МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ имени М.В.ЛОМОНОСОВА

**Практикум по курсу**

**"Распределенные системы"**

**Разработка параллельной версии программы для решения определенного интеграла методом Симпсона MPI**

**ОТЧЕТ**

**о выполненном задании**

Выполнил студент ПМ401 группы

Хаметов Марк Владиминович

edu-cmc-skpod22-313-7

Севастополь, 2024 г.

**Оглавление**

[1 Постановка задачи](#_heading=h.gjdgxs) **2**

[2 Описание](#_heading=h.30j0zll) **3**

[2.1 Основа: последовательный алгоритм](#_heading=h.1fob9te) 3

[2.2 Параллельный алгоритм](#_heading=h.3znysh7) 4

[3 Результаты замеров времени выполнения](#_heading=h.tyjcwt) **5**

[3.1 Таблица](#_heading=h.3dy6vkm) 5

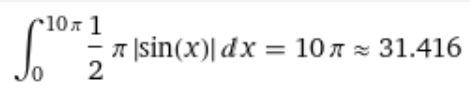
[3.2 График](#_heading=h.1t3h5sf) 6

[4 Анализ результатов](#_heading=h.4d34og8) **7**

[5 Выводы](#_heading=h.2s8eyo1) **7**

# Постановка задачи

Ставится задача нахождения значения интеграла на интервале.

Дана функция π**/**2 \* |sin X| , дан интервал [A;B] = [0; 10π ], требуется получить значение  .

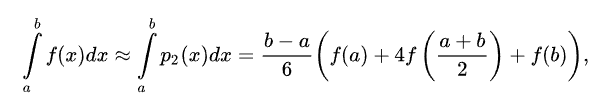
Требуется:

1. Реализовать параллельный алгоритм с помощью библиотеки параллельного программирования MPI.
2. Исследовать масштабируемость программы построив график времени выполнения программы от числа используемых потоков и при разном количестве интервалов разбиения функции.

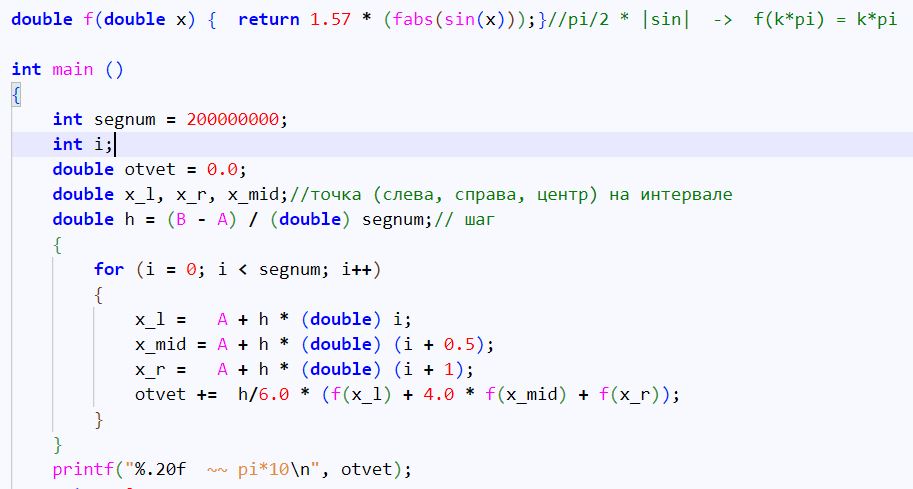
# Описание

## Основа: последовательный алгоритм

По следующей формуле была написана программа:



Алгоритм имеет вид:



Этот алгоритм имеет сложность O(nm), где m сложность функции, а n сложность метода Симпсона.

## Параллельный алгоритм

#include <mpi.h>

#include <math.h>

#include <stdio.h>

#include <time.h>

#define **A** 0.0

#define **B** 31.41592653589793 // pi\*10

double **f**(double x) { return 1.57 \* (**fabs**(**sin**(x))); } // pi/2 \* |sin| -> f(k\*pi) = k\*pi

int **main**(int argc, char\*\* argv) {

**MPI\_Init**(**NULL**, **NULL**);

    int world\_size;

**MPI\_Comm\_size**(MPI\_COMM\_WORLD, &world\_size);

    int world\_rank;

**MPI\_Comm\_rank**(MPI\_COMM\_WORLD, &world\_rank);

    int segnum = 200000000;

    int local\_segnum = segnum / world\_size;

    double otvet = 0.0;

    double x\_l, x\_r, x\_mid; // точка (слева, справа, центр) на интервале

    double h = (**B** - **A**) / (double) segnum; // шаг

    clock\_t start = **clock**(); // Start timing

    for (int i = world\_rank \* local\_segnum; i < (world\_rank + 1) \* local\_segnum; i++) {

        x\_l = **A** + h \* (double) i;

        x\_mid = **A** + h \* (double) (i + 0.5);

        x\_r = **A** + h \* (double) (i + 1);

        otvet += h / 6.0 \* (**f**(x\_l) + 4.0 \* **f**(x\_mid) + **f**(x\_r));

    }

    double total\_otvet = 0.0;

**MPI\_Reduce**(&otvet, &total\_otvet, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    if (world\_rank == 0) {

        clock\_t end = **clock**(); // End timing

        double time\_used = ((double) (end - start)) / **CLOCKS\_PER\_SEC**;

**printf**("Time used: %.6f seconds\n", time\_used);

**printf**("Number of processes: %d\n", world\_size);

**printf**("%.20f ~~ pi\*10\n", total\_otvet);

    }

**MPI\_Finalize**();

    return 0;

}

Пример использования для 4 процессов

/baxtin/bin/mpicc /baxtin/simpmpi.c -o simpmpi -lm

/baxtin/bin/mpirun -np 4 --oversubscribe ./simpmpi

**MPI\_Init(NULL, NULL);**

Используется для инициализации среды библиотеки

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &****world\_size);**

Используется для получения количества процессов в группе. Значение записывается в параметр world\_size. Количество процессов при вызове программы задавалось флагом -np.

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);**

Используется для записи в world\_rank текущего ранга процесса в группе. Т.е получаем номер процесса. Нулевой номер – номер единственного процесса, а остальные номера получаются с считая от нуля 0,1,2,..,максимум.

**MPI\_Reduce(&otvet, &total\_otvet, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);**

Используется для приема результатов **otvet** всех процессов и применения к ним **MPI\_SUM (или другой)** функции для получения суммы **total\_otvet,** все процессы обьединяются в кореной процесс Root. **MPI\_DOUBLE** указывает тип данных результатов процессов. MPI\_Count = 1 количество элементов в каждом **otvet.** Root = 0 номер процесса к которому мы сжимаем все(редуцируемые) процессы.

**MPI\_Finalize();**

Закрываем среду библиотеки.

# Результаты замеров времени выполнения

Программа запускалась со следующими параметрами:

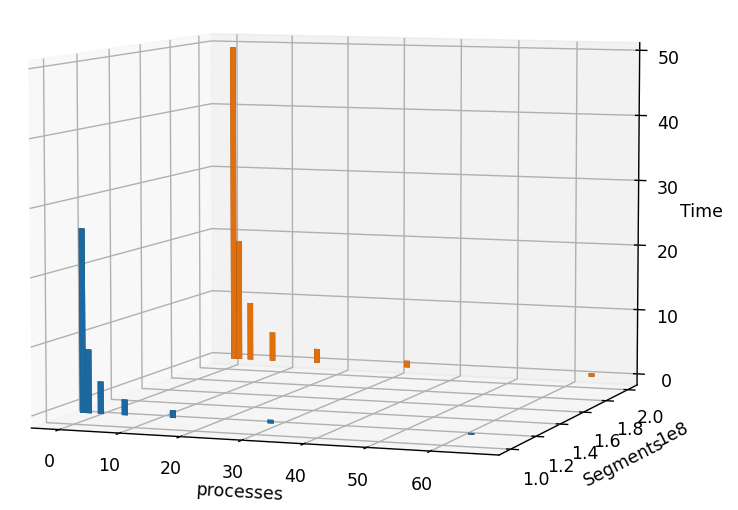
* Количество интервалов {100000000, 200000000}
* Количество потоков {1,2,4,8,16,32,64}

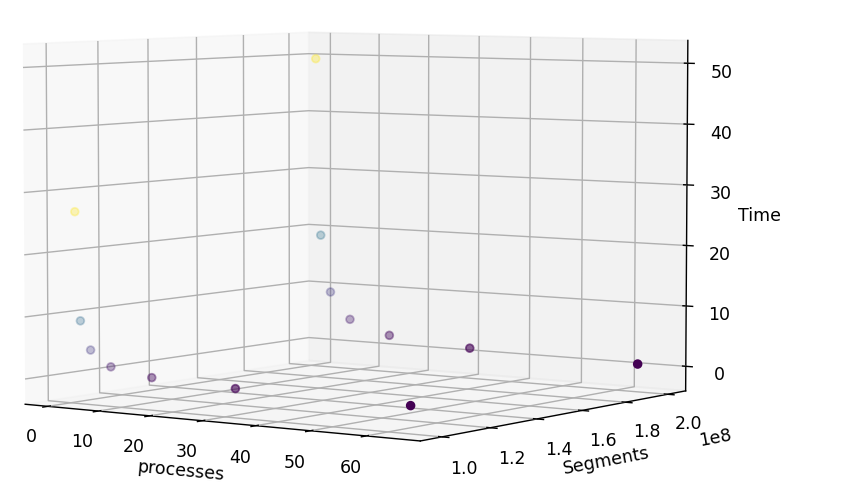
## Таблица

Таблица затраченного времени для запуска на виртуальной машине

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 100000000 | 26.826566 | 9.305235 | 4.724391 | 2.291392 | 1.121618 | 0.525470 | 0.244212 |
| 200000000 | 49.415730 | 18.799150 | 8.999815 | 4.475489 | 2.205053 | 1.042899 | 0.524470 |

## График





# Анализ результатов

1. В обоих случаях количества интервалов при изменении количества процессов результаты про времени формируют график z=1/х, где z-время, x-кол-во процессов. Сложно сказать почему так.

# Выводы

Изучена библиотека MPI для написания параллельных программ.