Отчет по второму заданию курса “Суперкомпьютерное моделирование и технологии” Численное интегрирование методом Монте-Карло

Сюй Минчуань, группа 617, номер варианта: 7

27 октября 2022 г.

1 Постановка задачи

Пусть задана функция $f(x, y, z)$ - непрерывна в ограниченной замкнутой области $G \subset \mathbb{R}^3$. Требуется вычислить определенный интеграл

$$I = \iiint_G f(x, y, z)dxdydz$$

В частности, для варианта 7, заданная функция и область имеют вид:

$$f(x, y, z) = \sqrt{y^2 + z^2}, \quad G = \{(x, y, z) : 0 \leqslant x \leqslant 2, y^2 + z^2 \leqslant 1\}$$

Данный интеграл может вычисляться аналитически следующем образом:

$$\iiint_G \sqrt{y^2 + z^2}dxdydz = \int_0^2 dx \iint_{y^2+z^2\leqslant 1} \sqrt{y^2 + z^2}dydz = \int_0^2 dx \int_0^{2\pi} d\phi \int_0^1 r \cdot rdr = \frac{4}{3}\pi$$

2 Численный метод решения

Пусть область G ограничена параллелепипедом $\Pi = \{(x, y, z) : 0 \leqslant x \leqslant 2, -1 \leqslant y \leqslant 1, -1 \leqslant z \leqslant 1\}$, и рассмотрим функцию

$$F(x, y, z) = \begin{cases} \sqrt{y^2 + z^2}, & (x, y, z) \in G \\ 0, & (x, y, z) \notin G \end{cases}$$

Значит, $I = \iiint_G f(x, y, z)dxdydz = \iiint_\Pi F(x, y, z)dxdydz$. Возмьем n случайных точек $p_1, p_2, \ldots$, равномерно распределенных в Π, тогда в качестве приближенного значения интеграла предлагается использовать выражение

$$I \approx |\Pi| \cdot \frac{1}{n} \sum_{i=1}^{n} F(p_i)$$

где $|\Pi|$ - объем параллелепипеда. Для удобства иллюстрации процесса работы численного решения предлагается схема алгоритма:

Algorithm 1 Вычисление интеграла методом Монте-Карло

инициализировать число случайных точек n, заданной точность ε, сумма значений функции F в точках, которые попадут в область G sum=0, известное точное значение интеграла I.
цикл по i от 1 до бесконечности:
    генерировать 3 случайных чисел x, y, z
    если $(x, y, z) \in G$: sum += $f(x, y, z)$
    integral = объем G / i * sum
    если $|\text{I} - \text{integral}| \leq \varepsilon$: **выход из цикла**
вывод: приближенное значение интеграла integral

3 Описание программной реализации

Для варианта 7 предлагается метод расспараллеливания “независимая генерация точек MPI-процессами” - параллелельные процессы генерируют случайные точки независимо друг от друга. Общий принцип работы реализованной программы такой:

1. Все процессы вычисляют свою часть суммы через формулу
2. Вычисляется общая сумма с помощью операции редукции
3. Вычисляется ошибка расчета, в случае ошибка выше требуемой точности, то генерируются дополнительные точки и продолжается расчет

Необходимо инициализировать генератор псевдослучайных чисел в разных MPI-процессами с разными seed’ами. Также, для увеличения эффективности параллельной реализации алгоритма вводится параметр m, значение которого равно числу новых сгенерированных случайных точек на каждом процессе при новой итерации расчета. Таким образом, при каждом вызове функции MPI_Reduce у нас уже отсэмплированы m*num_procs новых точек.

Прилагается код реализации:

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

double f(double x, double y, double z){
    return sqrt(y*y+z*z);
}

double generate_point(double left_range, double right_range){
    return ((double) rand() / RAND_MAX) * (right_range - left_range) + left_range;
}

int main(int argc, char * argv[]){
        // to ensure the randomness for every process
        // add ten more points for every fail of approximating real integral
        srand(time(0));
        int n_count = 1, my_id, num_procs, i;
        double real_integral = 4.0 / 3 * M_PI, sum = 0.0, root_value, integral,
    current_gap;
        int m = atoi(argv[1]);
        double eps = atof(argv[2]);
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

```
25          MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
26          double start = MPI_Wtime();
27
28          while(1){
29              for (i = my_id + 1; i <= m * num_procs; i+= num_procs){
30                      double x = generate_point(0.0, 2.0);
31                      double y = generate_point(-1.0, 1.0);
32                      double z = generate_point(-1.0, 1.0);
33                      if (y*y+z*z<=1){
34                          sum += f(x,y,z);
35                      }
36              }
37
38              MPI_Reduce(&sum, &root_value, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
39              if (!my_id){
40                  integral = 8.0 * root_value / (m * num_procs * n_count);
41                  current_gap = fabs(integral - real_integral);
42
43                  if (current_gap <= eps){
44                      printf("\nWell done! The integral is approximately %.6f \n",
    integral);
45                      printf("The final computing error is %.6f\n", current_gap);
46                      printf("Generated %d random points\n", m * num_procs * n_count
    );
47                      break;
48                  }
49
50
51                  else {
52                      //printf("simulated points: %d, The current gap: %.8f \n", m *
     num_procs * n_count, current_gap);
53                      n_count += 1;
54                  }
55              }
56          }
57          double end = MPI_Wtime();
58          printf("The running time is %.6f s\n", end - start);
59          MPI_Abort(MPI_COMM_WORLD, 666);
60          return 0;
61
62  }
```

Листинг 1: Реализация на C

4 Исследование масштабируемости программы на вычислительных системах

На основе написанной программы провел разные эксперименты, чтобы исследовать масштабируемость реализованной программы. Программа запустилась для различного числа MPI-процессов и различных значений входного параметра ε. Заполнил таблицу с полученными результатами. Построил графики зависимости ускорения и времени выпонения программы от числа используемых MPI-процессов для каждого значения ε.

Под ускорением программы, запущенной на p MPI-процессах, понимается величина

$$S_p = \frac{T_1}{T_p}$$

где T_1 - время работы программы на 1 MPI-процессе, T_p - время работы программы на p MPI-процессах. Для каждого значения в таблице ниже берется среднее время выполнения на 10 запусков.

Выводы:

1. программа на Polus как-то может масштабироваться: при увеличении числа процессоров было больше ускорения, но делако от линейного отношения не дойдет, так как при увеличения числа процессоров накладные расходы стали всё большими.

2. маленькая точность, безусловно, повлияла на время выполнения (стала всё дольше), но и повлияла на ускорение: эффект ускорения стал меньше, так как при маленькой точности требуется генерировать больше точек на вычисление, соответственно и требуются дополнительные накладные расходы на передачу данных.

Точность	Число MPI-процессов	Время работы программы(s)	Ускорение	Ошибка
$3.0 \cdot 10^{-5}$	1	0.0268840	1.00	$1.5 \cdot 10^{-5}$
	4	0.0136136	1.97	$1.2 \cdot 10^{-5}$
	16	0.0107648	2.50	$2.1 \cdot 10^{-5}$
	32	0.0092318	2.91	$1.0 \cdot 10^{-5}$
$5.0 \cdot 10^{-6}$	1	0.0839926	1.00	$2.0 \cdot 10^{-6}$
	4	0.07308984	1.15	$3.4 \cdot 10^{-6}$
	16	0.0492395	1.71	$3.9 \cdot 10^{-6}$
	32	0.0358244	2.34	$2.7 \cdot 10^{-6}$
$1.5 \cdot 10^{-6}$	1	0.2042897	1.00	$1.1 \cdot 10^{-6}$
	4	0.1403447	1.46	$5.0 \cdot 10^{-7}$
	16	0.1239581	1.65	$1.2 \cdot 10^{-6}$
	32	0.1339453	1.53	$8.0 \cdot 10^{-7}$

Таблица 1: Таблица с результатами расчетов для системы Polus

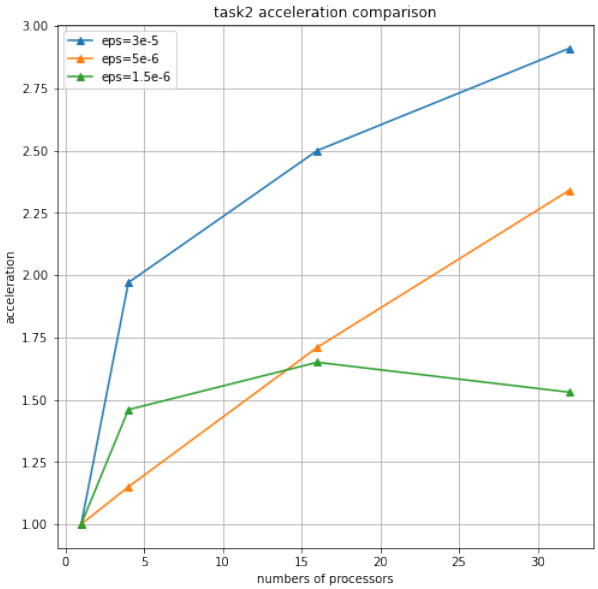

Рис. 1: Зависимость ускорения программы от числа используемых MPI-процессов для каждого значения ε

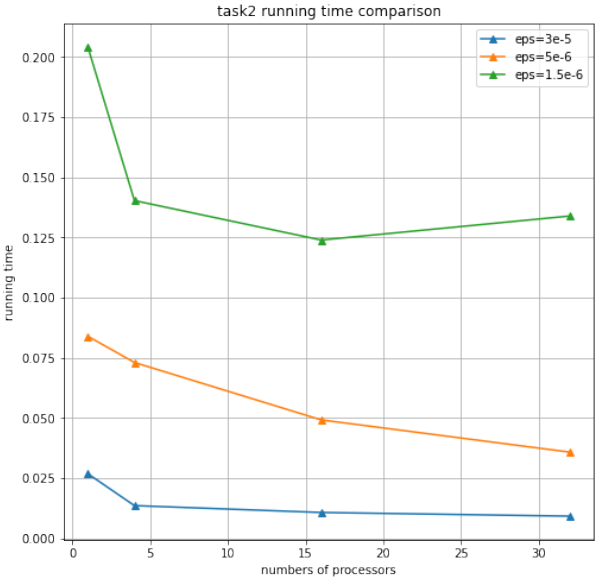

Рис. 2: Зависимость времени выполнения программы от числа используемых MPI-процессов для каждого значения ε