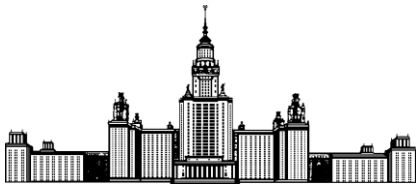

Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра математического моделирования гетерогенных систем

Практическое задание №1 по курсу «Суперкомпьютерное моделирование и технологии»

Выполнил:
студент 608 группы

Трибрат Вадим Дмитриевич

Москва, 2022

1 Постановка задачи.

Необходимо вычислить многомерный интеграл $\int_{-1}^{0}\int_{-1}^{0}\int_{-1}^{0} x^3y^2z\,dxdydz$ методом Монте-Карло с применением технологии MPI (мастер - рабочие).

Программа получает в качестве аргумента командной строки требуемую точность ε и выводит четыре числа:

• Посчитанное приближённое значение интеграла.

• Ошибка посчитанного значения: модуль разности между приближённым и точным значениями интеграла.

• Количество сгенерированных случайных точек.

• Время работы программы в секундах.

2 Аналитическое решение интеграла.

$$\int_{-1}^{0}\int_{-1}^{0}\int_{-1}^{0} x^3y^2z\,dxdydz = \frac{1}{2}\int_{-1}^{0}dx\int_{-1}^{0}x^3y^2 * (0^2 - (-1)^2)dy$$

$$= -\frac{1}{2 * 3}\int_{-1}^{0}x^3 * (0^3 - (-1)^3)dy$$

$$= -\frac{1}{6 * 4} * (0^4 - (-1)^4) = \frac{1}{24}$$

3 Численный подход.

Метод Монте-Карло основан на законе больших чисел. Его основная идея заключается в следующем: выберем произвольную случайную величину ξ и рассмотрим другую с.в. $\zeta = \frac{\mathrm{f}(\xi)}{\mathrm{p}(\xi)}$, где $\mathrm{p}(\xi)$ – плотность распределения ξ. Тогда $\mathrm{E}(\zeta) = \int \mathrm{f}(\xi)\,\mathrm{d}\xi$. А согласно закону больших чисел, мы можем оценить $\mathrm{E}(\zeta)$ как среднее арифметическое $\frac{1}{N}\sum\frac{\mathrm{f}(\xi_i)}{\mathrm{p}(\xi_i)}$.

4 Описание программы.

Мастер генерирует случайные точки, равномерно распределенные на кубе $[-1, 0]^3$. Всего генерируется N точек на каждой итерации алгоритма,

затем они делятся на приблизительно равные подмассивы и рассылаются работникам. Рассылка производится с помощью функции MPI_Scatterv, что позволяет экономить память в рабочих процессах. Затем каждый поток вычисляет свою часть суммы и с помощью операции редукции вычисляется текущее приближение интеграла. Если точность недостаточна, то процесс повторяется. Вычислялось время работы рабочих процессов за вычетом времени генерации точек, т.к. время генерации много больше времени подсчета суммы элементов массива.

5 Исследование масштабируемости.

Таблица для seed = 17.

Таблица 1. Результаты работы программы.

Точность ε	Число процессов	Время работы (с)	Ускорение	Ошибка
$3.0 * 10^{-5}$	2	0.05	1	$2.8 * 10^{-5}$
	4	0.0317	1.76	$2.8 * 10^{-5}$
	16	0.02	2.5	$2.8 * 10^{-5}$
$5.0 * 10^{-6}$	2	0.182	1	$4.7 * 10^{-6}$
	4	0.109	1.67	$4.7 * 10^{-6}$
	16	0.074	2.46	$4.7 * 10^{-6}$
$1.5 * 10^{-6}$	2	0.188	1	$4 * 10^{-7}$
	4	0.115	1.63	$4 * 10^{-7}$
	16	0.082	2.29	$4 * 10^{-7}$

Таблица 2. Зависимость результатов от seed.

Значение seed	Точность ε	Число процессов	Время работы (с)	Кол-во точек
17	$3.0 * 10^{-5}$	2	0.05	5’000’000
		4	0.0317	
		16	0.02	
	$5.0 * 10^{-6}$	2	0.182	18’200’00
		4	0.109	
		16	0.074	
	$1.5 * 10^{-6}$	2	0.188	19’000’000
		4	0.115	
		16	0.082	
57	$3.0 * 10^{-5}$	2	0.063	600’000
		4	0.044	
		16	0.029	
	$5.0 * 10^{-6}$	2	0.01	1’000’000
		4	0.072	
		16	0.044	
	$1.5 * 10^{-6}$	2	0.01	1’000’000
		4	0.07	
		16	0.06	

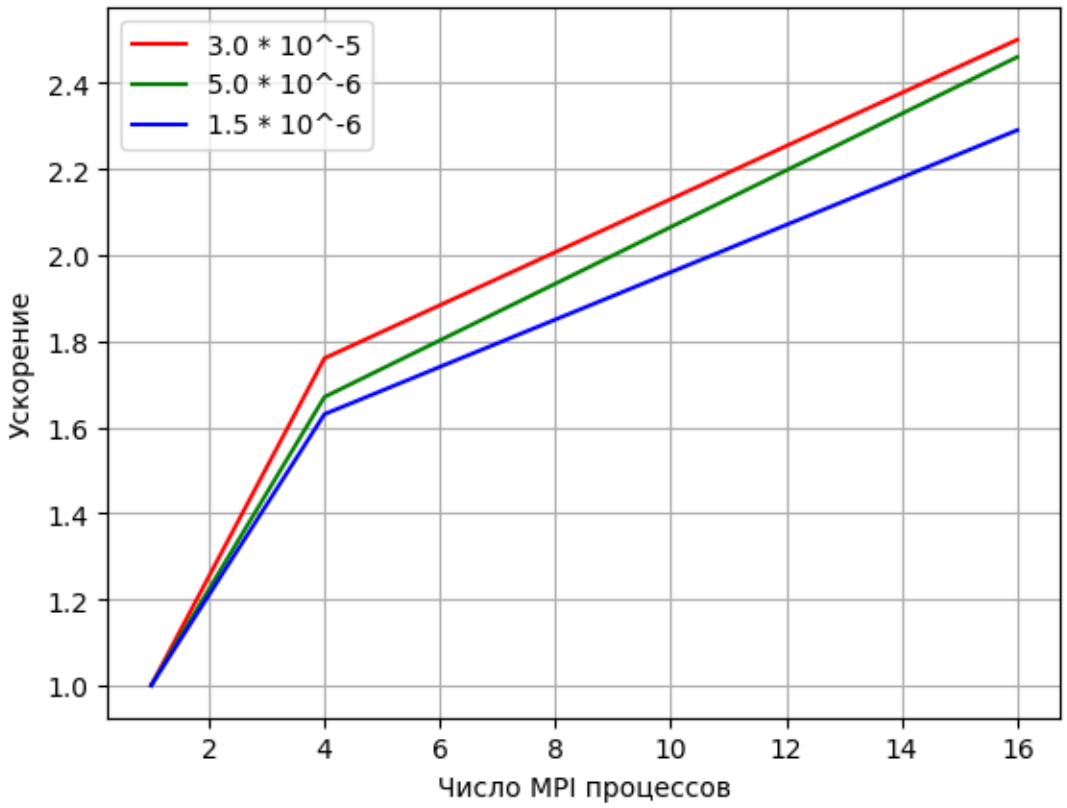

Рис. 1. Зависимость ускорения от числа поток.

6 Заключение.

В силу особенностей задачи, а именно низкой сложности вычисления суммы элементов массива, данный подход (мастер-рабочие) показывает слабую масштабируемость.

Листинг 1.

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

int main(int argc, char** argv)
{
    int rank, size, flag = 0, total_N = 0, bufElems = 0;
    int *displs, *scounts;
    const int N = 100000;
    double eps = strtold(argv[1], NULL), glob_sum = 0, sum = 0, acc_sum = 0;
    double diff_time = 0, max_time, gen_time = 0;
    const double real_val = 1.0 / 24;
    MPI_Status status_x, status_y, status_z, status;
    MPI_Init(&argc, &argv);
    srand48(57);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double *x, *y, *z, * x_sl, *y_sl, *z_sl;
    if (!rank)
    {
        x = malloc(N * sizeof(double));
        y = malloc(N * sizeof(double));
        z = malloc(N * sizeof(double));
    }
    else
    {
        bufElems = (rank != size - 1) ? N / (size - 1) : N - (N / (size - 1)) *
(size - 2);
    }
    x_sl = malloc(bufElems * sizeof(double));
    y_sl = malloc(bufElems * sizeof(double));
    z_sl = malloc(bufElems * sizeof(double));
    displs = (int*)malloc(size * sizeof(int));
    scounts = (int*)malloc(size * sizeof(int));
    displs[0] = 0;
    scounts[0] = 0;
    for (int i = 1; i < size - 1; ++i)
    {
        displs[i] = (N / (size - 1)) * (i - 1);
        scounts[i] = N / (size - 1);
    }
    displs[size - 1] = (N / (size - 1)) * (size - 2);
    scounts[size - 1] = N - (N / (size - 1)) * (size - 2);
    diff_time -= MPI_Wtime();
```

```
    do
    {
        if (rank == 0)
        {
            gen_time -= MPI_Wtime();
            for (int i = 0; i < N; ++i)
            {
                x[i] = -1 * drand48();
                y[i] = -1 * drand48();
                z[i] = -1 * drand48();
                //printf("%d (%f %f %f)\n", rank, x[i], y[i], z[i]);
            }
            gen_time += MPI_Wtime();
            //printf("\n\n");
        }
        MPI_Scatterv(x, scounts, displs, MPI_DOUBLE, x_sl, bufElems, MPI_DOUBLE,
0, MPI_COMM_WORLD);
        MPI_Scatterv(y, scounts, displs, MPI_DOUBLE, y_sl, bufElems, MPI_DOUBLE,
0, MPI_COMM_WORLD);
        MPI_Scatterv(z, scounts, displs, MPI_DOUBLE, z_sl, bufElems, MPI_DOUBLE,
0, MPI_COMM_WORLD);
        if (rank)
        {
            sum = 0;
            for (int i = 0; i < bufElems; ++i)
            {
                //printf("%d (%f %f %f)\n", rank, x_sl[i], y_sl[i], z_sl[i]);
                sum += x_sl[i] * x_sl[i] * x_sl[i] * y_sl[i] * y_sl[i] * z_sl[i];
            }
            //printf("\n\n");
        }
        MPI_Reduce(&sum, &glob_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (!rank)
        {
            total_N += N;
            acc_sum += glob_sum;
            flag = fabs(acc_sum / total_N - real_val) < eps;
        }
        MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (!flag);
    diff_time += MPI_Wtime();
    MPI_Bcast(&gen_time, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    diff_time -= gen_time;
    //printf("%d time=%f\n", rank, diff_time);
    if (!rank)
    {
        diff_time = 0;
    }
    MPI_Reduce(&diff_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

```
    if (!rank)
    {
        //printf("Gen_time = %f\n", gen_time);
        printf("Total_N=%d, value = %.10f,  error=%.10f, time=%f\n",
            total_N, acc_sum / total_N, fabs(acc_sum / total_N - real_val),
max_time);
    }
    if (!rank)
    {
        free(x);
        free(y);
        free(z);
    }
    free(x_sl);
    free(y_sl);
    free(z_sl);
    free(displs);
    free(scounts);
    MPI_Finalize();
    return 0;
}
```