Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчёт по курсу
«Суперкомпьютерное моделирование и технологии»

«Численное интегрирование многомерных функций методом Монте-Карло»

Вариант 6

Студент 608 группы
И. А. Кулешов

Москва, 2022

1 Постановка задачи и методы решения

1.1 Математическая постановка задачи

Функция $f(x, y, z)$ — непрерывна в ограниченной замкнутой области $G \subset R^3$. Требуется вычислить определённый интеграл (вариант 6):

$$I = \iiint\limits_G \sin(x^2 + z^2) \cdot \ y \ dx\, dy\, dz,$$

где область $G = \{(x, y, z) : x^2 + y^2 + z^2 \leqslant 1, x > 0, y > 0, z > 0\}$.

1.2 Вычисление точного значения

$$I = \iiint\limits_G \sin(x^2 + z^2) \cdot \ y \ dx\, dy\, dz = \Big\{x = r\cos(\phi), y = y, z = r\sin(\phi),\ dx\, dy\, dz \ = r\, d\phi\, dr\, dy\Big\}$$
$$= \int_0^{\frac{\pi}{2}} d\phi \int_0^1 r\sin(r^2)\, dr \int_0^{\sqrt{1-r^2}} y\, dy = \int_0^{\frac{\pi}{2}} d\phi \int_0^1 r\ \sin(r^2)\frac{1-r^2}{2}\, dr = \Big\{\gamma = r^2, dr = \frac{d\gamma}{2r}\Big\}$$
$$= \frac{\pi}{8}\int_0^1 (1-\gamma)\sin(\gamma) = \frac{\pi}{8}\Big(1 - \sin(1)\Big) \approx 0.06225419868$$

1.3 Численный метод решения

Пусть область G ограниченна параллелепипедом: $\Pi = \begin{cases} a_1 \leqslant x \leqslant b_1 \\ a_2 \leqslant y \leqslant b_2 \\ a_3 \leqslant z \leqslant b_3 \end{cases}$

Рассмотрим функцию $F(x, y, z) = \begin{cases} f(x, y, z) & (x, y, z) \in G, \\ 0 & (x, y, z) \notin G, \end{cases}$

Преобразуем искомый интеграл:

$$I = \iiint\limits_G f(x, y, z)\ dx\, dy\, dz = \iiint\limits_\Pi F(x, y, z)\ dx\, dy\, dz$$

Пусть $p_1(x_1, y_1, z_1)$, $p_2(x_2, y_2, z_2)$, ... — случайные точки, равномерно распределённые в Π. Возьмём n таких случайных точек. В качестве приближённого значения интеграла предлагается использовать выражение:

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

где $|\Pi|$ - объём параллелепипеда Π. $|\Pi| = (b_1 - a_1)(b_2 - a_2)(b_3 - a_3) = (1-0)(1-0)(1-0) = 1$

Вариант 6 предлагает метод распараллеливания независимой генерацией точек MPI- процессами для решения задачи.

2 Алгоритм решения

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

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

    const double pi = 4.0 * atan(1.0);
    // 0.06225419868
    double exact_integral_value = pi / 8.0 * (1 - sin(1.0));
    double P_region_volume = 1.0 * 1.0 * 1.0;
    double integral_approx;
    double x_rand, y_rand, z_rand;
    double func_sum;
    int64_t n_dots = 0;
    double iteration_sum;
    int iterations = 0;
    double total_sum = 0.0;
    double error;
    double eps;
    double max_duration;
    int num_procs, my_rank, root = 0;

    if (argc > 1 && argv[1] != NULL) {
        sscanf(argv[1], "%lf", &eps);
    } else {
        fprintf(stderr,"eps is missing!\n");
        return 1;
    }
    int seed_bias = 0;
    if (argc > 2 && argv[2] != NULL) {
        sscanf(argv[2], "%d", &seed_bias);
    } else {
        fprintf(stderr,"seed_bias is missing!\n");
        return 2;
    }
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // balance between common sense and precision
    int64_t n_dots_per_cycle = (int) pow(2, (int) log2(1/eps));
    int64_t n_dots_per_proc = n_dots_per_cycle;
    if (num_procs > 1) {
        n_dots_per_proc /= num_procs;
        n_dots_per_proc *= 2;
```

```
    }
    int my_seed = seed_bias * 100 + my_rank;
    srand(my_seed);

    double start = MPI_Wtime();
    do {
        iterations += 1;
        func_sum = 0;
        for (size_t i = 0; i < n_dots_per_proc; ++i) {
            x_rand = (double) rand() / RAND_MAX;
            y_rand = (double) rand() / RAND_MAX;
            z_rand = (double) rand() / RAND_MAX;
            if ((x_rand * x_rand + y_rand * y_rand + z_rand * z_rand) > 1) {
                func_sum += 0;
            } else {
                func_sum += sin(x_rand * x_rand + z_rand * z_rand) * y_rand;
            }
        }
        MPI_Reduce(&func_sum, &iteration_sum, 1,
                   MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);
        if (my_rank == root) {
            n_dots += n_dots_per_proc;
            total_sum += (1.0 / num_procs) * iteration_sum;
            integral_approx = P_region_volume * (1.0 / n_dots) * total_sum;
            error = fabs(integral_approx - exact_integral_value);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(&error, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
    } while (error > eps);
    double cur_rank_duration = MPI_Wtime() - start;

    MPI_Reduce(&cur_rank_duration, &max_duration, 1,
               MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);
    MPI_Finalize();
    if (my_rank == root) {
        printf("num_procs: %d, eps: %.10f, time: %.10f sec,"
               " integral approx value: %.10f, error: %.10f ,"
               " iterations: %d, dots generated: %ld, seed_bias: %d\n",
               num_procs, eps, max_duration, integral_approx,
               error, iterations, num_procs * n_dots, seed_bias);
    }
    return 0;
}
```

3 Результатами расчётов для системы Polus

Для уменьшения фактора влияния псевдослучайности на результат были проведены 4 серии измерений с разными параметрами seed функции srand().

После этого было произведено медианнное усреднение, дающее устойчивость к возможным выбросам, вызванных неудачной генерацией точек.

Из множества проверенных вариантов конфигурации параметров был выбран набор с лучшим результатом масштабирования.

3.1 Таблица усредненных результатов

Точность	Число MPI-процессов	Время работы программы (с)	Ускорение	Ошибка
$3.0 \cdot 10^{-5}$	1	0.0389530	1.000	0.00001545
	4	0.0133889	2.909	0.00001787
	16	0.0168472	2.312	0.00002137
	32	0.0090133	4.322	0.00002319
$5.0 \cdot 10^{-6}$	1	0.3380623	1.000	0.00000171
	4	0.2723511	1.241	0.00000278
	16	0.0673020	5.023	0.00000306
	32	0.0288433	11.721	0.00000453
$1.5 \cdot 10^{-6}$	1	1.2583672	1.000	0.00000125
	4	0.6463072	1.947	0.00000096
	16	0.4564240	2.757	0.00000097
	32	0.1323673	9.507	0.00000099

3.2 Графики усредненных результатов ускорения

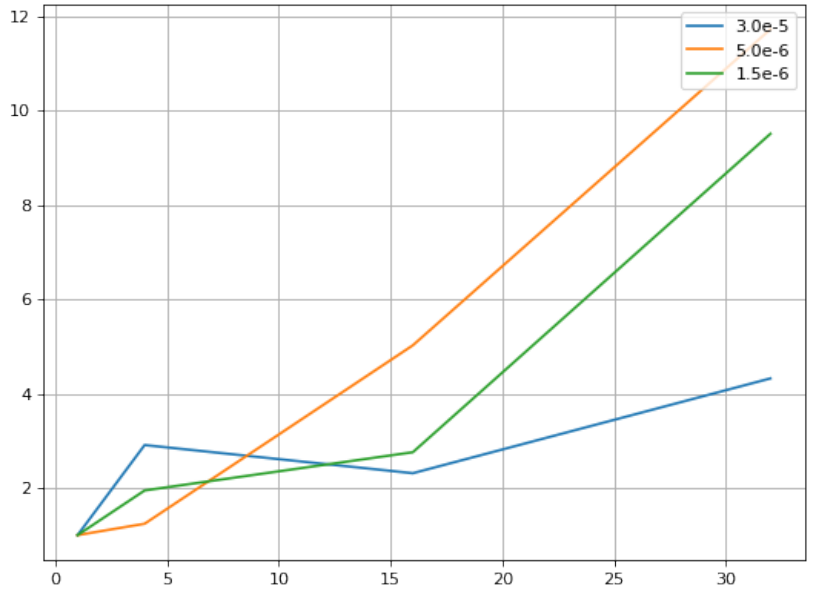


Рис. 1: График зависимости ускорения от числа процессов на Polus

3.3 Исследования

Будем называть итерацией один цикл обработки несколькими ядрами случайно сгенерированных точек. Будем обозначать $n\_dots\_per\_cycle$ число суммарное число точек, обработанных за одну итерацию всеми процессами, $n\_dots\_per\_proc$ - обработанных за одну итерацию каждым процессом, num_procs - число процессов, $num\_iterations$ - число итераций, потребовавшихся для достижения заданной точности ε. Очевидно, что $n\_dots\_per\_cycle = n\_dots\_per\_proc * num\_procs$.

Необходимо в первую очередь заострить внимание на том, что только время работы программы является плохой сравнительной метрикой масштабируемости программы. Это связано с тем, что при генерации неоправданно большого количестве точек (например порядка $\frac{1}{\varepsilon^2}$, дающих теоретическую оценку скорости сходимости) для первого расчета в цикле, программа будет выполнять ненужную работу и не сможет остановиться, пока не дойдет до первой проверки условия выхода из цикла. В таком случае мы будем видеть, что почти во всех случаях параметр $num\_iterations$ будет равен единице вне зависимости от количества ядер.

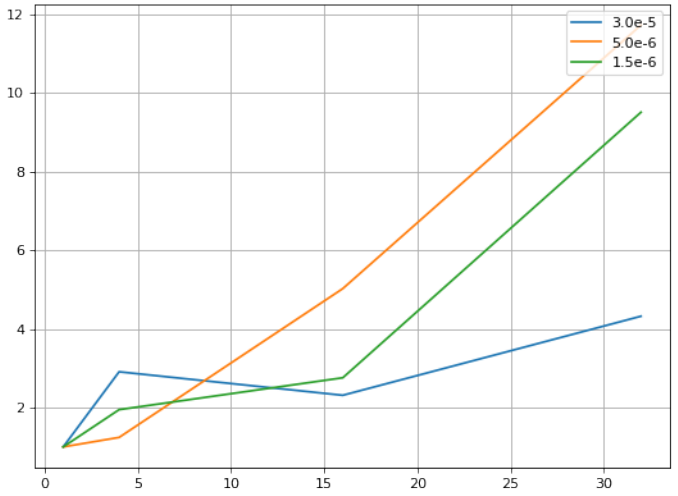
Также стоит отметить, что важную роль играет необходимость синхронизации между процессами (в том числе блокирующие ожидания). Если каждый из процессов будет во время одной итерации обрабатывать мало точек, то это не даст нивелировать те накладные расходы, которые возникают при их взаимодействии. Это ярко показано в примере 3.3.6.

Таким образом основным**и** параметром**и** системы являются $n\_dots\_per\_cycle$, и $n\_dots\_per\_proc$, поскольку именно они определяют баланс между объемом вычислений и объемом синхронизаций. Для проведения исследования были разработаны скрипты, которые позволяют получать график из пункта 3.2 за буквально несколько минут (если свободен кластер).

3.3.1 Пример 1. Ближайшая степень двойки к $\frac{1}{\varepsilon}$

```
int64_t n_dots_per_cycle =
    pow(2, (int) log2(1/eps));

int64_t n_dots_per_proc =
          n_dots_per_cycle;
if (num_procs > 1) {
    n_dots_per_proc /= num_procs;
    n_dots_per_proc *= 2;
}
```



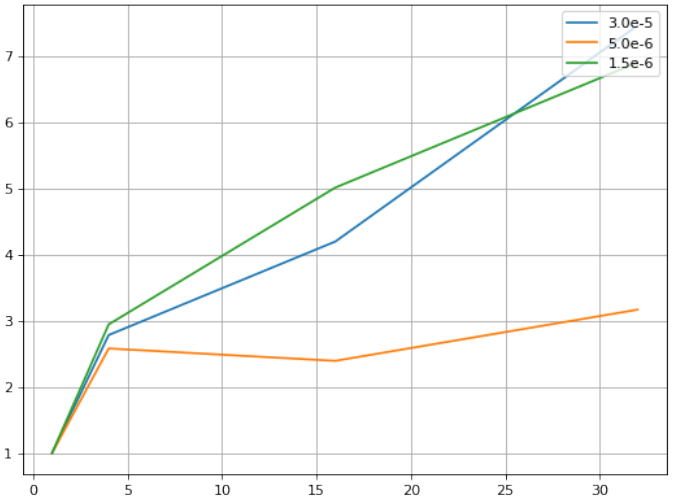
Важно, что при такой настройке программа в однопроцессороном режиме ОТРАБАТЫВАЕТ НЕСКОЛЬКО ЦИКЛОВ (от 1 до 20 в зависимости от seed и eps). Это значит что ядро не загружено излишними вычислениями, а решает задачу с неопределенностью с вполне комфортным шагом на каждом цикле генерации точек. Может повести и процедура сойдется за 1 цикл, а может не повести и сойдется только через 20 циклов. В среднем, 5-15 циклов необходимо при выбранных параметрах. Также хорошему результату способствует, то, что на самом деле, за каждой итерацию мы считаем $2 * n\_dots\_per\_proc$. Чуть сильнее загружая процессы мы можем эффективно компенсировать потери на синхронизацию.

3.3.2 Пример 2. Степень 10

Неожиданно, такой подбор количества точек явно опережает метод из примера 1 для количества процессов меньше 32, но в контексте задания важнее все же масштабируемость, поэтому данные по первому примеру представлены в качестве результатов.

```
int64_t n_dots_per_cycle = 10;
int pow = (int)log10(1/eps);
for (int i = 1; i <= pow; ++i) {
    n_dots_per_cycle *= 10;
}

int64_t n_dots_per_proc =
            n_dots_per_cycle;
if (num_procs > 1) {
    n_dots_per_proc /= num_procs;
    n_dots_per_proc *= 2;
}
```

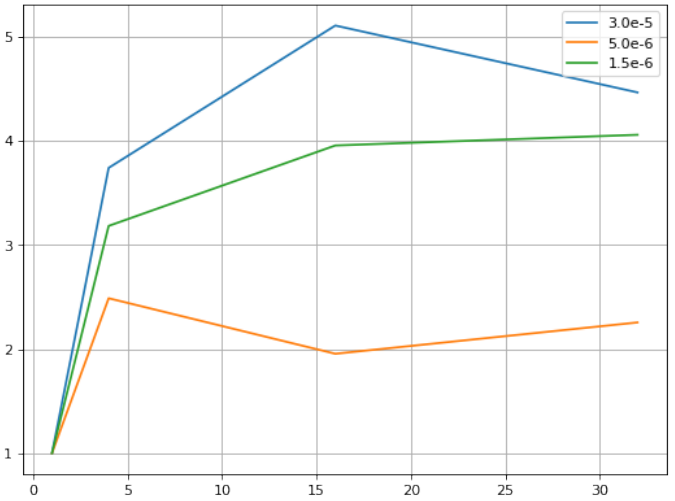


3.3.3 Пример 3. Другое распределение вычислений

Как мы можем видеть, таким параметры дают лучший баланс (и, соответственно, результат) для 4 процессоров. На 32 процессах явно видна недозагруженность ядер и что это может даже уменьшить производительность при большем количестве ресурсов.

```
int64_t n_dots_per_cycle = 10;
int pow = (int)log10(1/eps);
for (int i = 1; i <= pow; ++i) {
    n_dots_per_cycle *= 10;
}

if (num_procs == 4) {
    n_dots_per_proc =
      n_dots_per_cycle / 2;
} else if (num_procs == 16) {
    n_dots_per_proc =
      n_dots_per_cycle / 4;
} else if (num_procs == 32) {
    n_dots_per_proc =
      n_dots_per_cycle / 8;
}
```

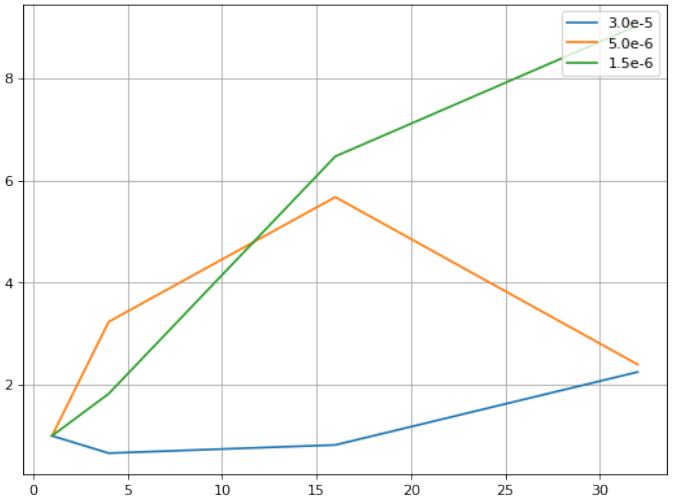


3.3.4 Пример 4. Невыровненное $\frac{1}{\varepsilon}$

Если мы берем все величины порядка $\frac{1}{\varepsilon}$, попробуем взять саму величину- вдруг она будет точнее попадать. Как видим, нет. Плохо ложащиеся на архитектуру кеша числа убивают всю масштабируемость.

```
int64_t n_dots_per_cycle =
          (int)(1/eps);

if (num_procs > 1) {
    n_dots_per_proc =
        n_dots_per_cycle
        / num_procs;
    n_dots_per_proc *= 2;
}
```

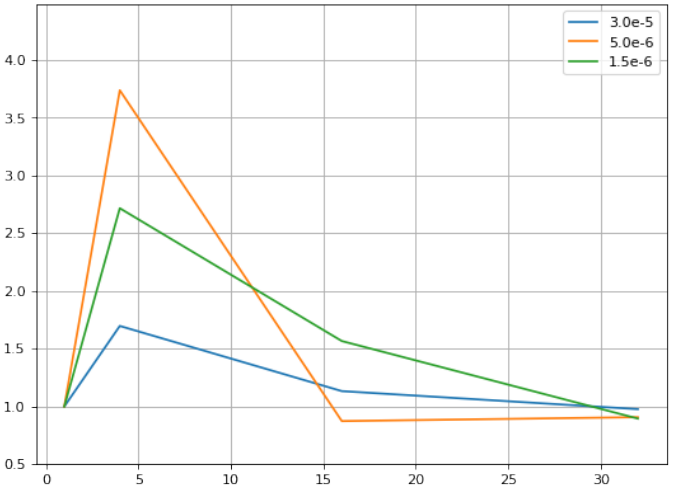


3.3.5 Пример 5. Слишком мало точек

Этот рисунок показывает, что у синхронизация накладывает накладные расходы, которые могут убить всякую масштабируемость, если не думать о балансе времени вычислений и коммуникации.

```
int64_t n_dots_per_cycle = 100;

int64_t n_dots_per_proc =
            n_dots_per_cycle;
```

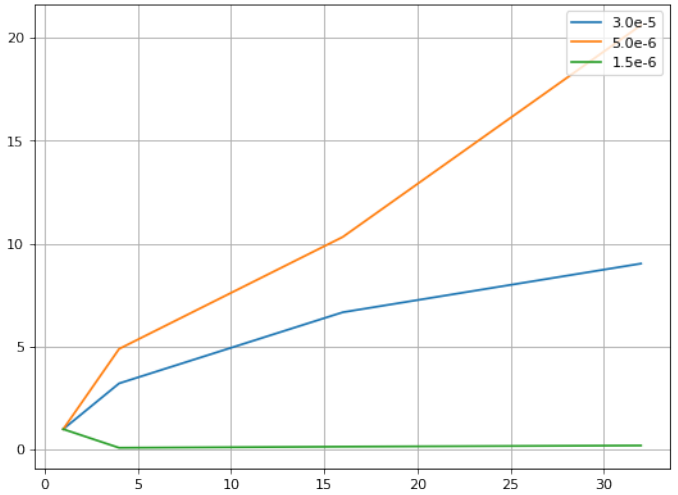


3.3.6 Пример 6. Слишком много точек

Как и было сказано выше, на всех точностях *num_iterations* почти всегда задача решается за 1 цикл. Скорость для $1.5e-6$ падает очевидно из-за колоссального объема вычислений порядка миллиарда(!) операций, который приходится проделывать на каждой итерации. Разница в 100 раз, на самом деле, может быть очень существенной: 100 раз по 10 и 100 раз по 1000- это совсем разные вещи.

```
int64_t n_dots_per_proc =
    (int)(1/sqrt(eps));

n_dots_per_proc =
    n_dots_per_proc
    * n_dots_per_proc
    * n_dots_per_proc
    / num_proc;
```



4 Выводы

Как мы видим, можно подогнать параметры так, что программа будет работать действительно ОЧЕНЬ быстро, хорошо масштабироваться, но это будет явное переобучение. Даже приведенные примеры проверены всего для 4 seed, и были усреднены с помощью медиан для устойчивости. Выбрать параметры системы для произвольных данных с сохранением такого же качества крайне сложно.