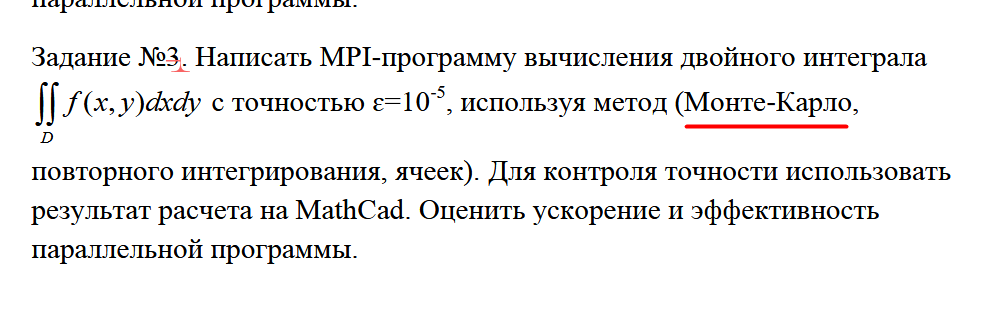
Контрольная №5

Задача на приближенное вычисление определенного интеграла

Боязитов Вадим, 932125

01.05.2025

## Задание 1



### Алгоритм

1.Инициализация MPI

Запускаем MPI среду, определяем ранг процесса и общее количество процессов.

2. Определение функции

CalculateFunctionValue(x,y) - вычисляет значение функции (x\*y)²

IsPointInIntegrationDomain(x,y) - проверяет условия попадания в D

3. Подготовка к вычислениям

Инициализация генератора случайных чисел с уникальным seed для каждого процесса

Распределение общего количества точек (total\_point\_count) между процессами

Замер времени начала вычислений

4. Метод Монте-Карло:

Генерация случайной точки в [0.5,2.0]×[0.5,2.0]

Для каждой точки проверяется принадлежность к области D

Если точка принадлежит области:

Увеличивается счетчик таких точек

Добавляется значение функции в этой точке к локальному интегралу

5. Сбор результатов

Суммирование всех points\_in\_domain (MPI\_SUM)

Суммирование всех local\_integral\_sum (MPI\_SUM)

Замер времени окончания вычислений

6. Вычисление интеграла

На процессе 0 вычисляется приближенное значение интеграла

7. Вывод результатов

***Выбор области интегрирования***

При x = 0.5 из условия x\*y > 1 получаем y > 2 → но это выходит за границы y\_max\_bound = 2.0

При x = 2.0 из x\*y < 2 получаем y < 1

Условие |x - y| < 1 добавляет ограничение "полосы" вокруг линии y = x

Пересечение xy=1 и y=x+1 → x(x+1)=1 → x ≈ 0.618

Пересечение xy=2 и y=x-1 → x(x-1)=2 → x ≈ 1.618

для простоты взяты округлённые значения [0.5, 2.0]

### Программная реализация

Окружение Linix. Язык C.

https://github.com/katet3/tsu\_parallel\_programming/blob/main/lab\_5/lab\_5.c

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <stdbool.h>

double CalculateFunctionValue(double x, double y) {

return (x \* y) \* (x \* y);

}

bool IsPointInIntegrationDomain(double x, double y) {

return (x \* y > 1 && x \* y < 2) && (fabs(x - y) < 1);

}

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

int process\_rank;

int process\_count;

long long total\_point\_count = 10000000;

long long local\_point\_count;

long long points\_in\_domain = 0;

double x\_coordinate, y\_coordinate;

double local\_integral\_sum = 0.0;

double global\_integral\_sum = 0.0;

double computation\_start\_time, computation\_end\_time;

double sequential\_time = 0.0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &process\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &process\_count);

local\_point\_count = total\_point\_count / process\_count;

srand(time(NULL) + process\_rank);

const double x\_min\_bound = 0.5;

const double x\_max\_bound = 2.0;

const double y\_min\_bound = 0.5;

const double y\_max\_bound = 2.0;

const double generation\_area = (x\_max\_bound - x\_min\_bound) \* (y\_max\_bound - y\_min\_bound);

// Измеряем время последовательного выполнения на процессе 0

if (process\_rank == 0) {

double seq\_start = MPI\_Wtime();

long long seq\_points\_in\_domain = 0;

double seq\_integral\_sum = 0.0;

for (long long i = 0; i < total\_point\_count; i++) {

double x = x\_min\_bound + (x\_max\_bound - x\_min\_bound) \* rand() / RAND\_MAX;

double y = y\_min\_bound + (y\_max\_bound - y\_min\_bound) \* rand() / RAND\_MAX;

if (IsPointInIntegrationDomain(x, y)) {

seq\_integral\_sum += CalculateFunctionValue(x, y);

seq\_points\_in\_domain++;

}

}

sequential\_time = MPI\_Wtime() - seq\_start;

}

// Синхронизация перед началом параллельного вычисления

MPI\_Barrier(MPI\_COMM\_WORLD);

computation\_start\_time = MPI\_Wtime();

for (long long iteration = 0; iteration < local\_point\_count; iteration++) {

x\_coordinate = x\_min\_bound + (x\_max\_bound - x\_min\_bound) \* rand() / RAND\_MAX;

y\_coordinate = y\_min\_bound + (y\_max\_bound - y\_min\_bound) \* rand() / RAND\_MAX;

if (IsPointInIntegrationDomain(x\_coordinate, y\_coordinate)) {

local\_integral\_sum += CalculateFunctionValue(x\_coordinate, y\_coordinate);

points\_in\_domain++;

}

}

long long total\_points\_in\_domain;

MPI\_Reduce(&points\_in\_domain, &total\_points\_in\_domain, 1, MPI\_LONG\_LONG\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&local\_integral\_sum, &global\_integral\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

computation\_end\_time = MPI\_Wtime();

if (process\_rank == 0) {

double parallel\_time = computation\_end\_time - computation\_start\_time;

double integral\_approximation = generation\_area \* global\_integral\_sum / total\_point\_count;

// Расчет ускорения и эффективности

double speedup = sequential\_time / parallel\_time;

double efficiency = speedup / process\_count;

printf("\n=== Integration Results ===\n");

printf("Approximate integral value = %.10f\n", integral\_approximation);

printf("Points inside domain = %lld / %lld (%.2f%%)\n",

total\_points\_in\_domain, total\_point\_count,

100.0 \* total\_points\_in\_domain / total\_point\_count);

printf("\n=== Performance Metrics ===\n");

printf("Sequential time = %.4f seconds\n", sequential\_time);

printf("Parallel time = %.4f seconds\n", parallel\_time);

printf("Speedup = %.2f\n", speedup);

printf("Efficiency = %.2f%%\n", efficiency \* 100);

if (total\_points\_in\_domain > 0) {

double standard\_error = generation\_area \*

sqrt(global\_integral\_sum / total\_point\_count -

(integral\_approximation / generation\_area) \*

(integral\_approximation / generation\_area)) /

sqrt(total\_point\_count);

printf("\nEstimated standard error = %.2e\n", standard\_error);

}

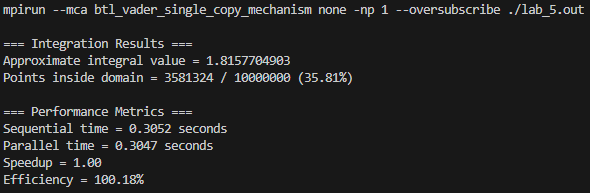
}

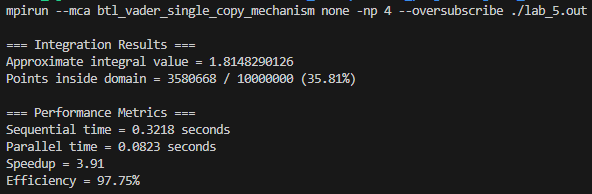
MPI\_Finalize();

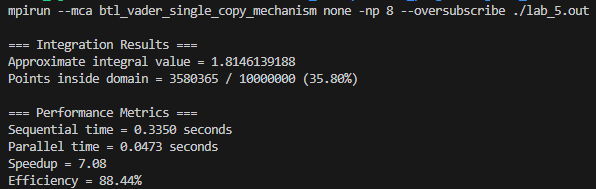
return 0;

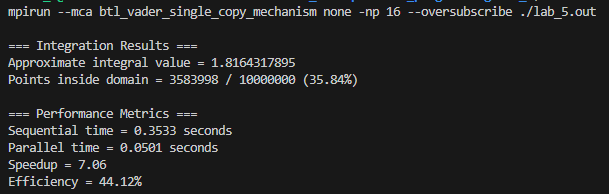
}

### Тестирование









### Вывод

### С ростом кол-ва процессов, уменьшается время на вычисление.Ускорение увеличивается, а эффективность будет падать за счет большого кол-ва процессов.