МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Кафедра программной инженерии

Направление подготовки: «Программная Инженерия»

ОТЧЕТ

по курсу «Параллельное программирование» на тему

«Вычисление многомерных интегралов методом Монте-Карло.»

Выполнил: Студент группы 381608,
Киселев Денис Сергеевич

Научный руководитель: Доцент кафедры МОиСТ, кандидат технических наук,
Сысоев Александр Владимирович

Нижний Новгород
2019 г.

Оглавление

Введение

Интеграл — одно из важнейших понятий математического анализа, которое возникает при решении задач о нахождении площади под кривой, пройденного пути при неравномерном движении, массы неоднородного тела, и тому подобных, а также в задаче о восстановлении функции по её производной (неопределённый интеграл). Упрощенно интеграл можно представить как аналог суммы для бесконечного числа бесконечно малых слагаемых. В зависимости от пространства, на котором задана подынтегральная функция, интеграл может быть — двойной, тройной, криволинейный, поверхностный и так далее.

Вычислить первообразные функции мы можем не всегда, но задача на дифференцирование может быть решена для любой функции. Именно поэтому единого метода интегрирования, который можно использовать для любых типов вычислений, не существует.

Методы Монте-Карло (ММК) — группа численных методов для изучения случайных процессов. Суть метода заключается в следующем: процесс моделируется при помощи генератора случайных величин. Это повторяется много раз, а потом на основе полученных случайных данных вычисляются вероятностные характеристики решаемой задачи. Например, чтобы узнать, какое в среднем будет расстояние между двумя случайными точками в круге, методом Монте-Карло, нужно взять много случайных пар точек, для каждой пары найти расстояние, а потом усреднить.

Используется для решения задач в различных областях физики, химии, математики, экономики, оптимизации, теории управления и др.

Постановка задачи

Задачей практической работы по курсу «Параллельное программирование» является разработка программы, реализующей последовательный и параллельный алгоритм вычисления многомерных интегралов методом Монте-Карло.

Цель данной работы:

1. Реализовать последовательную версию алгоритма.
2. Реализовать параллельную версию алгоритма с помощью технологии OpenMP.
3. Реализовать параллельную версию алгоритма с помощью технологии TBB.
4. Оценить эффективность и масштабируемость данной программы на кластере (или многопоточный запуск на персональном компьютере).
5. Сделать выводы.

Описание алгоритма интегрирования методом Монте-Карло

Для определения площади под графиком функции можно использовать следующий стохастический алгоритм:

- ограничим функцию прямоугольником (n-мерным параллелепипедом в случае многих измерений), площадь которого S_{par} можно легко вычислить; любая сторона прямоугольника содержит хотя бы 1 точку графика функции, но не пересекает его;
- «набросаем» в этот прямоугольник (параллелепипед) некоторое количество точек (N штук), координаты которых будем выбирать случайным образом;
- определим число точек (K штук), которые попадут под график функции;
- площадь области, ограниченной функцией и осями координат, S даётся выражением $S = S_{par} \cdot \frac{K}{N}$

Для малого числа измерений интегрируемой функции производительность Монте-Карло интегрирования гораздо ниже, чем производительность детерминированных методов. Тем не менее, в некоторых случаях, когда функция задана неявно, а необходимо определить область, заданную в виде сложных неравенств, стохастический метод может оказаться более предпочтительным.

Описание алгоритма решения

Последовательная версия

Алгоритм работы последовательной версии программы можно разделить на следующие этапы:

1. Определение интегрируемой функции и области интегрирования (n-мерный параллелепипед).
2. Генерация в области интегрирования нужного количества точек со случайными координатами.
3. Вычисление результата согласно формуле и вывод результатов.

На первом этапе мы определяем нашу n-мерную функцию, которую мы собираемся интегрировать. В нашей реализации это функция эллипсоида ($\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}=1,$ где a=1, b=2, c=3). Вместе с функцией мы определяем и область, по которой будем производить интегрирование. В нашем случае это: $\{ D : x \in [-1,\ 1],\ y \in [-2,\ 2],\ z \in [-3,\ 3] \}$.

```
double ellipsoid(double x, double y, double z) {
    return - 1.0 + pow(x / ELLPS_A, 2)
                 + pow(y / ELLPS_B, 2)
                 + pow(z / ELLPS_C, 2);
}
```

Функция эллипсоида

```
#define X1 -1.0
#define X2 1.0
#define Y1 -2.0
#define Y2 2.0
#define Z1 -3.0
#define Z2 3.0
```

Границы области интегрирования

Второй этап самый важный. Мы генерируем нужное количество точек (задаем заранее) со случайными координатами в нашу интегрируемую область.

```
int nHitsOnEllipsoid = 0;
for (int i = 0; i < nPoints; i++) {
    double value = ellipsoid(xAxisDistr(generator),
                             yAxisDistr(generator),
                             zAxisDistr(generator));
    if (value <= 0) nHitsOnEllipsoid++;
}
```

Генерация и подсчет попаданий случайных точек в эллипсоид

Подсчитав количество точек, попавших в эллипсоид, и определив на основании этого значения соответствующую частоту попадания, мы переходим к последнему этапу. Тут мы считаем объем нашего заранее задаваемого n-мерного параллелепипеда, который является интегрируемой областью. Умножив значение этого объема на ранее рассчитанную частоту попадания, мы получим наш результат.

Чтобы определить точность результата, воспользуемся формулой объема эллипсоида ($V = \frac{4}{3}\pi abc$) и подсчитаем действительное значение нашего n-мерного интеграла. Сравнив это значение со значением, которое мы получили ранее путем интегрирования эллипсоида методом Монте-Карло, мы можем получить определенное представление о точности нашего результата.

```
double avgValueOfHits = static_cast<double>(nHitsOnEllipsoid) / nPoints;
double res = (X2 - X1) * (Y2 - Y1) * (Z2 - Z1) * avgValueOfHits;
double realRes = 4.0 / 3.0 * std::acos(-1) * ELLPS_A * ELLPS_B * ELLPS_C;
```

Подсчет результата методом Монте-Карло и по формуле объема

Параллельная версия с использованием OpenMP

Алгоритм работы параллельной версии программы несколько отличается от последовательной:

1. Определение интегрируемой функции и области интегрирования (n-мерный параллелепипед).
2. Преждевременная генерация случайных точек по области интегрирования, и сохранение этих точек в массиве.
3. Подсчет количества точек, попадающих в эллипсоид.
4. Вычисление результата согласно формуле и вывод результатов.

Первый и последние этапы ничем не отличаются от этапов алгоритма последовательной версии.

На втором этапе мы преждевременно генерируем точки со случайными координатами в основном потоке и сохраняем эти точки в массиве. Эта преждевременность необходима нам для соблюдения потокобезопасности.

```
std::vector<std::vector<double> > generateRandomPointsByArea(
        int nPoints, const std::vector<std::pair<double, double> >& limits) {
    int nDimensions = limits.size();

    std::vector<std::uniform_real_distribution<> > distrs;
    distrs.reserve(nDimensions);
    for (int i = 0; i < nDimensions; i++) {
        distrs.emplace_back(limits[i].first, limits[i].second);
    }

    std::random_device r;
    std::default_random_engine generator(r());
    std::vector<std::vector<double> > points(nPoints, std::vector<double>(nDimensions));
    for (int i = 0; i < nPoints; i++) {
        for (int j = 0; j < nDimensions; j++) {
            points[i][j] = distrs[j](generator);
        }
    }

    return points;
}
```

Функция генерирования случайных точек по области

На третьем этапе мы определяем цикл, который будем распараллеливать. В этом цикле мы бежим по заранее сгенерированным точкам и фиксируем количество тех, что попадают в наш эллипсоид.

```
int nPointsInEllipsoid = 0;
#pragma omp parallel reduction(+: nPointsInEllipsoid)
{
    #pragma omp for schedule(guided)
    for (int i = 0; i < nPoints; i++) {
        double value = func(points[i]);
        if (value <= 0) nPointsInEllipsoid++;
    }
}
```

Основной цикл, который распараллеливается при помощи директив OpenMP

Параллельная версия с использованием TBB

Алгоритм работы параллельной версии с использованием библиотеки TBB ничем не отличается от OpenMP версии. Код отличается только распараллеливаемым участком.

```
int nPointsInEllipsoid = tbb::parallel_reduce(
    tbb::blocked_range<size_t>(0, nPoints), 0,
    [&] (const tbb::blocked_range<size_t>& r, int anotherValue) -> int {
        size_t begin = r.begin(), end = r.end();
        for (size_t i = begin; i != end; i++) {
            double value = func(points[i]);
            if (value <= 0) anotherValue++;
        }
        return anotherValue;
    },
    std::plus<int>());
```

Основной цикл, который распараллеливается при помощи библиотеки TBB

Результаты экспериментов

Для определения эффективности параллельных версий были проведены запуски программы на разном количестве потоков с разным количеством точек.

Программа запускалась 100 раз в каждой конфигурации для более точного определения результата.

Запуски проводились на персональном компьютере со следующими характеристиками центрального процессора:

Наименование: Intel® Core™ i5-7300HQ

Количество ядер: 4

Количество потоков: 4

Результаты запусков отображены на следующей таблице:

Количество потоков	1 поток	2 потока		4 потока		8 потоков		
Количество точек	Linear	OpenMP	TBB	OpenMP	TBB	OpenMP	TBB	Точность
4’000	0,00002185	0,00004602	0,00007955	0,00003638	0,00008226	0,00004992	0,00009533	0,99892618
40’000	0,00024628	0,00036442	0,00049601	0,0002018	0,00037579	0,00021783	0,00037759	0,99961079
400’000	0,00279343	0,00369738	0,00392442	0,00192089	0,00221053	0,00193943	0,00216749	0,99993380
4’000’000	0,0284323	0,03679737	0,03853111	0,01931232	0,0200349	0,0197299	0,01992644	0,99998147
40’000’000	0,28484502	0,36566282	0,38037792	0,19179319	0,19821047	0,19486848	0,19803679	0,99999165

Таблица результатов (время работы программы указано в секундах)

На основании этой таблицы построим график зависимости ускорения от количества точек параллельной версии программы на 4-х потоках:

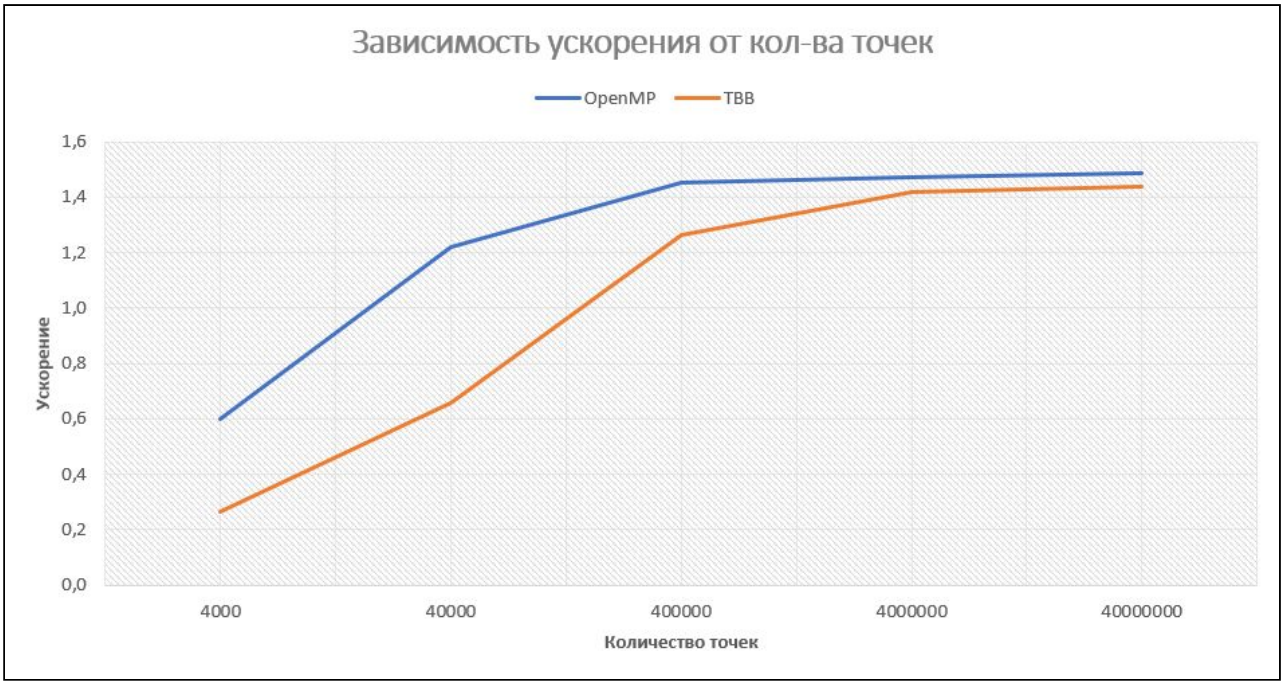

График зависимости ускорения от количества точек

На следующем графике отобразим зависимость точности вычислений от количества точек:

График зависимости точности вычислений от количества точек

Вывод

Рассмотрев таблицу результатов запусков, можно сделать вывод, что ускорение программы с помощью распараллеливания библиотекой TBB меньше, чем с OpenMP. Однако разница в ускорении стремительно сокращается с увеличением объемов вычислений. Это можно объяснить тем, что распараллеливаемый участок кода является очень коротким и простым. В таком случае время на организацию параллелизма играет более важную и значительную роль. Этим же объясняется и то, что ускорение параллельной версии программы на 4-х потоках упирается в отметку ~1,5, а на 2-х потоках ускорение и вовсе отсутствует.

Итог: сравнительно небольшие объемы вычислений, а также простота распараллеливаемого участка кода в совокупности приводят к тому, что ускорение программы и при использовании OpenMP, и TBB является не таким существенным, как этого бы хотелось.

Заключение

В ходе работы был успешно разработаны и реализованы последовательный и параллельный алгоритмы интегрирования многомерных интегралов методом Монте-Карло. Была проведена серия запусков программы на различном количестве потоков и точек. Эта серия позволила нам оценить ускорение параллельных версий алгоритма с использованием OpenMP и TBB.

Источники

1. Википедия (электронный ресурс)
   1.1. Метод Монте-Карло
   https://ru.wikipedia.org/wiki/Метод_Монте-Карло
   1.2. OpenMp
   https://ru.wikipedia.org/wiki/OpenMP
   1.3. Intel Threading Building Blocks
   https://ru.wikipedia.org/wiki/Intel_Threading_Building_Blocks
2. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.
3. Reinders, James (2007, July). Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism (Paperback) Sebastopol: O'Reilly Media, ISBN 978-0-596-51480-8.

Приложения

Приложение №1

```
// Copyright Kiselev Denis 2019

#include <cfloat>
#include <iostream>
#include <random>

#define DEFAULT_NPOINTS 10000

// Ellipsoid options
#define ELLPS_A 1.0
#define ELLPS_B 2.0
#define ELLPS_C 3.0

// Limits of integration
#define X1 -1.0
#define X2 1.0
#define Y1 -2.0
#define Y2 2.0
#define Z1 -3.0
#define Z2 3.0

// This is a simple elipsoid function, which we
// will integrate using the Monte Carlo method.
double ellipsoid(double x, double y, double z) {
    return - 1.0 + pow(x / ELLPS_A, 2)
                 + pow(y / ELLPS_B, 2)
                 + pow(z / ELLPS_C, 2);
}

int main(int argc, char *argv[]) {
    // Creating a generator and distributions
    // for random double numbers in the ranges
    std::random_device rd;
    std::default_random_engine generator(rd());
    std::uniform_real_distribution<> xAxisDistr(X1, X2);
    std::uniform_real_distribution<> yAxisDistr(Y1, Y2);
    std::uniform_real_distribution<> zAxisDistr(Z1, Z2);
```

```
    int nPoints = (argc == 1) ? DEFAULT_NPOINTS : atoi(argv[1]);

    // Ellipsoid hit counting
    int nHitsOnEllipsoid = 0;
    for (int i = 0; i < nPoints; i++) {
        double value = ellipsoid(xAxisDistr(generator),
                                 yAxisDistr(generator),
                                 zAxisDistr(generator));
        if (value <= 0) nHitsOnEllipsoid++;
    }

    // Calculation of the result according to the Monte Carlo method
    double avgValueOfHits = static_cast<double>(nHitsOnEllipsoid) / nPoints;
    double res = (X2 - X1) * (Y2 - Y1) * (Z2 - Z1) * avgValueOfHits;
    double realRes = 4.0 / 3.0 * std::acos(-1) * ELLPS_A * ELLPS_B * ELLPS_C;

    std::cout.precision(8);
    std::cout << "Result: " << std::fixed << res << std::endl
              << "Real result: " << std::fixed << realRes << std::endl
              << "Error: " << std::fixed << res - realRes << std::endl;
    return 0;
}
```

Приложение №2

```
// Copyright Kiselev Denis 2019

#include <omp.h>

#include <functional>
#include <iostream>
#include <random>
#include <utility>
#include <vector>

#define DEFAULT_NPOINTS 100000

// Ellipsoid options
#define ELLPS_A 1.0
#define ELLPS_B 2.0
#define ELLPS_C 3.0

// Limits of integration
#define X1 -1.0
#define X2 1.0
#define Y1 -2.0
#define Y2 2.0
#define Z1 -3.0
#define Z2 3.0

double ellipsoid(const std::vector<double>& args) {
    return -1.0 + pow(args[0] / ELLPS_A, 2)
                + pow(args[1] / ELLPS_B, 2)
                + pow(args[2] / ELLPS_C, 2);
}
```

```cpp
std::vector<std::vector<double> > generateRandomPointsByArea(
        int nPoints, const std::vector<std::pair<double, double> >& limits) {
    int nDimensions = limits.size();

    std::vector<std::uniform_real_distribution<> > distrs;
    distrs.reserve(nDimensions);
    for (int i = 0; i < nDimensions; i++) {
        distrs.emplace_back(limits[i].first, limits[i].second);
    }

    std::random_device r;
    std::default_random_engine generator(r());
    std::vector<std::vector<double> > points(nPoints, std::vector<double>(nDimensions));
    for (int i = 0; i < nPoints; i++) {
        for (int j = 0; j < nDimensions; j++) {
            points[i][j] = distrs[j](generator);
        }
    }

    return points;
}

double integrateByMonteCarlo(
        std::function<double(const std::vector<double>&)> func,
        const std::vector<std::pair<double, double> >& limits,
        const std::vector<std::vector<double> >& points) {
    int nDimensions = limits.size();
    int nPoints = points.size();

    int nPointsInEllipsoid = 0;
    for (int i = 0; i < nPoints; i++) {
        double value = func(points[i]);
        if (value <= 0) nPointsInEllipsoid++;
    }

    double measure = 1.0;
    for (int i = 0; i < nDimensions; i++) {
        measure *= limits[i].second - limits[i].first;
    }

    double hitProbability = static_cast<double>(nPointsInEllipsoid) / nPoints;
    return measure * hitProbability;
}
```

```cpp
double integrateByMonteCarloParallel(
        std::function<double(const std::vector<double>&)> func,
        const std::vector<std::pair<double, double> >& limits,
        const std::vector<std::vector<double> >& points) {
    int nDimensions = limits.size();
    int nPoints = points.size();

    int nPointsInEllipsoid = 0;
    #pragma omp parallel reduction(+: nPointsInEllipsoid)
    {
        #pragma omp for schedule(guided)
        for (int i = 0; i < nPoints; i++) {
            double value = func(points[i]);
            if (value <= 0) nPointsInEllipsoid++;
        }
    }
```

```
    double measure = 1.0;
    for (int i = 0; i < nDimensions; i++) {
        measure *= limits[i].second - limits[i].first;
    }

    double hitProbability = static_cast<double>(nPointsInEllipsoid) / nPoints;
    return measure * hitProbability;
}

int main(int argc, char *argv[]) {
    int nPoints = (argc > 1) ? atoi(argv[1]) : DEFAULT_NPOINTS;

    std::vector<std::pair<double, double> > limits = { { X1, X2 }, { Y1, Y2 }, { Z1, Z2 } };
    auto points = generateRandomPointsByArea(nPoints, limits);

    // Sequential
    double t1 = omp_get_wtime();
    double seqResult = integrateByMonteCarlo(ellipsoid, limits, points);
    double seqTime = omp_get_wtime() - t1;

    // Parallel
    t1 = omp_get_wtime();
    double parResult = integrateByMonteCarloParallel(ellipsoid, limits, points);
    double parTime = omp_get_wtime() - t1;

    double realRes = 4.0 / 3.0 * std::acos(-1) * ELLPS_A * ELLPS_B * ELLPS_C;
    double speedUp = seqTime / parTime;
    std::cout << "Sequential alg:\n"
                 "\tResult: " << seqResult << "\n"
                 "\tTime: " << seqTime << " sec\n"
                 "Parallel alg:\n"
                 "\tResult: " << parResult << "\n"
                 "\tTime: " << parTime << " sec\n"
                 "Real result: " << realRes << "\n"
                 "Speed up: " << speedUp << "\n";
    return 0;
}
```

Приложение №3

```
// Copyright Kiselev Denis 2019

#include <tbb/tbb.h>

#include <functional>
#include <iostream>
#include <random>
#include <utility>
#include <vector>

#define DEFAULT_NPOINTS 1000000

// Ellipsoid options
#define ELLPS_A 1.0
#define ELLPS_B 2.0
#define ELLPS_C 3.0
```

```cpp
// Limits of integration
#define X1 -1.0
#define X2 1.0
#define Y1 -2.0
#define Y2 2.0
#define Z1 -3.0
#define Z2 3.0

double ellipsoid(const std::vector<double>& args) {
    return -1.0 + pow(args[0] / ELLPS_A, 2)
                + pow(args[1] / ELLPS_B, 2)
                + pow(args[2] / ELLPS_C, 2);
}

std::vector<std::vector<double> > generateRandomPointsByArea(
        int nPoints, const std::vector<std::pair<double, double> >& limits) {
    int nDimensions = limits.size();

    std::vector<std::uniform_real_distribution<> > distrs;
    distrs.reserve(nDimensions);
    for (int i = 0; i < nDimensions; i++) {
        distrs.emplace_back(limits[i].first, limits[i].second);
    }

    std::random_device r;
    std::default_random_engine generator(r());
    std::vector<std::vector<double> > points(nPoints, std::vector<double>(nDimensions));
    for (int i = 0; i < nPoints; i++) {
        for (int j = 0; j < nDimensions; j++) {
            points[i][j] = distrs[j](generator);
        }
    }

    return points;
}

double integrateByMonteCarlo(
        std::function<double(const std::vector<double>&)> func,
        const std::vector<std::pair<double, double> >& limits,
        const std::vector<std::vector<double> >& points) {
    int nDimensions = limits.size();
    int nPoints = points.size();

    int nPointsInEllipsoid = 0;
    for (int i = 0; i < nPoints; i++) {
        double value = func(points[i]);
        if (value <= 0) nPointsInEllipsoid++;
    }

    double measure = 1.0;
    for (int i = 0; i < nDimensions; i++) {
        measure *= limits[i].second - limits[i].first;
    }

    double hitProbability = static_cast<double>(nPointsInEllipsoid) / nPoints;
    return measure * hitProbability;
}
```

```
double integrateByMonteCarloParallel(
        std::function<double(const std::vector<double>&)> func,
        const std::vector<std::pair<double, double> >& limits,
        const std::vector<std::vector<double> >& points) {
    int nDimensions = limits.size();
    int nPoints = points.size();

    int nPointsInEllipsoid = tbb::parallel_reduce(
        tbb::blocked_range<size_t>(0, nPoints), 0,
        [&] (const tbb::blocked_range<size_t>& r, int anotherValue) -> int {
            size_t begin = r.begin(), end = r.end();
            for (size_t i = begin; i != end; i++) {
                double value = func(points[i]);
                if (value <= 0) anotherValue++;
            }
            return anotherValue;
        },
        std::plus<int>());

    double measure = 1.0;
    for (int i = 0; i < nDimensions; i++) {
        measure *= limits[i].second - limits[i].first;
    }

    double hitProbability = static_cast<double>(nPointsInEllipsoid) / nPoints;
    return measure * hitProbability;
}

int main(int argc, char *argv[]) {
    int nPoints = (argc > 1) ? atoi(argv[1]) : DEFAULT_NPOINTS;

    std::vector<std::pair<double, double> > limits = { { X1, X2 }, { Y1, Y2 }, { Z1, Z2 } };
    auto points = generateRandomPointsByArea(nPoints, limits);

    // Sequential
    tbb::tick_count t1 = tbb::tick_count::now();
    double seqResult = integrateByMonteCarlo(ellipsoid, limits, points);
    double seqTime = (tbb::tick_count::now() - t1).seconds();

    // Parallel
    t1 = tbb::tick_count::now();
    double parResult = integrateByMonteCarloParallel(ellipsoid, limits, points);
    double parTime = (tbb::tick_count::now() - t1).seconds();

    double realRes = 4.0 / 3.0 * std::acos(-1) * ELLPS_A * ELLPS_B * ELLPS_C;
    double speedUp = seqTime / parTime;
    std::cout << "Sequential alg:\n"
                 "\tResult: " << seqResult << "\n"
                 "\tTime: " << seqTime << " sec\n"
                 "Parallel alg:\n"
                 "\tResult: " << parResult << "\n"
                 "\tTime: " << parTime << " sec\n"
                 "Real result: " << realRes << "\n"
                 "Speed up: " << speedUp << "\n";
    return 0;
}
```