j));
            }
        }

        // allocate memory for part_data speciffied for process master 0
        part_data = (double *) calloc(k+1, sizeof(double));

        // fill partial-data for process 0
        for(int i=0; i<k+1; i++)
        {
            *(part_data + i) = *(data_x + i);
        }

        if(v>0)
        {
```

```
            printf("\n Process %d have this data: \n", myid);

            for(int i=0; i<k+1; i++)
                printf("%f \t", *(part_data + i));
            printf("\n");
        }

        // **************

    }else if(myid != 0){    // slaves processes ==> take the partial-data

        printf("\n Process %d \n", myid);

        // PROCESS OF RECEIVING.................................

        // process i <== receive h from process 0

        MPI_Recv(&h, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
        // initialize part-data
        part_data = (double *) calloc(k+1, sizeof(double));
        // process i <== receive slice from process 0
        MPI_Recv(part_data, k+1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);

        if(v>0)
        {
            printf("\n Process %d <== receive %d values from process 0: \n",
myid, k+1);

            for(int i=0; i<k+1; i++)
                printf("%f \t", *(part_data + i));
            printf("\n");
        }

        // *************************
    }

    // each process show the corresponding partial-data
    printf("\n ******************************** \n");
    printf("Data for process %d: \n", myid);

    for(int i=0; i<k+1; i++)
        printf("%f \t", *(part_data + i));

    printf("\n ******************************** \n");


    // ******************** STEP 3: Compute the partial integral
**********************
```

```
    double *part_y, part_int, start_time, end_time, count_time;

    // initialize timing of computation calculus
    start_time = MPI_Wtime();

    // evaluate expression in vector variable .....................

    // initialize y
    part_y = (double *) calloc(k+1, sizeof(double));

    // calculate y
    for(int i=0; i<k+1; i++)
    {
        *(part_y + i) = func(*(part_data + i), expression);
    }

    // calculate partial-integral
    part_int = 0;

    for(int j=0; j<k+1; j++)
        part_int += pmatrix_ncotes_Ci[k][j] * part_y[j];

    part_int = pvec_ncotes_Ak[k] * h * part_int;

    // ******************************************************

    // stop clock
    end_time = MPI_Wtime();

    // count time in ms
    count_time = (end_time - start_time) * 1000;

    MPI_Barrier(MPI_COMM_WORLD);

    printf("\n ++++++++++ PARTIAL RESULTS +++++++++ \n");
    printf("\n Process %d has f(x) = \n", myid);

    for(int i=0; i<k+1; i++)
        printf("%f \t", *(part_y + i));

    printf("\n Process %d has partial-integral = %f", myid, part_int);
    printf("\n Time to calculate partial-integral: %f ms", count_time);

    MPI_Barrier(MPI_COMM_WORLD);


    // ******************** STEP 4: Calculate the approximate integral
**********************
```

```
    // variables
    double final_time, *vec_pint, *vec_times, start_t2, end_t2, start_t3,
end_t3;

    // master process 0 ==> manage the computation of final-results of
numerical integral
    if(myid == 0)
    {
        // initialize time
        final_time = 0;

        // define array of partial-integral sums
        vec_pint = (double *) calloc(numproc, sizeof(double));

        // copy result of part-int in process 0 to partial-integral-sums[0]
        vec_pint[0] = part_int;

        // define array of time-stamps
        vec_times = (double *) calloc(numproc, sizeof(double));

        // copy the time-processing of processing 0
        vec_times[0] = count_time;

        // capture the partial-integral results and times for each process
        // and save in vector of results in master-process

        for(int j=1; j<numproc; j++)
        {
            // initialize time
            start_t2 = MPI_Wtime();

            // receive the partial-integral result corresponding to j-th
process
            MPI_Recv(&vec_pint[j], 1, MPI_DOUBLE, j, tag, MPI_COMM_WORLD,
&status);

            // receive the j-th speending time of process
            MPI_Recv(&vec_times[j], 1, MPI_DOUBLE, j, tag, MPI_COMM_WORLD,
&status);

            // finalize time
            end_t2 = MPI_Wtime();

            // calculate total time
            final_time += (end_t2 - start_t2) * 1000;

            // print time2
            printf("\n Time to send/recv part-integrals = %f ms", (end_t2 -
start_t2)*1000);
```

```
        }

        // initialize time
        start_t3 = MPI_Wtime();

        // Finnaly... calculate the approximate value of integral
        approx_int = 0;

        for(int i=0; i<numproc; i++)
        {
            approx_int += *(vec_pint + i);
        }

        // calculate absolute-error
        abs_error = fabs(I_exact - approx_int);

        // calculate relative-error
        rel_error = (abs_error/I_exact)*100;

        // finish time
        end_t3 = MPI_Wtime();

        // print time to calculate approx-integral
        printf("\n Time to calculate apprx. integral= %f ms \n", (end_t3 -
start_t3)*1000);

        // calculate final accumulate time
        final_time = 0;

        for(int i=0; i<numproc; i++)
        {
            final_time += *(vec_times + i);
        }

        final_time += (end_t3 - start_t3) * 1000;

        // print the final results
        printf("\n ****************************************** \n");
        printf("FINAL RESULTS from process %d \n", myid);
        printf("\n ****************************************** \n");

        printf("\n Integral %s with limits = (%f , %f) ========> ",
expression, a, b);
        printf("\n Exact-integral = %f", I_exact);
        printf("\n Approximate-integral = %.15f", approx_int);
        printf("\n Absolute-error = %.15f", abs_error);
        printf("\n Relative-error = %.15f %%", rel_error);
        printf("\n Total time to compute integral = %.8f ms", final_time);
```

```
        printf("\n ********* FINISH! ************");

    }else if(myid != 0){       // slave processes [1...q] ==> sends (part-res)
==> to master-process 0

        // send the partial-integrals-sum to master process 0
        MPI_Send(&part_int, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);

        // send the partial-integrals-sum
        MPI_Send(&count_time, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    printf("\n");
    MPI_Finalize();

    return 0;
}

// evaluate expression in function
double func(double a, char expression[200])
{
    double x;
    int err;

    // define the input variables
    te_variable vars[] = {"x", &x};

    // compile the math-expression
    te_expr *expr = te_compile(expression, vars, 1, &err);

    if(expr)
    {
        x = a;
        const double res = te_eval(expr);
        te_free(expr);
        return res;
    }else{
        return 0;
    }
}

// define function to read files
void read_files(char name_file[40], char lines[5][200])
{
    // open file
    FILE *file = fopen(name_file, "r");
    size_t len = 200;
```

```
    // allocate memory
    char *line = malloc(sizeof(char) * len);

    // check if file to read exits
    if(file == NULL)
    {
        printf("Can't open this file ... this file not exist \n");
        return;
    }

    // save each line in array of lines
    int i=0;

    while(fgets(line, len, file)!= NULL)
    {
        strcpy(lines[i], line);
        i++;
    }

    free(line);
}

void input_variables(double *a, double *b, int *k, char expression[200],
                        double *I_exact, char lines[5][200])
{
    *a = te_interp(lines[0], 0);
    *b = te_interp(lines[1], 0);
    *k = te_interp(lines[2], 0);
    strcpy(expression, lines[3]);
    *I_exact = te_interp(lines[4], 0);
}

void print_results(double a, double b, int k, char expression[200], double
I_exact)
{
    printf("\n a = %f", a);
    printf("\n b = %f", b);
    printf("\n k = %d", k);
    printf("\n expr = %s", expression);
    printf("\n I = %f", I_exact);
}

// define parameters
void define_parameters(double *x, double *y, double **z)
{
    double vec_ncotes_k[MAX] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for(int i=0; i<MAX; i++)
```

```
    {
        *(x+i) = vec_ncotes_k[i];
    }

    double vec_ncotes_Ak[MAX] = {
        0.0, 1.0/2.0, 1.0/3.0, 3.0/8.0, 2.0/45.0, 5.0/288.0,
        1.0/140.0, 7.0/17280.0, 4.0/14175.0, 9.0/89600.0, 5.0/299376.0
    };

    for(int i=0; i<MAX; i++)
    {
        *(y+i) = vec_ncotes_Ak[i];
    }

    double matrix_ncotes_Ci[MAX][MAX] = {
        {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {1.0, 4.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {1.0, 3.0, 3.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {7.0, 32.0, 12.0, 32.0, 7.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {19.0, 75.0, 50.0, 50.0, 75.0, 19.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {41.0, 216.0, 27.0, 272.0, 27.0, 216.0, 41.0, 0.0, 0.0, 0.0, 0.0},
        {751.0, 3577.0, 1323.0, 2989.0, 2989.0, 1323.0, 3577.0, 751.0, 0.0,
0.0, 0.0},
        {989.0, 5888.0, -928.0, 10496.0, -4540.0, 10496.0, -928.0, 5888.0,
                    989.0, 0.0, 0.0},
        {2857.0, 15741.0, 1080.0, 19344.0, 5778.0, 5778.0, 19344.0, 1080.0,
                    15741.0, 2857.0, 0.0},
        {16067.0, 106300.0, -48525.0, 272400.0, -260550.0, 427368.0, -
260550.0,
                    272400.0, -48525.0, 106300.0, 16067.0}
    };

    for(int i=0; i<MAX; i++)
    {
        for(int j=0; j<MAX; j++)
        {
            z[i][j] = matrix_ncotes_Ci[i][j];
        }
    }
}

// print vectors
void print_vector(double *x)
{
    for(int i=0; i<MAX; i++)
    {
        printf("%f \t", *(x+i));
    }
```

```
}

// print matrices
void print_matrix(double **x)
{
    for(int i=0; i<MAX;i++)
    {
        printf("k = %d \n", i);

        for(int j=0; j<MAX; j++)
        {
            printf("%f \t", x[i][j]);
        }

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

[575 lines of code]

File: tinyexpr.h

Description: This file has the header to link the 3-rd library tinyexpr (library to interpreter string math functions and convert to mathematical functions, and compute the numerical value of this function given some argument) implemented in C.

```
// SPDX-License-Identifier: Zlib
/*
 * TINYEXPR - Tiny recursive descent parser and evaluation engine in C
 *
 * Copyright (c) 2015-2020 Lewis Van Winkle
 *
 * http://CodePlea.com
 *
 * This software is provided 'as-is', without any express or implied
 * warranty. In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 * claim that you wrote the original software. If you use this software
 * in a product, an acknowledgement in the product documentation would be
 * appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 * misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 */

#ifndef TINYEXPR_H
#define TINYEXPR_H


#ifdef __cplusplus
extern "C" {
#endif



typedef struct te_expr {
    int type;
    union {double value; const double *bound; const void *function;};
    void *parameters[1];
```

```
} te_expr;


enum {
    TE_VARIABLE = 0,

    TE_FUNCTION0 = 8, TE_FUNCTION1, TE_FUNCTION2, TE_FUNCTION3,
    TE_FUNCTION4, TE_FUNCTION5, TE_FUNCTION6, TE_FUNCTION7,

    TE_CLOSURE0 = 16, TE_CLOSURE1, TE_CLOSURE2, TE_CLOSURE3,
    TE_CLOSURE4, TE_CLOSURE5, TE_CLOSURE6, TE_CLOSURE7,

    TE_FLAG_PURE = 32
};

typedef struct te_variable {
    const char *name;
    const void *address;
    int type;
    void *context;
} te_variable;



/* Parses the input expression, evaluates it, and frees it. */
/* Returns NaN on error. */
double te_interp(const char *expression, int *error);

/* Parses the input expression and binds variables. */
/* Returns NULL on error. */
te_expr *te_compile(const char *expression, const te_variable *variables, int
var_count, int *error);

/* Evaluates the expression. */
double te_eval(const te_expr *n);

/* Prints debugging information on the syntax tree. */
void te_print(const te_expr *n);

/* Frees the expression. */
/* This is safe to call on NULL pointers. */
void te_free(te_expr *n);


#ifdef __cplusplus
}
#endif

#endif /*TINYEXPR_H*/
```

4. DEFINITION OF EXPERIMENTS

In this project, was considered 4 different experiments in increasing order of complexity:

1) Experiment 01: Study the influence of the quadrature parameter “k”
Formula:

$$I_{apx} = \int_0^{\pi} \sin(x)dx$$

Input data:
a = 0.000000 // this is the lower limit
b = 3.141593 // this is the upper limit
expr = sin(x) // this is the function to integrate
I = 2.000000 // this is the exact value of the integral

2) Experiment 02: Study the influence of the number of processes “q”
Formula:

$$I_{apx} = \int_1^2 \frac{\ln(x)}{x} dx$$

Input data:
a = 1
b = 2
expr = ln(x)/x
I = (ln(2)^2)/2

3) Experiment 03: Study the influence of the number of nodes “N”
Formula:

$$I_{apx} = \int_0^1 e^{-x^2} dx$$

Input data:

a = 0

b = 1

expr = exp(-(x^2))

I = sqrt(pi)/2 * erf(1)

4) Experiment 04: Study the influence of the programming language

Formula:

$$I_{apx} = \int_{0.75}^{1} \frac{\arctan(x)}{1+\frac{1}{x^2}} dx$$

Input data:

a = 0.75

b = 1

expr = math.atan(x)/(1+(1/x)**2)

I = 1/32 * (8*math.pi - (math.pi)**2 - 8 * (math.log(1024/625)) - 24 * math.atan(3/4) + 16 * (math.atan(3/4))**2)

5. EXPERIMENTAL RESULTS

1) Experiment 01: study **k**

Study the influence of quadrature (k) in performance of error and time. Here, we fix the number of processes in 4 (q = 4) and number of nodes in 1 (N = 1).

#processes = 4

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 4 ./newton_cotes_vc14

For q = 4, k = 1

OUTPUTS:

**** All data: ***

0.000000 0.785398 1.570796 2.356194 3.141593

****

Data for process 1:

0.785398 1.570796

****

Data for process 0:

0.000000 0.785398

****

Data for process 3:

2.356194 3.141593

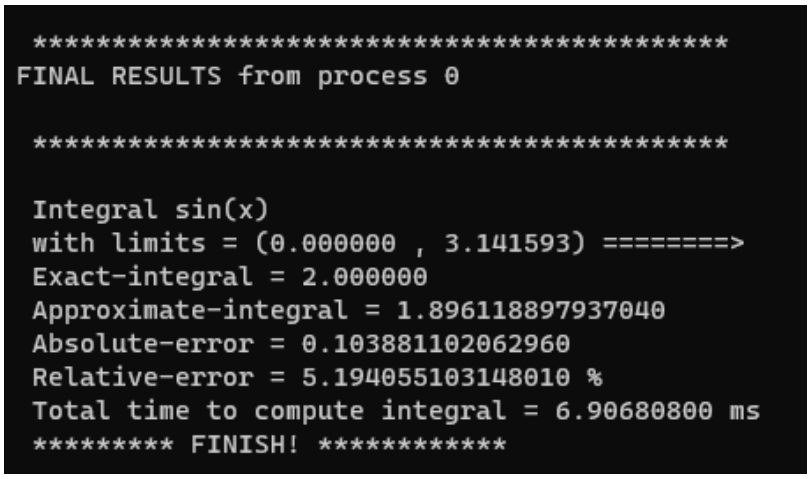
****

Data for process 2:

1.570796 2.356194

****

FINAL RESULTS:



****
FINAL RESULTS from process 0
****
Integral sin(x)
with limits = (0.000000 , 3.141593) ========>
Exact-integral = 2.000000
Approximate-integral = 1.896118897937040
Absolute-error = 0.103881102062960
Relative-error = 5.194055103148010 %
Total time to compute integral = 6.90680800 ms
********* FINISH! **********

For q = 4, k = 2

OUTPUTS:

**** All data: ***

0.000000 0.392699 0.785398 1.178097 1.570796 1.963495
2.356194 2.748894 3.141593

****

Data for process 0:

0.000000 0.392699 0.785398

****

Data for process 3:

2.356194 2.748894 3.141593

****

Data for process 1:

0.785398 1.178097 1.570796

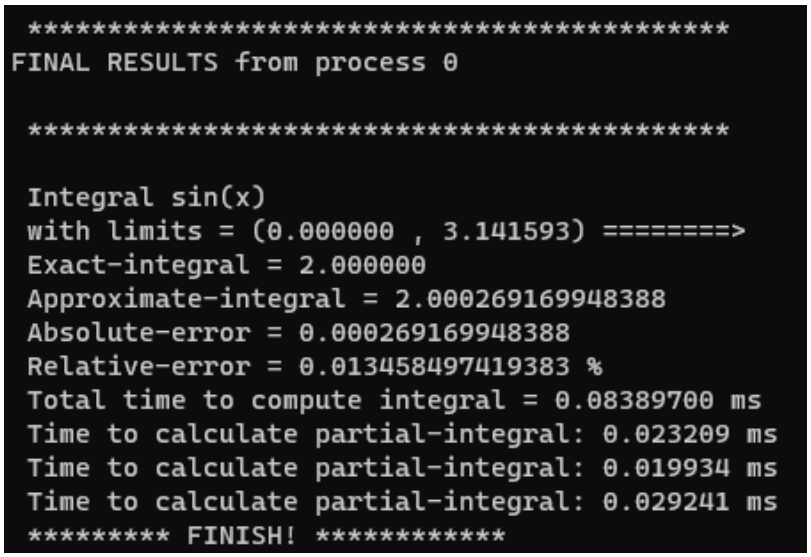
****

Data for process 2:

1.570796 1.963495 2.356194

****

FINAL RESULTS



****
FINAL RESULTS from process 0
****
Integral sin(x)
with limits = (0.000000, 3.141593) ========>
Exact-integral = 2.000000
Approximate-integral = 2.000269169948388
Absolute-error = 0.000269169948388
Relative-error = 0.013458497419383 %
Total time to compute integral = 0.08389700 ms
********* FINISH! **********

For q = 4, k = 3

OUTPUTS:

**** All data: ***

0.000000 0.261799 0.523599 0.785398 1.047198 1.308997
1.570796 1.832596 2.094395 2.356194 2.617994 2.879793 3.141593

****

Data for process 0:

0.000000 0.261799 0.523599 0.785398

****

Data for process 2:

1.570796 1.832596 2.094395 2.356194

****

Data for process 3:

2.356194 2.617994 2.879793 3.141593

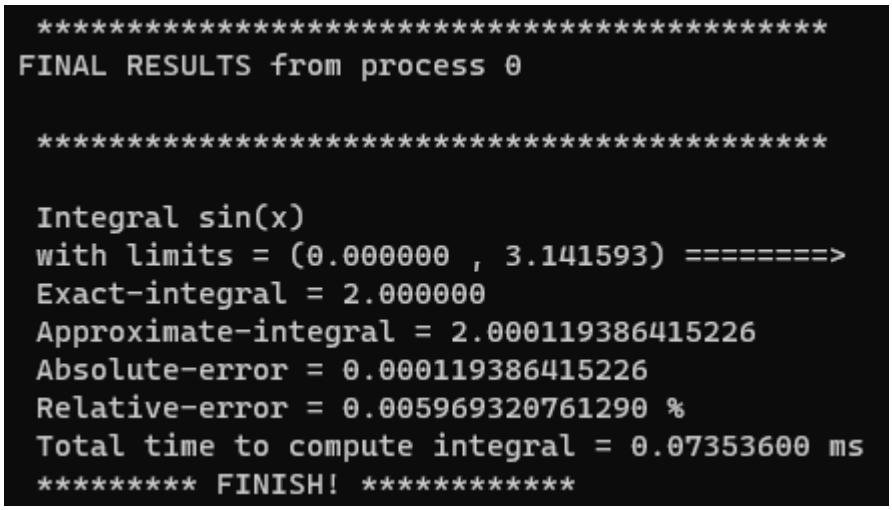
****

Data for process 1:

0.785398 1.047198 1.308997 1.570796

****

FINAL RESULTS



****

FINAL RESULTS from process 0

****

Integral sin(x)

with limits = (0.000000, 3.141593) ========>

Exact-integral = 2.000000

Approximate-integral = 2.000119386415226

Absolute-error = 0.000119386415226

Relative-error = 0.005969320761290 %

Total time to compute integral = 0.07353600 ms

********* FINISH! **********

For q = 4, k = 4

OUTPUTS:

*** All data: ***

0.000000 0.196350 0.392699 0.589049 0.785398 0.981748
1.178097 1.374447 1.570796 1.767146 1.963495 2.159845
2.356194 2.552544 2.748894 2.945243 3.141593

****

Data for process 1:

0.785398 0.981748 1.178097 1.374447 1.570796

****

Data for process 0:

0.000000 0.196350 0.392699 0.589049 0.785398

****

Data for process 2:

1.570796 1.767146 1.963495 2.159845 2.356194

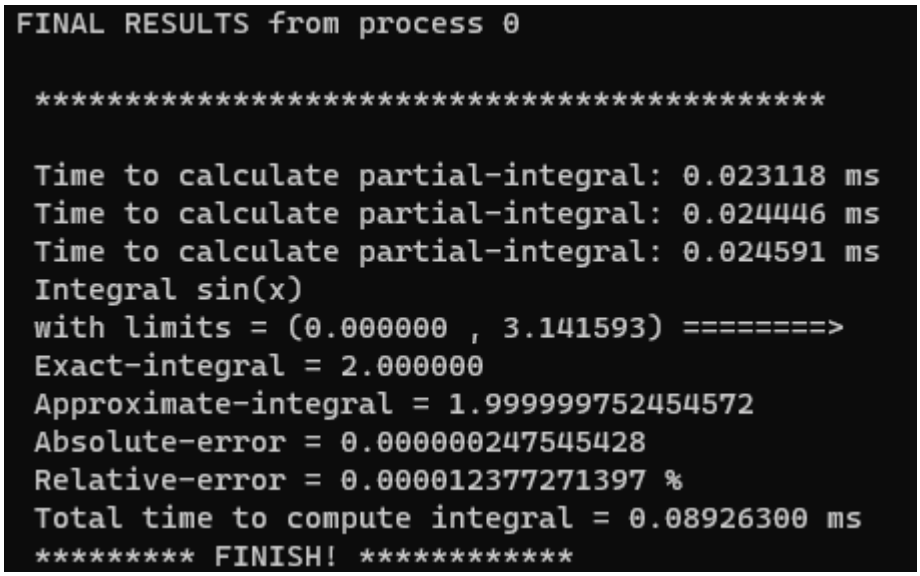
****

Data for process 3:

2.356194 2.552544 2.748894 2.945243 3.141593

****

FINAL RESULTS



****

FINAL RESULTS from process 0

****

Integral sin(x)

with limits = (0.000000, 3.141593) ========>

Exact-integral = 2.000000

Approximate-integral = 1.999999752454572

Absolute-error = 0.000000247545428

Relative-error = 0.000012377271397 %

Total time to compute integral = 0.08926300 ms

********* FINISH! **********

For q = 4, k = 5

OUTPUTS:

*** All data: ***

0.000000 0.157080 0.314159 0.471239 0.628319 0.785398
0.942478 1.099557 1.256637 1.413717 1.570796 1.727876 1.884956
2.042035 2.199115 2.356194 2.513274 2.670354 2.827433 2.984513
3.141593

****

Data for process 0:

0.000000 0.157080 0.314159 0.471239 0.628319 0.785398

****

Data for process 1:

0.785398 0.942478 1.099557 1.256637 1.413717 1.570796

****

Data for process 3:

2.356194 2.513274 2.670354 2.827433 2.984513 3.141593

****

Data for process 2:

1.570796 1.727876 1.884956 2.042035 2.199115 2.356194

****

FINAL RESULTS:

```
 ******************************************
FINAL RESULTS from process 0

 ******************************************

 Integral sin(x)
 with limits = (0.000000 , 3.141593) ========>
 Exact-integral = 2.000000
 Approximate-integral = 1.999999860668031
 Absolute-error = 0.000000139331969
 Relative-error = 0.000006966598454 %
 Total time to compute integral = 0.10419600 ms
 ********* FINISH! ************
```

****

FINAL RESULTS from process 0

****

Integral sin(x)

with limits = (0.000000, 3.141593) ========>

Exact-integral = 2.000000

Approximate-integral = 1.999999860668031

Absolute-error = 0.000000139331969

Relative-error = 0.000006966598454 %

Total time to compute integral = 0.10419600 ms

********* FINISH! **********

EXPERIMENT 01: ANALYSIS OF K

N = 1, q = 4			
Function: sin(x); limits: [0, pi]			
k (degree of quadrature)	Absolute error (abs_err)	Relative error (%)	Total time (ms)
1	0.103881102062960	5.194055103148010	6.90680800
2	0.000269169948388	0.013458497419383	0.08389700
3	0.000119386415226	0.005969320761290	0.07353600
4	0.000000247545428	0.000012377271397	0.08926300
5	0.000000139331969	0.000006966598454	0.10419600

2) Experiment 02: study **q**

Study the influence of number of processes (q) in performance of error and time. Here, we fix degree of quadrature in 3 (k = 3) and number of nodes in 1 (N = 1).

#processes = 4

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 4 ./newton_cotes_vc14

File: exp2_c/exp2c_k3.txt

For q = 4, k = 3

OUTPUTS:

**** All data: ***

1.000000 1.083333 1.166667 1.250000 1.333333 1.416667
1.500000 1.583333 1.666667 1.750000 1.833333 1.916667 2.000000

****

Data for process 0:

1.000000 1.083333 1.166667 1.250000

****

Data for process 2:

1.500000 1.583333 1.666667 1.750000

****

Data for process 1:

1.250000 1.333333 1.416667 1.500000

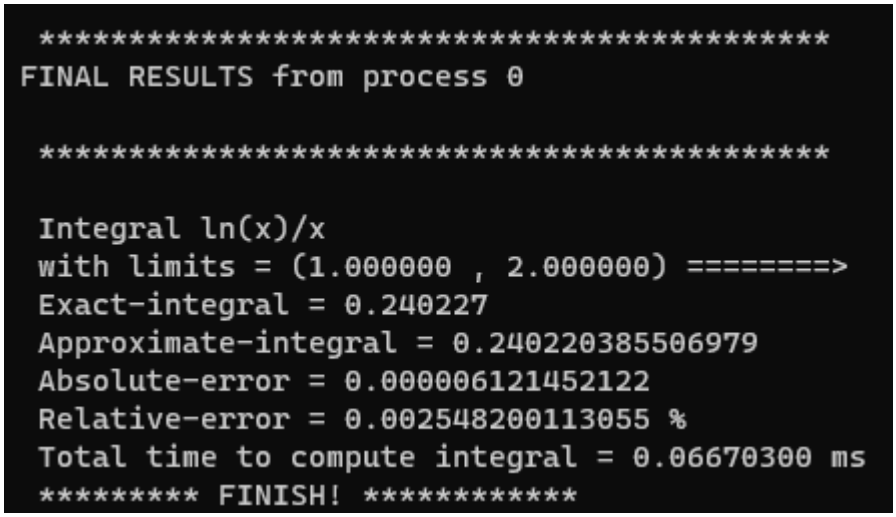
****

Data for process 3:

1.750000 1.833333 1.916667 2.000000

****

FINAL RESULTS:



****
FINAL RESULTS from process 0
****
Integral ln(x)/x
with limits = (1.000000, 2.000000) ========>
Exact-integral = 0.240227
Approximate-integral = 0.240220385506979
Absolute-error = 0.000006121452122
Relative-error = 0.002548200113055 %
Total time to compute integral = 0.06670300 ms
********* FINISH! **********

#processes = 8

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 8 ./newton_cotes_vc14

File: exp2_c/exp2c_k3.txt

For q = 8, k = 3

OUTPUTS:

**** All data: ***

1.000000 1.041667 1.083333 1.125000 1.166667 1.208333
1.250000 1.291667 1.333333 1.375000 1.416667 1.458333 1.500000
1.541667 1.583333 1.625000 1.666667 1.708333 1.750000 1.791667
1.833333 1.875000 1.916667 1.958333 2.000000

****

Data for process 0:

1.000000 1.041667 1.083333 1.125000

****

Data for process 6:

1.750000 1.791667 1.833333 1.875000

****

Data for process 1:

1.125000 1.166667 1.208333 1.250000

****

Data for process 7:

1.875000 1.916667 1.958333 2.000000

****

Data for process 2:

1.250000 1.291667 1.333333 1.375000

****

Data for process 3:

1.375000 1.416667 1.458333 1.500000

****

Data for process 4:

1.500000 1.541667 1.583333 1.625000

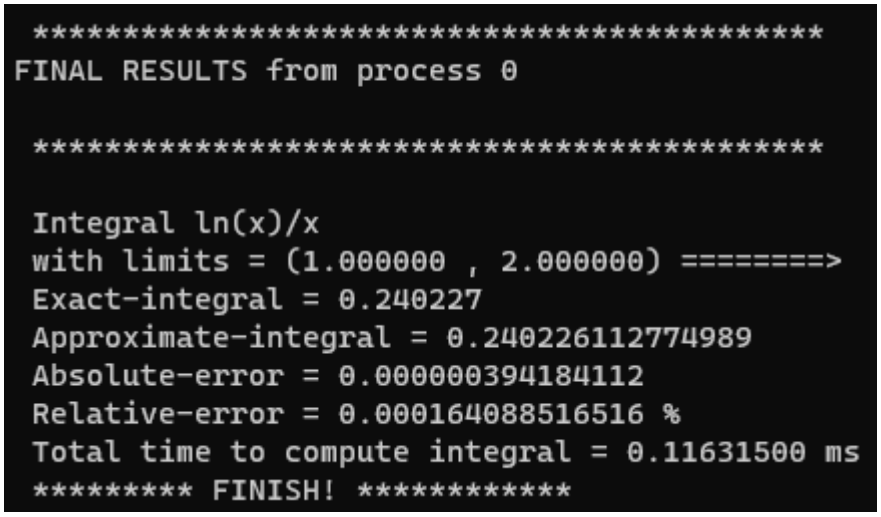
****

Data for process 5:

1.625000 1.666667 1.708333 1.750000

****

FINAL RESULTS:



****

FINAL RESULTS from process 0

****

Integral ln(x)/x

with limits = (1.000000, 2.000000) ========>

Exact-integral = 0.240227

Approximate-integral = 0.240226112774989

Absolute-error = 0.000000394184112

Relative-error = 0.000164088516516 %

Total time to compute integral = 0.11631500 ms

********* FINISH! **********

#processes = 12

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 12 ./newton_cotes_vc14

File: exp2_c/exp2c_k3.txt

For q = 12, k = 3

OUTPUTS:

**** All data: ***

1.000000 1.027778 1.055556 1.083333 1.111111 1.138889
1.166667 1.194444 1.222222 1.250000 1.277778 1.305556 1.333333
1.361111 1.388889 1.416667 1.444444 1.472222 1.500000 1.527778
1.555556 1.583333 1.611111 1.638889 1.666667 1.694444
1.722222 1.750000 1.777778 1.805556 1.833333 1.861111 1.888889
1.916667 1.944444 1.972222 2.000000

****

Data for process 10:

1.833333 1.861111 1.888889 1.916667

****

Data for process 7:

1.583333 1.611111 1.638889 1.666667

****

Data for process 8:

1.666667 1.694444 1.722222 1.750000

****

Data for process 11:

1.916667 1.944444 1.972222 2.000000

****

Data for process 0:

1.000000 1.027778 1.055556 1.083333

****

Data for process 9:

1.750000 1.777778 1.805556 1.833333

****

Data for process 2:

1.166667 1.194444 1.222222 1.250000

****

Data for process 1:

1.083333 1.111111 1.138889 1.166667

****

Data for process 4:

1.333333 1.361111 1.388889 1.416667

****

Data for process 3:

1.250000 1.277778 1.305556 1.333333

****

Data for process 6:

1.500000 1.527778 1.555556 1.583333

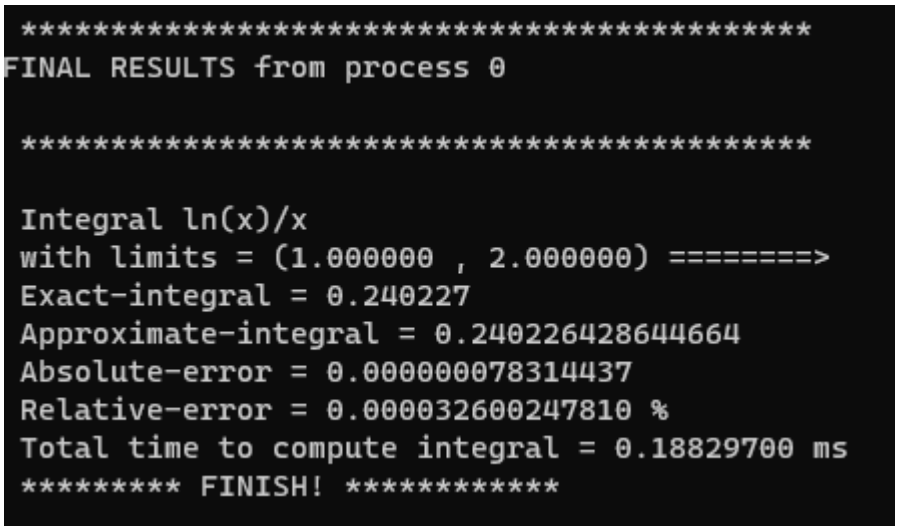
****

Data for process 5:

1.416667 1.444444 1.472222 1.500000

****

FINAL RESULTS:



****
FINAL RESULTS from process 0
****
Integral ln(x)/x
with limits = (1.000000, 2.000000) ========>
Exact-integral = 0.240227
Approximate-integral = 0.240226428644664
Absolute-error = 0.000000078314437
Relative-error = 0.000032600247810 %
Total time to compute integral = 0.18829700 ms
********* FINISH! **********

#processes = 16

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 16 ./newton_cotes_vc14

File: exp2_c/exp2c_k3.txt

For q = 16, k = 3

OUTPUTS:

**** All data: ***

1.000000 1.020833 1.041667 1.062500 1.083333 1.104167
1.125000 1.145833 1.166667 1.187500 1.208333 1.229167 1.250000
1.270833 1.291667 1.312500 1.333333 1.354167 1.375000 1.395833
1.416667 1.437500 1.458333 1.479167 1.500000 1.520833
1.541667 1.562500 1.583333 1.604167 1.625000 1.645833 1.666667
1.687500 1.708333 1.729167 1.750000 1.770833 1.791667 1.812500
1.833333 1.854167 1.875000 1.895833 1.916667 1.937500
1.958333 1.979167 2.000000

****

Data for process 0:

1.000000 1.020833 1.041667 1.062500

****

Data for process 15:

1.937500 1.958333 1.979167 2.000000

****

Data for process 2:

1.125000 1.145833 1.166667 1.187500

****

Data for process 3:

1.187500 1.208333 1.229167 1.250000

****

Data for process 1:

1.062500 1.083333 1.104167 1.125000

****

Data for process 4:

1.250000 1.270833 1.291667 1.312500

****

Data for process 5:

1.312500 1.333333 1.354167 1.375000

****

Data for process 7:

1.437500 1.458333 1.479167 1.500000

****

Data for process 6:

1.375000 1.395833 1.416667 1.437500

****

Data for process 8:

1.500000 1.520833 1.541667 1.562500

****

Data for process 9:

1.562500 1.583333 1.604167 1.625000

****

Data for process 11:

1.687500 1.708333 1.729167 1.750000

****

Data for process 12:

1.750000 1.770833 1.791667 1.812500

****

Data for process 13:

1.812500 1.833333 1.854167 1.875000

****

Data for process 14:

1.875000 1.895833 1.916667 1.937500

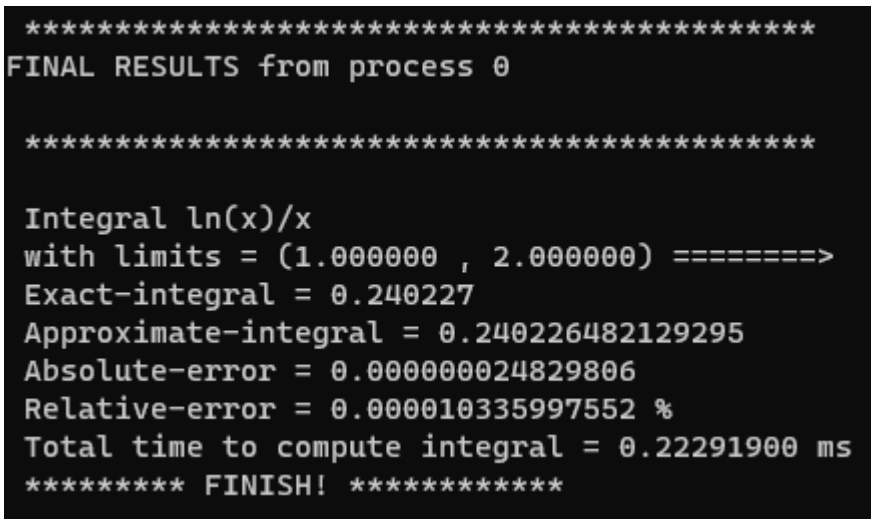
****

Data for process 10:

1.625000 1.645833 1.666667 1.687500

****

FINAL RESULTS:



****

FINAL RESULTS from process 0

****

Integral ln(x)/x

with limits = (1.000000, 2.000000) ========>

Exact-integral = 0.240227

Approximate-integral = 0.240226482129295

Absolute-error = 0.000000024829806

Relative-error = 0.000010335997552 %

Total time to compute integral = 0.22291900 ms

********* FINISH! **********

#processes = 20

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 20 ./newton_cotes_vc14

File: exp2_c/exp2c_k3.txt

For q = 20, k = 3

OUTPUTS:

**** All data: ***

1.000000 1.016667 1.033333 1.050000 1.066667 1.083333
1.100000 1.116667 1.133333 1.150000 1.166667 1.183333 1.200000
1.216667 1.233333 1.250000 1.266667 1.283333 1.300000 1.316667
1.333333 1.350000 1.366667 1.383333 1.400000 1.416667
1.433333 1.450000 1.466667 1.483333 1.500000 1.516667 1.533333
1.550000 1.566667 1.583333 1.600000 1.616667 1.633333 1.650000
1.666667 1.683333 1.700000 1.716667 1.733333 1.750000
1.766667 1.783333 1.800000 1.816667 1.833333 1.850000 1.866667
1.883333 1.900000 1.916667 1.933333 1.950000 1.966667 1.983333
2.000000

****

Data for process 0:

1.000000 1.016667 1.033333 1.050000

****

Data for process 1:

1.050000 1.066667 1.083333 1.100000

****

Data for process 2:

1.100000 1.116667 1.133333 1.150000

****

Data for process 3:

1.150000 1.166667 1.183333 1.200000

****

Data for process 4:

1.200000 1.216667 1.233333 1.250000

****

Data for process 5:

1.250000 1.266667 1.283333 1.300000

****

Data for process 6:

1.300000 1.316667 1.333333 1.350000

****

Data for process 19:

1.950000 1.966667 1.983333 2.000000

****

Data for process 7:

1.350000 1.366667 1.383333 1.400000

****

Data for process 8:

1.400000 1.416667 1.433333 1.450000

****

Data for process 9:

1.450000 1.466667 1.483333 1.500000

****

Data for process 10:

1.500000 1.516667 1.533333 1.550000

****

Data for process 11:

1.550000 1.566667 1.583333 1.600000

****

Data for process 12:

1.600000 1.616667 1.633333 1.650000

****

Data for process 13:

1.650000 1.666667 1.683333 1.700000

****

Data for process 14:

1.700000 1.716667 1.733333 1.750000

****

Data for process 15:

1.750000 1.766667 1.783333 1.800000

****

Data for process 16:

1.800000 1.816667 1.833333 1.850000

****

Data for process 17:

1.850000 1.866667 1.883333 1.900000

****

Data for process 18:

1.900000 1.916667 1.933333 1.950000

****

FINAL RESULTS:

```
 ****************************************
FINAL RESULTS from process 0

 ****************************************

 Integral ln(x)/x
 with limits = (1.000000 , 2.000000) ========>
 Exact-integral = 0.240227
 Approximate-integral = 0.240226496779163
 Absolute-error = 0.000000010179938
 Relative-error = 0.0000042376414 73 %
 Total time to compute integral = 0.30391400 ms
 ********* FINISH! ************
```

```
*******************************************
FINAL RESULTS from process 0
*******************************************
 Integral ln(x)/x
 with limits = (1.000000 , 2.000000) ========>
 Exact-integral = 0.240227
 Approximate-integral = 0.240226496779163
 Absolute-error = 0.000000010179938
 Relative-error = 0.000004237641473 %
 Total time to compute integral = 0.30391400 ms
********* FINISH! ************
```

ANALYSIS OF Q

N = 1, k = 3			
Function: ln(x)/x; limits: [1, 2]			
q (number of processes)	Absolute error (abs_err)	Relative error (%)	Total time (ms)
4	0.000006121452122	0.002548200113055	0.06670300
8	0.000000394184112	0.000164088516516	0.11631500
12	0.000000078314437	0.000032600247810	0.18829700
16	0.000000024829806	0.000010335997552	0.22291900
20	0.000000010179938	0.000004237641473	0.30391400

3) Experiment 03: study **N**

Study the influence of number of nodes (N) in performance of error and time. Here, we fix degree of quadrature in 5 (k = 5) and number of processes in 3 (q = 3).

#processes = 3

tm5u6@login1:~/project_C

$ salloc -N 1 -p cascade

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 3 ./newton_cotes_vc14

File: exp3_c/exp3c_k5.txt

For N = 1, q = 3, k = 5

OUTPUTS:

**** All data: ***

0.000000 0.066667 0.133333 0.200000 0.266667 0.333333
0.400000 0.466667 0.533333 0.600000 0.666667 0.733333 0.800000
0.866667 0.933333 1.000000

****

Data for process 0:

0.000000 0.066667 0.133333 0.200000 0.266667 0.333333

****

Data for process 1:

0.333333 0.400000 0.466667 0.533333 0.600000 0.666667

****

Data for process 2:

0.666667 0.733333 0.800000 0.866667 0.933333 1.000000

****

FINAL RESULTS:

```
 *********************************************
FINAL RESULTS from process 0

 *********************************************

 Integral exp(-(x^2))
```

```
 with limits = (0.000000 , 1.000000) ========>
 Exact-integral = 0.746824
 Approximate-integral = 0.746824134245647
 Absolute-error = 0.000000001433220
 Time to calculate partial-integral: 0.667042 ms
 Time to calculate partial-integral: 0.671013 ms
 Relative-error = 0.000000191908678 %
 Total time to compute integral = 2.04927500 ms
 ********* FINISH! ************
```

****

FINAL RESULTS from process 0

****

Integral exp(-(x^2))

with limits = (0.000000 , 1.000000) ========>

Exact-integral = 0.746824

Approximate-integral = 0.746824134245647

Absolute-error = 0.000000001433220

Time to calculate partial-integral: 0.667042 ms

Time to calculate partial-integral: 0.671013 ms

Relative-error = 0.000000191908678 %

Total time to compute integral = 2.04927500 ms

********* FINISH! **********

For N = 2, q = 3, k = 5

#processes = 3

tm5u6@login1:~/project_C

$ salloc -N 2 -p cascade

```
tm5u6@login1:~/project_C
$ salloc -N 2 -p cascade
salloc: Pending job allocation 2657639
salloc: job 2657639 queued and waiting for resources
salloc: job 2657639 has been allocated resources
salloc: Granted job allocation 2657639
tm5u6@login1:~/project_C
$
tm5u6@login1:~/project_C
$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2657639   cascade       sh    tm5u6  R       0:02      2 n06p[001-002]
```

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 3 ./newton_cotes_vc14

File: exp3_c/exp3c_k5.txt

OUTPUTS:

4) Experiment 04: study **language**

Study the influence of number of programming language (C or Python) in performance of error and time. Here, we fix number of processes (q = 5) and number of nodes in 1 (N = 1).

#processes = 5

tm5u6@login1:~/project_C

$ salloc -N 1 -p cascade

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 5 ./newton_cotes_vc14

File: exp4_c/exp4c_k4.txt

For N = 1, q = 5, k = 4 (Language C)

OUTPUTS:

**** All data: ***

0.750000 0.762500 0.775000 0.787500 0.800000 0.812500
0.825000 0.837500 0.850000 0.862500 0.875000 0.887500 0.900000
0.912500 0.925000 0.937500 0.950000 0.962500 0.975000 0.987500
1.000000

****

Data for process 0:

0.750000 0.762500 0.775000 0.787500 0.800000

****

Data for process 2:

0.850000 0.862500 0.875000 0.887500 0.900000

****

Data for process 1:

0.800000 0.812500 0.825000 0.837500 0.850000

****

Data for process 4:

0.950000 0.962500 0.975000 0.987500 1.000000

****

Data for process 3:

0.900000 0.912500 0.925000 0.937500 0.950000

****

FINAL RESULTS:

```
 ****************************************
FINAL RESULTS from process 0

 ****************************************

 Integral atan(x)/(1+(1/x)^2)
 with limits = (0.750000 , 1.000000) ========>
 Exact-integral = 0.077964
 Approximate-integral = 0.077963993811546
 Absolute-error = 0.000000000000226
 Relative-error = 0.000000000290518 %
 Total time to compute integral = 0.78074400 ms
 ********* FINISH! ************
```

****

FINAL RESULTS from process 0

****

Integral atan(x)/(1+(1/x)^2)

with limits = (0.750000, 1.000000) ========>

Exact-integral = 0.077964

Approximate-integral = 0.077963993811546

Absolute-error = 0.000000000000226

Relative-error = 0.000000000290518 %

Total time to compute integral = 0.78074400 ms

********* FINISH! **********

For N = 1, q = 5, k = 4 (Language Python)

#processes = 5

(ex) [tm5u6@n06p001 labpy]$ mpirun -np 5 python3 newton_cotes_final1.py

File: exp4/exp4_k4.txt

OUTPUTS:

0 process ======> [0.75 0.7625 0.775 0.7875 0.8]

4 process ======> [0.95 0.9625 0.975 0.9875 1.]

3 process ======> [0.9 0.9125 0.925 0.9375 0.95]

2 process ======> [0.85 0.8625 0.875 0.8875 0.9]

1 process ======> [0.8 0.8125 0.825 0.8375 0.85]

FINAL RESULTS:

```
***** FINAL RESULTS ***** from process  0
Integral  math.atan(x)/(1+(1/x)**2)  with limits = (  0.75 ,  1.0 ) ===>
Exact-integral =  0.07796399381177285
Approximate-integral =  0.07796399381154635
Absolute-error =  2.26499374811339 75e-13
Relative-error =  2.905179221041105e-10 %
Total-time to compute aproximate integral =  1.636413  ms
Finish!
```

***** FINAL RESULTS ***** from process 0

Integral math.atan(x)/(1+(1/x)**2) with limits = (0.75 , 1.0) ===>

Exact-integral = 0.07796399381177285

Approximate-integral = 0.07796399381154635

Absolute-error = 2.2649937481133975e-13

Relative-error = 2.905179221041105e-10 %

Total-time to compute aproximate integral = 1.636413 ms

Finish!

For N = 1, q = 5, k = 6 (Language C)

#processes = 5

tm5u6@login1:~/project_C

$ salloc -N 1 -p cascade

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 5 ./newton_cotes_vc14

File: exp4_c/exp4c_k6.txt

OUTPUTS:

**** All data: ***

0.750000 0.758333 0.766667 0.775000 0.783333 0.791667
0.800000 0.808333 0.816667 0.825000 0.833333 0.841667 0.850000
0.858333 0.866667 0.875000 0.883333 0.891667 0.900000 0.908333
0.916667 0.925000 0.933333 0.941667 0.950000 0.958333
0.966667 0.975000 0.983333 0.991667 1.000000

****

Data for process 2:

0.850000 0.858333 0.866667 0.875000 0.883333 0.891667
0.900000

****

Data for process 4:

0.950000 0.958333 0.966667 0.975000 0.983333 0.991667
1.000000

****

Data for process 0:

0.750000 0.758333 0.766667 0.775000 0.783333 0.791667
0.800000

****

Data for process 3:

0.900000 0.908333 0.916667 0.925000 0.933333 0.941667 0.950000

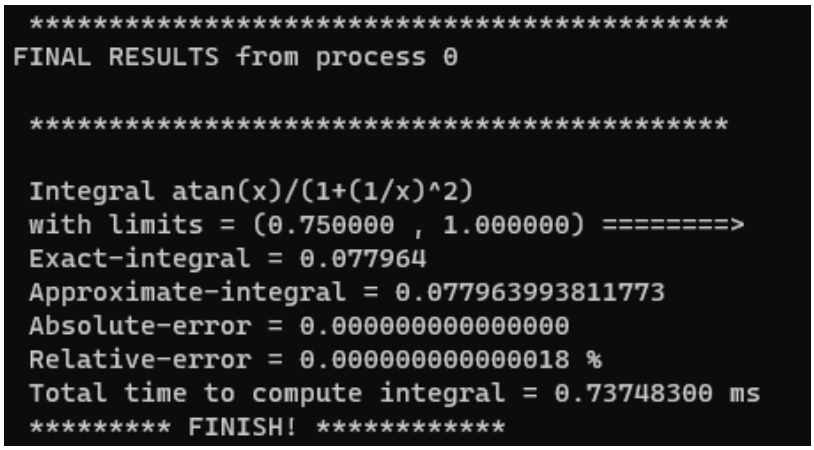
****

Data for process 1:

0.800000 0.808333 0.816667 0.825000 0.833333 0.841667 0.850000

****

FINAL RESULTS:



****

FINAL RESULTS from process 0

****

Integral atan(x)/(1+(1/x)^2)

with limits = (0.750000 , 1.000000) ========>

Exact-integral = 0.077964

Approximate-integral = 0.077963993811773

Absolute-error = 0.000000000000000

Relative-error = 0.000000000000018 %

Total time to compute integral = 0.73748300 ms

********* FINISH! **********

For N = 1, q = 5, k = 6 (Language Python)

#processes = 5

(ex) [tm5u6@n06p001 labpy]$ mpirun -np 5 python3 newton_cotes_final1.py

File: exp4/exp4_k6.txt

OUTPUTS:

0 process ======> [0.75 0.75833333 0.76666667 0.775 0.78333333 0.79166667 0.8]

4 process ======> [0.95 0.95833333 0.96666667 0.975 0.98333333 0.99166667 1.]

2 process ======> [0.85 0.85833333 0.86666667 0.875 0.88333333 0.89166667 0.9]

3 process ======> [0.9 0.90833333 0.91666667 0.925 0.93333333 0.94166667 0.95]

1 process ======> [0.8 0.80833333 0.81666667 0.825 0.83333333 0.84166667 0.85]

FINAL RESULTS:

```
***** FINAL RESULTS ***** from process  0
Integral  math.atan(x)/(1+(1/x)**2)  with limits = (  0.75 ,  1.0 ) ===>
Exact-integral =  0.07796399381177285
Approximate-integral =  0.07796399381177287
Absolute-error =  1.38777878078144567e-17
Relative-error =  1.7800252564432968e-14 %
Total-time to compute aproximate integral =  1.9301719999999998  ms
Finish!
```

***** FINAL RESULTS ***** from process 0

Integral math.atan(x)/(1+(1/x)**2) with limits = (0.75 , 1.0) ===>

Exact-integral = 0.07796399381177285

Approximate-integral = 0.07796399381177287

Absolute-error = 1.3877787807814457e-17

Relative-error = 1.7800252564432968e-14 %

Total-time to compute aproximate integral = 1.9301719999999998 ms

Finish!

For N = 1, q = 5, k = 8 (Language C)

#processes = 5

tm5u6@login1:~/project_C

$ salloc -N 1 -p cascade

[tm5u6@n06p001 project_C]$ mpicc newton_cotes_vc14.c tinyexpr.c -o newton_cotes_vc14 -l m

[tm5u6@n06p001 project_C]$ mpirun -np 8 ./newton_cotes_vc14

File: exp4_c/exp4c_k8.txt

OUTPUTS:

**** All data: ***

0.750000 0.753906 0.757812 0.761719 0.765625 0.769531
0.773438 0.777344 0.781250 0.785156 0.789062 0.792969 0.796875
0.800781 0.804688 0.808594 0.812500 0.816406 0.820312 0.824219
0.828125 0.832031 0.835938 0.839844 0.843750 0.847656
0.851562 0.855469 0.859375 0.863281 0.867188 0.871094 0.875000
0.878906 0.882812 0.886719 0.890625 0.894531 0.898438 0.902344
0.906250 0.910156 0.914062 0.917969 0.921875 0.925781
0.929688 0.933594 0.937500 0.941406 0.945312 0.949219 0.953125
0.957031 0.960938 0.964844 0.968750 0.972656 0.976562 0.980469
0.984375 0.988281 0.992188 0.996094 1.000000

****

Data for process 0:

0.750000 0.753906 0.757812 0.761719 0.765625 0.769531
0.773438 0.777344 0.781250

****

Data for process 6:

0.937500 0.941406 0.945312 0.949219 0.953125 0.957031
0.960938 0.964844 0.968750

****

Data for process 7:

0.968750 0.972656 0.976562 0.980469 0.984375 0.988281
0.992188 0.996094 1.000000

****

Data for process 1:

0.781250 0.785156 0.789062 0.792969 0.796875 0.800781
0.804688 0.808594 0.812500

****

Data for process 2:

0.812500 0.816406 0.820312 0.824219 0.828125 0.832031
0.835938 0.839844 0.843750

****

Data for process 3:

0.843750 0.847656 0.851562 0.855469 0.859375 0.863281
0.867188 0.871094 0.875000

****

Data for process 4:

0.875000 0.878906 0.882812 0.886719 0.890625 0.894531
0.898438 0.902344 0.906250

****

Data for process 5:

0.906250 0.910156 0.914062 0.917969 0.921875 0.925781
0.929688 0.933594 0.937500

****

FINAL RESULTS:

```
 ******************************************
FINAL RESULTS from process 0

 ******************************************

 Integral atan(x)/(1+(1/x)^2)
 with limits = (0.750000 , 1.000000) ========>
 Exact-integral = 0.077964
 Approximate-integral = 0.077963993811773
 Absolute-error = 0.000000000000000
 Relative-error = 0.000000000000000 %
 Total time to compute integral = 0.26206300 ms
 ********* FINISH! ************
```

```
*****************************************
FINAL RESULTS from process 0
 *****************************************
 Integral atan(x)/(1+(1/x)^2)
 with limits = (0.750000, 1.000000) ========>
 Exact-integral = 0.077964
 Approximate-integral = 0.077963993811773
 Absolute-error = 0.000000000000000
 Relative-error = 0.000000000000000 %
 Total time to compute integral = 0.26206300 ms
 ********* FINISH! ************
```

For N = 1, q = 5, k = 8 (Language Python)

#processes = 5

(ex) [tm5u6@n06p001 labpy]$ mpirun -np 5 python3 newton_cotes_final1.py

File: exp4/exp4_k8.txt

OUTPUTS:

0 process ======> [0.75 0.75625 0.7625 0.76875 0.775 0.78125 0.7875 0.79375 0.8]

4 process ======> [0.95 0.95625 0.9625 0.96875 0.975 0.98125 0.9875 0.99375 1.]

3 process ======> [0.9 0.90625 0.9125 0.91875 0.925 0.93125 0.9375 0.94375 0.95]

2 process ======> [0.85 0.85625 0.8625 0.86875 0.875 0.88125 0.8875 0.89375 0.9]

1 process ======> [0.8 0.80625 0.8125 0.81875 0.825 0.83125 0.8375 0.84375 0.85]

FINAL RESULTS:

```
***** FINAL RESULTS ***** from process  0
Integral  math.atan(x)/(1+(1/x)**2)  with limits = (  0.75 ,  1.0 ) ===>
Exact-integral =  0.07796399381177285
Approximate-integral =  0.07796399381177288
Absolute-error =  2.7755575615628914e-17
Relative-error =  3.5600505128865937e-14 %
Total-time to compute aproximate integral =  2.3629139999999995  ms
Finish!
```

***** FINAL RESULTS ***** from process 0

Integral math.atan(x)/(1+(1/x)**2) with limits = (0.75 , 1.0) ===>

Exact-integral = 0.07796399381177285

Approximate-integral = 0.07796399381177288

Absolute-error = 2.7755575615628914e-17

Relative-error = 3.5600505128865937e-14 %

Total-time to compute aproximate integral = 2.3629139999999995 ms

Finish!

ANALYSIS OF LANGUAGE

N = 1, q = 5			
Function: math.atan(x)/(1+(1/x)**2); limits: [0.75, 1]			
LANGUAGE C			
k (degree of quadrature)	Absolute error (abs_err)	Relative error (%)	Total time (ms)
4	000000000000226	0.000000000290518	0.78074400
6	0.000000000000000	0.000000000000018	0.73748300
8	0.000000000000000	0.000000000000000	0.26206300
LANGUAGE PYTHON			
4	2.2649937481133975e-13	2.905179221041105e-10	1.636413
6	1.3877787807814457e-17	1.7800252564432968e-14	1.9301719999999998
8	2.7755575615628914e-17	3.5600505128865937e-14	2.3629139999999995

5. CONCLUSION

In this research work, was concluded in the next aspects:

- The increasing of parameter k, improves the performance of absolute error and the relative error to calculate the approximate integral. In other words, as we use a higher degree quadrature, better is the approximation of the numerical integral. Use a higher degree of quadrature affects the time. The higher the degree of quadrature, the faster the algorithm is in the calculation process and therefore the shorter the time.
- The increasing of parameter q, improves the performance of absolute error and relative error, but take a more time to make calculations. In other words, as we use a much more processes/threads the performance of calculations increasing considerably, but the algorithm expends more time to achieve the results.
- The programming language C experimentally demonstrates experimentally it was shown to reach comparable levels of performance with respect to the Python language in less time.
- The influence of problem's complexity related with the quadrature's degree, has opposite behaviors between C and Python. Whereas in C, increasing the degree of quadrature complexity makes the algorithm perform faster, in the case of Python, increasing the degree of quadrature makes the algorithm run slower and even with a higher degree of quadrature. quasi-linear increasing trend with respect to time.