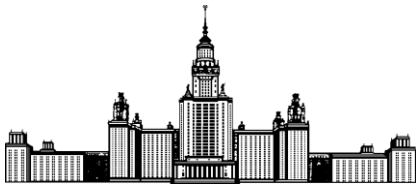


Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра математического моделирования гетерогенных систем

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 $\frac{1}{N}\sum\frac{\mathrm{f}(\xi_i)}{\mathrm{p}(\xi_i)}$.

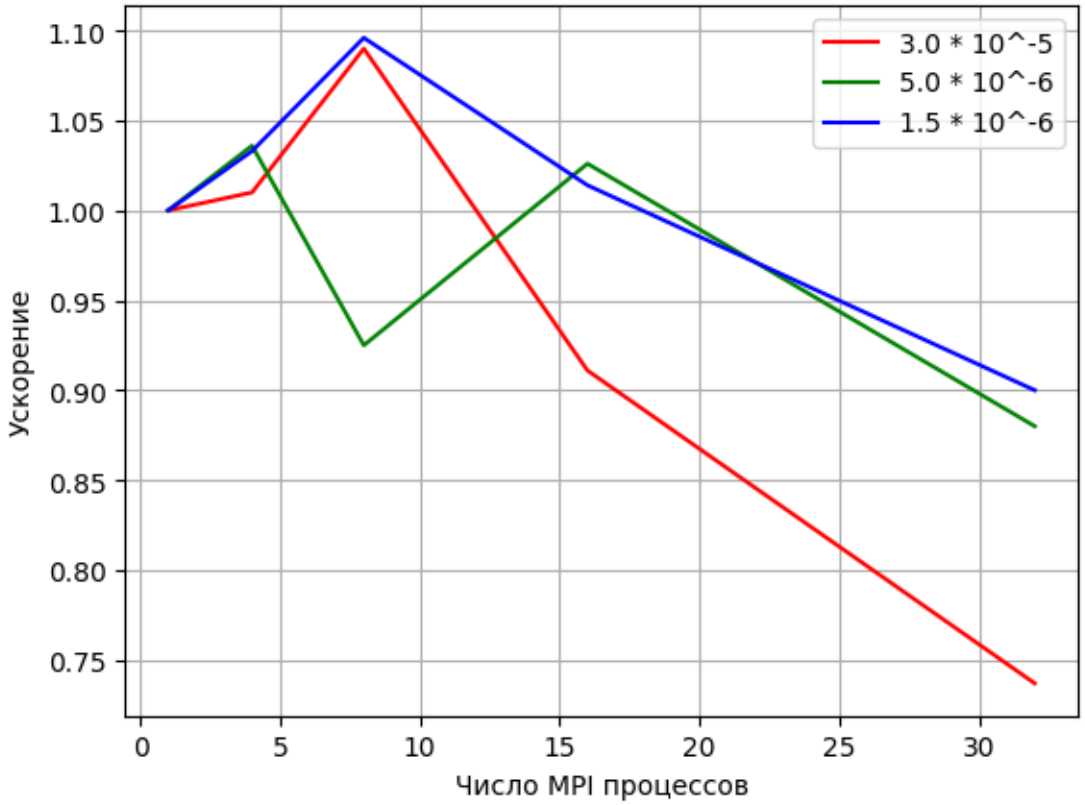
4 Описание программы.

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_Recv, 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.

5 Исследование масштабируемости.

Точность ε	Число процессов	Время работы (с)	Ускорение	Ошибка
$3.0 * 10^{-5}$	2	0.019375	1	$2 * 10^{-5}$
	4	0.019179	1.01	$2 * 10^{-5}$
	8	0.017768	1.09	$2 * 10^{-5}$
	16	0.021252	0.911	$2 * 10^{-5}$
	32	0.026061	0.737	$2 * 10^{-5}$
$5.0 * 10^{-6}$	2	1.222195	1	$4.7 * 10^{-6}$
	4	1.179662	1.036	$4.7 * 10^{-6}$
	8	1.321153	0.925	$4.7 * 10^{-6}$
	16	1.190702	1.026	$4.7 * 10^{-6}$
	32	1.378899	0.88	$4.7 * 10^{-6}$
$1.5 * 10^{-6}$	2	1.278977	1	$1.3 * 10^{-6}$
	4	1.237776	1.033	$1.3 * 10^{-6}$
	8	1.166506	1.096	$1.3 * 10^{-6}$
	16	1.261415	1.014	$1.3 * 10^{-6}$
	32	1.410919	0.9	$1.3 * 10^{-6}$



6 Заключение.

В силу особенностей задачи, а именно низкой сложности вычисления суммы элементов массива, данный подход (мастер-рабочие) показывает слабую масштабируемость. В работе рассматривались и другие способы рассылки данных рабочим, однако во всех случаях не было замечено значительной разницы в результатах.

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;
    const int N = 10000;
    double eps = strtold(argv[1], NULL), glob_sum = 0, sum = 0, acc_sum = 0;
    double diff_time, max_time, start_time, end_time;
    const double real_val = 1.0 / 24;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    srand48(17);
    start_time = MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double x[N], y[N], z[N];
    do
    {
        if (rank == 0)
        {
            sum = 0;
            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]);
            }
            for (size_t k = 1; k < size - 1; k++)
            {
                MPI_Send(x + (k - 1) * (N / (size - 1)), N / (size - 1),
MPI_DOUBLE, k, 0, MPI_COMM_WORLD);
                MPI_Send(y + (k - 1) * (N / (size - 1)), N / (size - 1),
MPI_DOUBLE, k, 0, MPI_COMM_WORLD);
                MPI_Send(z + (k - 1) * (N / (size - 1)), N / (size - 1),
MPI_DOUBLE, k, 0, MPI_COMM_WORLD);
            }
            //printf("\n\n");
            MPI_Send(x + (size - 2) * (N / (size - 1)), N - N / (size - 1) * (size
- 2), MPI_DOUBLE, size - 1, 0, MPI_COMM_WORLD);
            MPI_Send(y + (size - 2) * (N / (size - 1)), N - N / (size - 1) * (size
- 2), MPI_DOUBLE, size - 1, 0, MPI_COMM_WORLD);
```

```
            MPI_Send(z + (size - 2) * (N / (size - 1)), N - N / (size - 1) * (size
- 2), MPI_DOUBLE, size - 1, 0, MPI_COMM_WORLD);
        }
        else
        {
            MPI_Status status_x, status_y, status_z, status;
            int bufElems;
            sum = 0;
            MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_DOUBLE, &bufElems);
            double x_sl[bufElems], y_sl[bufElems], z_sl[bufElems];
            MPI_Recv(x_sl, bufElems, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status_x);
            MPI_Recv(y_sl, bufElems, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status_y);
            MPI_Recv(z_sl, bufElems, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status_z);
            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");
        }
        total_N += N;
        MPI_Allreduce(&sum, &glob_sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        acc_sum += glob_sum;
        flag = fabs(acc_sum / total_N - real_val) < eps;
    } while (!flag);
    end_time = MPI_Wtime();
    diff_time = end_time - start_time;
    MPI_Reduce(&diff_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    if (!rank)
    {
        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);
    }
    MPI_Finalize();
    return 0;
}
```