Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

«Вычисление многомерных интегралов с использованием многошаговой схемы. Метод Симпсона»

Выполнил:
студент группы 381806-1
Власов М. С.

Проверил:
доцент кафедры МОСТ,
кандидат технических наук
Сысоев А. В.

Нижний Новгород
2021

Оглавление

Введение

Как правило, как людям, так и компьютерам удобнее работать с такими наборами или списками данных, которые упорядочены по определенному свойству. Одним из способов упорядочения множества данных является сортировка.

Но обычно сортировки представляют собой алгоритмы, требующие многократных обходов сортируемого диапазона. Отсюда следует, что чем больше количество объектов, которые необходимо отсортировать, тем больше времени требуется для сортировки. Уменьшить затраты времени на сортировку может помочь распараллеливание реализации алгоритма с помощью его запуска на нескольких процессорах вычислительного оборудования.

Таким образом, мы можем разделить исходный набор данных на части, для каждой вычислительной единицы поровну, отсортировать их параллельно, а затем объединить, чтобы получить набор с начальным размером. Эффективным подходом для такого слияния в данном случае может служить применение алгоритма четно-нечетного слияния Бэтчера.

Итак, в данной лабораторной работе будет реализован один из алгоритмов сортировок - сортировка Шелла, а также алгоритм четно-нечетного слияния Бэтчера.

Постановка задачи

В рамках лабораторной работы необходимо реализовать последовательную и параллельную реализации сортировки Шелла для целых чисел с четно-нечетным слиянием Бэтчера, проверить корректность работы алгоритмов, провести эксперименты для оценки эффективности параллелизации. По полученным результатам сделать выводы.

Для реализации параллельной версии необходимо использовать библиотеку межпроцессорного взаимодействия MPI. Для проверки корректности работы алгоритмов используется Google Testing Framework.

Описание алгоритмов

Формула Симпсона относится к приёмам численного интегрирования. Суть классического метода заключается в приближении подынтегральной функции $f(x)$ на отрезке $[a, b]$ интерполяционным многочленом второй степени $p_2(x)$, то есть приближение графика функции на отрезке параболой. Метод Симпсона имеет порядок погрешности 4 и алгебраический порядок точности 3.

Формулой Симпсона называется интеграл от интерполяционного многочлена второй степени на отрезке $[a, b]$:

$$\int_a^b f(x)dx \approx \int_a^b p_2(x)dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a-b}{2}\right) + f(b)\right)$$

Для более точного вычисления интеграла, интервал $[a, b]$ разбивают на $N = 2n$ элементарных отрезков одинаковой длины и применяют формулу Симпсона на составных отрезках. Каждый составной отрезок состоит из соседней пары элементарных отрезков. Значение исходного интеграла является суммой результатов интегрирования на составных отрезках:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(x_0) + 2\sum_{j=1}^{\frac{N}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{N}{2}} f(x_{2j-1}) + f(x_N)\right),$$

где $h = \frac{b-a}{N}$ - величина шага, а $x_j = a + jh$ - чередующиеся границы и середины составных отрезков, на которых применяется формула Симпсона.

Эта более точная формула также называется формулой Котеса.

Для многомерных интегралов формула имеет следующее очевидное обобщение:

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2}...\int_{a_n}^{b_n} f(X)dx_1dx_2...dx_n \approx$$

$$\approx \frac{h_1h_2...h_n}{3N}\left(f(X_0) + 2\sum_{j=1}^{\frac{N}{2}-1} f(X_{2j}) + 4\sum_{j=1}^{\frac{N}{2}} f(X_{2j-1}) + f(X_N)\right),$$

где N - число шагов, $X = (x_1, x_2, ..., x_n)$ - вектор переменных-аргументов функции, $h_1 = |a_1 - b_1|, h_2 = |a_2 - b_2|, ..., h_n = |a_n - b_n|$ - величины шага.

Схема распараллеливания

Для распараллеливания алгоритма вычисления многомерного интеграла с помощью формулы Симпсона необходимо распределить итерации цикла, выполняющего пошаговое вычисление, между доступными потоками, таким образом, чтобы на каждый получилось одинаковое число итераций. При этом результаты вычислений от каждого потока необходимо сложить.

Если случится так, что число шагов (общее число итераций) не делится нацело на число доступных потоков, необходимо досчитать неучтенные итерации после параллельного выполнения одним из потоков и так же прибавить к общему результату.

Описание программной реализации

Алгоритм последовательного вычисления многомерного интеграла с использованием многошагового метода Симпсона вызывается с помощью функции:

```
double SimpsonMethod::sequential(
    const std::function<double(const std::vector<double>&)>& func,
    const std::vector<double>& seg_begin,
    const std::vector<double>& seg_end,
    int steps_count);
```

Входными параметрами являются: функция, которую необходимо проинтегрировать (в виде функтора, указателя на функцию или другого функционального объекта, приводимого к типу std::function), векторы значений - начал и концов отрезков для каждого из параметров интегрируемой функции, а также число шагов. Выходными данными является непосредственно значение интеграла.

Реализация распараллеливания метода Симпсона представлена в функции:

```
double SimpsonMethod::parallel(
    const std::function<double(const std::vector<double>&)>& func,
    const std::vector<double>& seg_begin,
    const std::vector<double>& seg_end,
    int steps_count);
```

Она имеет те же входные параметры и выходные данные, что и предыдущая.

Эта функция, в зависимости от используемой библиотеки, использует для параллелизации средства либо OpenMP, либо oneTBB.

Подтверждение корректности

Для подтверждения корректности в программе представлен набор тестов, разработанных с Google Testing Framework.

Набор представляет из себя тесты, которые проверяют корректность вычислений (сравнивается значение интеграла, полученное благодаря параллельному вычслению, со значением, полученным с помощью последовательной реализации), а также эффективность (вычисление занимаемого последовательной и параллельной реализациями времени и сравнение полученных данных).

Успешное прохождение всех тестов подтверждает корректность работы написанной программы.

Результаты экспериментов

Эксперименты для оценки эффективности проводились на ПК со следующими характеристиками:

- Процессор: AMD A6, 2.60 GHz, 2 ядра;
- Оперативная память: 8 ГБ;
- ОС: Microsoft Windows 10 Pro.

Для проведения экспериментов производилось вычисление многомерного интеграла функции $x^2 + y^2 = z$ с помощью метода Симпсона - последовательно и параллельно - с количеством шагов, равным 10^7.

Таблица 1: Результаты экспериментов. Сравнение с OpenMP

Число потоков	Последовательно	Параллельно	Ускорение
1	2,935	3,298	0,899
2	3,312	1,826	1,814
3	2,930	1,742	1,682
4	2,784	1,701	1,637

Таблица 2: Результаты экспериментов. Сравнение с oneTBB

Число потоков	Последовательно	Параллельно	Ускорение
1	3,075	3,079	0,999
2	2,903	1,573	1,846
3	2,847	1,646	1,729
4	2,834	1,658	1,709

По данным, полученным в результате экспериментов, можно сделать вывод о том, что параллельный случай работает действительно быстрее, чем последовательный.

Если увеличить количество выделяемых потоков до 4, возможно получить деградацию ускорения. Так происходит из-за увеличения накладных расходов на менеджмент ресурсов (переключение между потоками, расчет частичных данных итераций). При дальнейшем увеличении количества потоков ускорения при данном количестве процессорных ядер компьютера, очевидно, не будет.

Заключение

В ходе выполнения лабораторной работы были разработаны последовательная и параллельная реализации алгоритма вычисления многомерных интегралов с помощью многошаговой схемы по формуле Симпсона.

Задача работы была успешно достигнута, поскольку результаты проведенных экспериментов по оценке эффективности показывают, что параллельная реализация работает быстрее, чем последовательная.

Кроме того, были написаны тесты с использованием Google Testing Framework, необходимые для подтверждения корректности работы программы.

Литература

1. Википедия: Формула Симпсона [Электронный ресурс] // URL: `https://en.wikipedia.org/wiki/Simpson's_rule` (дата обращения: 08.05.2021)

2. Studwood: Формула Симпсона для вычисления двойных интегралов [Электронный ресурс] // URL: `https://studwood.ru/1710068/matematika_himiya_fizika/formula_simpsona_vychisleniya_dvoynyh_integralov` (дата обращения: 08.05.2021)

Приложение

В этом разделе находится листинг всего кода, написанного в рамках лабораторной работы.

1. Библиотека с последовательной реализацией. Файл: simpson_method.h

```
// Copyright 2021 Vlasov Maksim

#ifndef MODULES_TASK_1_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_
#define MODULES_TASK_1_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_

#include <functional>
#include <vector>

namespace SimpsonMethod {

double integrate(const std::function<double(const std::vector<double>&)>& func,
                 const std::vector<double>& seg_begin,
                 const std::vector<double>& seg_end, int steps_count);

}  // namespace SimpsonMethod

#endif  // MODULES_TASK_1_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_
```

2. Библиотека с последовательной реализацией. Файл: simpson_method.cpp

```
// Copyright 2021 Vlasov Maksim

#include "../../../modules/task_1/vlasov_m_simpson_method/simpson_method.h"

#include <algorithm>
#include <cassert>
#include <cmath>
#include <numeric>
#include <stdexcept>
#include <utility>

static void sumUp(std::vector<double>* accum, const std::vector<double>& add) {
    assert(accum->size() == add.size());
    for (size_t i = 0; i < accum->size(); i++)
        accum->at(i) += add[i];
}

double SimpsonMethod::integrate(
    const std::function<double(const std::vector<double>&)>& func,
    const std::vector<double>& seg_begin, const std::vector<double>& seg_end,
    int steps_count) {
    if (steps_count <= 0)
        throw std::runtime_error("Steps count must be positive");
    if (seg_begin.empty() || seg_end.empty())
        throw std::runtime_error("No segments");
```

```
    if (seg_begin.size() != seg_end.size())
        throw std::runtime_error("Invalid segments");
    size_t dim = seg_begin.size();
    std::vector<double> steps(dim), segments(dim);
    for (size_t i = 0; i < dim; i++) {
        steps[i] = (seg_end[i] - seg_begin[i]) / steps_count;
        segments[i] = seg_end[i] - seg_begin[i];
    }
    std::pair<double, double> sum = std::make_pair(0.0, 0.0);
    std::vector<double> args = seg_begin;
    for (int i = 0; i < steps_count; i += 2) {
        sumUp(&args, steps);
        sum.first += func(args);
        sumUp(&args, steps);
        sum.second += func(args);
    }
    double seg_prod = std::accumulate(segments.begin(), segments.end(), 1.0,
                                      [](double p, double s) { return p * s; });
    return (func(seg_begin) + 4 * sum.first + 2 * sum.second - func(seg_end)) *
           seg_prod / (3.0 * steps_count);
}
```

3. Библиотека с последовательной реализацией. Файл: main.cpp

```
// Copyright 2021 Vlasov Maksim

#include <gtest/gtest.h>

#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>

#include "./simpson_method.h"

#define MULTIDIM_FUNC(FNAME, FVARCOUNT, FCOMP)                                 \
    double FNAME(const std::vector<double>& x) {                               \
        assert(x.size() == (FVARCOUNT));                                       \
        return (FCOMP);                                                        \
    }

MULTIDIM_FUNC(generic, 1, 0);
MULTIDIM_FUNC(parabola, 1, -x[0] * x[0] + 4);
MULTIDIM_FUNC(body, 2, x[0] * x[0] + x[1] * x[1]);
MULTIDIM_FUNC(super, 3, std::sin(x[0] + 3) - std::log(x[1]) + x[2] * x[2]);

TEST(Sequential_SimpsonMethodTest, can_integrate_2d_function) {
    std::vector<double> seg_begin = {0};
    std::vector<double> seg_end = {2};
    double square = SimpsonMethod::integrate(parabola, seg_begin, seg_end, 100);
    ASSERT_NEAR(16.0 / 3.0, square, 1e-6);
}

TEST(Sequential_SimpsonMethodTest, can_integrate_3d_function) {
    std::vector<double> seg_begin = {0, 0};
    std::vector<double> seg_end = {1, 1};
```

```
    double volume = SimpsonMethod::integrate(body, seg_begin, seg_end, 100);
    ASSERT_NEAR(2.0 / 3.0, volume, 1e-6);
}

// Calculated by WolframAlpha with the following query:
// integrate (sin(x + 3) - ln(y) + z^2), x=[-2, 1], y=[1, 3], z=[0, 2]
TEST(Sequential_SimpsonMethodTest, can_integrate_super_function) {
    std::vector<double> seg_begin = {-2, 1, 0};
    std::vector<double> seg_end = {1, 3, 2};
    double integral = SimpsonMethod::integrate(super, seg_begin, seg_end, 100);
    ASSERT_NEAR(13.0007625, integral, 1e-6);
}

TEST(Sequential_SimpsonMethodTest, cannot_accept_empty_segment_vectors) {
    ASSERT_ANY_THROW(SimpsonMethod::integrate(generic, {}, {}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::integrate(generic, {0}, {}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::integrate(generic, {}, {0}, 100));
}

TEST(Sequential_SimpsonMethodTest, cannot_accept_invalid_segment_vectors) {
    ASSERT_ANY_THROW(SimpsonMethod::integrate(generic, {1, 2}, {1, 2, 3}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::integrate(generic, {1, 2, 3}, {1, 2}, 100));
}

TEST(Sequential_SimpsonMethodTest, cannot_accept_invalid_steps_count) {
    ASSERT_ANY_THROW(SimpsonMethod::integrate(generic, {0}, {0}, 0));
    ASSERT_ANY_THROW(SimpsonMethod::integrate(generic, {0}, {0}, -1));
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

4. Библиотека с реализацией на OpenMP. Файл: simpson_method.h

```
// Copyright 2021 Vlasov Maksim

#ifndef MODULES_TASK_2_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_
#define MODULES_TASK_2_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_

#include <functional>
#include <vector>

namespace SimpsonMethod {

double sequential(const std::function<double(const std::vector<double>&)>& func,
                  const std::vector<double>& seg_begin,
                  const std::vector<double>& seg_end, int steps_count);

double parallel(const std::function<double(const std::vector<double>&)>& func,
                std::vector<double> seg_begin, std::vector<double> seg_end,
                int steps_count);

}  // namespace SimpsonMethod
```

```
#endif  // MODULES_TASK_2_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_
```

5. Библиотека с реализацией на OpenMP. Файл: simpson_method.cpp

```
// Copyright 2021 Vlasov Maksim

#include "../../../modules/task_2/vlasov_m_simpson_method/simpson_method.h"

#include <omp.h>

#include <algorithm>
#include <cassert>
#include <cmath>
#include <iostream>
#include <numeric>
#include <stdexcept>
#include <utility>

static void sumUp(std::vector<double>* accum, const std::vector<double>& add) {
    assert(accum->size() == add.size());
    for (size_t i = 0; i < accum->size(); i++)
        accum->at(i) += add[i];
}

double SimpsonMethod::sequential(
    const std::function<double(const std::vector<double>&)>& func,
    const std::vector<double>& seg_begin, const std::vector<double>& seg_end,
    int steps_count) {
    if (steps_count <= 0)
        throw std::runtime_error("Steps count must be positive");
    if (seg_begin.empty() || seg_end.empty())
        throw std::runtime_error("No segments");
    if (seg_begin.size() != seg_end.size())
        throw std::runtime_error("Invalid segments");
    size_t dim = seg_begin.size();
    std::vector<double> steps(dim), segments(dim);
    for (size_t i = 0; i < dim; i++) {
        steps[i] = (seg_end[i] - seg_begin[i]) / steps_count;
        segments[i] = seg_end[i] - seg_begin[i];
    }
    std::pair<double, double> sum = std::make_pair(0.0, 0.0);
    std::vector<double> args = seg_begin;
    for (int i = 0; i < steps_count; i++) {
        sumUp(&args, steps);
        if (i % 2 == 0)
            sum.first += func(args);
        else
            sum.second += func(args);
    }
    double seg_prod = std::accumulate(segments.begin(), segments.end(), 1.0,
                                      [](double p, double s) { return p * s; });
    return (func(seg_begin) + 4 * sum.first + 2 * sum.second - func(seg_end)) *
           seg_prod / (3.0 * steps_count);
}

double SimpsonMethod::parallel(
```

```
    const std::function<double(const std::vector<double>&)>& func,
    std::vector<double> seg_begin, std::vector<double> seg_end,
    int steps_count) {
    if (steps_count <= 0)
        throw std::runtime_error("Steps count must be positive");
    if (seg_begin.empty() || seg_end.empty())
        throw std::runtime_error("No segments");
    if (seg_begin.size() != seg_end.size())
        throw std::runtime_error("Invalid segments");
    size_t dim = seg_begin.size();
    std::vector<double> steps(dim), segments(dim);
    for (size_t i = 0; i < dim; i++) {
        steps[i] = (seg_end[i] - seg_begin[i]) / steps_count;
        segments[i] = seg_end[i] - seg_begin[i];
    }
    double sum_first = 0, sum_second = 0;
    std::vector<double> args(dim);
    int t_count = 0, t_steps = 0;
#pragma omp parallel firstprivate(args) reduction(+ : sum_first, sum_second)
    {
        int t_id = omp_get_thread_num();
        t_count = omp_get_num_threads();
        t_steps = steps_count / t_count;
        for (size_t i = 0; i < dim; i++)
            args[i] = seg_begin[i] + steps[i] * t_id * t_steps;
        int t_start = t_id * t_steps;
        int t_end = t_start + t_steps;
        for (int i = t_start; i < t_end; i++) {
            sumUp(&args, steps);
            if (i % 2 == 0)
                sum_first += func(args);
            else
                sum_second += func(args);
        }
    }
    if (steps_count % t_count != 0) {
        int passed_steps_count = t_count * t_steps;
        for (size_t i = 0; i < dim; i++)
            args[i] = seg_begin[i] + steps[i] * passed_steps_count;
        for (int i = passed_steps_count; i < steps_count; i++) {
            sumUp(&args, steps);
            if (i % 2 == 0)
                sum_first += func(args);
            else
                sum_second += func(args);
        }
    }
    double seg_prod = std::accumulate(segments.begin(), segments.end(), 1.0,
                                      [](double p, double s) { return p * s; });
    return (func(seg_begin) + 4 * sum_first + 2 * sum_second - func(seg_end)) *
           seg_prod / (3.0 * steps_count);
}
```

6. Библиотека с реализацией на OpenMP. Файл: main.cpp

```
// Copyright 2021 Vlasov Maksim
```

```cpp
#include <gtest/gtest.h>
#include <omp.h>

#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>

#include "./simpson_method.h"

#define MULTIDIM_FUNC(FNAME, FVARCOUNT, FCOMP)                                 \
    double FNAME(const std::vector<double>& x) {                               \
        assert(x.size() == (FVARCOUNT));                                       \
        return (FCOMP);                                                        \
    }

MULTIDIM_FUNC(generic, 1, 0);
MULTIDIM_FUNC(parabola, 1, -x[0] * x[0] + 4);
MULTIDIM_FUNC(body, 2, x[0] * x[0] + x[1] * x[1]);
MULTIDIM_FUNC(super, 3, std::sin(x[0] + 3) - std::log(x[1]) + x[2] * x[2]);

// Performance test - for demo purposes, not for CI
/*TEST(Parallel_SimpsonMethodTest, same_result_as_sequential) {
    std::vector<double> seg_begin = {0, 0};
    std::vector<double> seg_end = {1, 1};
    std::pair<double, double> time = {0, 0};
    int num_threads;
    std::cout << "num_threads: ";
    std::cin >> num_threads;
    omp_set_num_threads(num_threads);
    time.first = omp_get_wtime();
    double seq = SimpsonMethod::sequential(body, seg_begin, seg_end, 10000000);
    time.second = omp_get_wtime();
    std::cout << "Sequential " << (time.second - time.first) << ' ' << seq
              << std::endl;
    time.first = omp_get_wtime();
    double par = SimpsonMethod::parallel(body, seg_begin, seg_end, 10000000);
    time.second = omp_get_wtime();
    std::cout << "Parallel " << (time.second - time.first) << ' ' << par
              << std::endl;
    ASSERT_NEAR(seq, par, 1e-6);
}*/

TEST(Parallel_SimpsonMethodTest, can_integrate_2d_function) {
    std::vector<double> seg_begin = {0};
    std::vector<double> seg_end = {2};
    double square = SimpsonMethod::parallel(parabola, seg_begin, seg_end, 100);
    ASSERT_NEAR(16.0 / 3.0, square, 1e-6);
}

TEST(Parallel_SimpsonMethodTest, can_integrate_3d_function) {
    std::vector<double> seg_begin = {0, 0};
    std::vector<double> seg_end = {1, 1};
    double volume = SimpsonMethod::parallel(body, seg_begin, seg_end, 100);
    ASSERT_NEAR(2.0 / 3.0, volume, 1e-6);
}
```

```
// Calculated by WolframAlpha with the following query:
// integrate (sin(x + 3) - ln(y) + z^2), x=[-2, 1], y=[1, 3], z=[0, 2]
TEST(Parallel_SimpsonMethodTest, can_integrate_super_function) {
    std::vector<double> seg_begin = {-2, 1, 0};
    std::vector<double> seg_end = {1, 3, 2};
    double integral = SimpsonMethod::parallel(super, seg_begin, seg_end, 100);
    ASSERT_NEAR(13.0007625, integral, 1e-6);
}

TEST(Parallel_SimpsonMethodTest, cannot_accept_empty_segment_vectors) {
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {}, {}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {0}, {}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {}, {0}, 100));
}

TEST(Parallel_SimpsonMethodTest, cannot_accept_invalid_segment_vectors) {
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {1, 2}, {1, 2, 3}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {1, 2, 3}, {1, 2}, 100));
}

TEST(Parallel_SimpsonMethodTest, cannot_accept_invalid_steps_count) {
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {0}, {0}, 0));
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {0}, {0}, -1));
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

7. Библиотека с реализацией на oneTBB. Файл: simpson_method.h

```
// Copyright 2021 Vlasov Maksim

#ifndef MODULES_TASK_3_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_
#define MODULES_TASK_3_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_

#include <functional>
#include <vector>

namespace SimpsonMethod {

double sequential(const std::function<double(const std::vector<double>&)>& func,
                  const std::vector<double>& seg_begin,
                  const std::vector<double>& seg_end, int steps_count);

double parallel(const std::function<double(const std::vector<double>&)>& func,
                std::vector<double> seg_begin, std::vector<double> seg_end,
                int steps_count);

}  // namespace SimpsonMethod

#endif  // MODULES_TASK_3_VLASOV_M_SIMPSON_METHOD_SIMPSON_METHOD_H_
```

8. Библиотека с реализацией на oneTBB. Файл: simpson_method.cpp

```cpp
// Copyright 2021 Vlasov Maksim

#include "../../../modules/task_3/vlasov_m_simpson_method/simpson_method.h"

#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>

#include <algorithm>
#include <cassert>
#include <cmath>
#include <numeric>
#include <stdexcept>
#include <utility>

static void sumUp(std::vector<double>* accum, const std::vector<double>& add) {
    assert(accum->size() == add.size());
    for (size_t i = 0; i < accum->size(); i++)
        accum->at(i) += add[i];
}

double SimpsonMethod::sequential(
    const std::function<double(const std::vector<double>&)>& func,
    const std::vector<double>& seg_begin, const std::vector<double>& seg_end,
    int steps_count) {
    if (steps_count <= 0)
        throw std::runtime_error("Steps count must be positive");
    if (seg_begin.empty() || seg_end.empty())
        throw std::runtime_error("No segments");
    if (seg_begin.size() != seg_end.size())
        throw std::runtime_error("Invalid segments");
    size_t dim = seg_begin.size();
    std::vector<double> steps(dim), segments(dim);
    for (size_t i = 0; i < dim; i++) {
        steps[i] = (seg_end[i] - seg_begin[i]) / steps_count;
        segments[i] = seg_end[i] - seg_begin[i];
    }
    std::pair<double, double> sum = std::make_pair(0.0, 0.0);
    std::vector<double> args = seg_begin;
    for (int i = 0; i < steps_count; i++) {
        sumUp(&args, steps);
        if (i % 2 == 0)
            sum.first += func(args);
        else
            sum.second += func(args);
    }
    double seg_prod = std::accumulate(segments.begin(), segments.end(), 1.0,
                                      [](double p, double s) { return p * s; });
    return (func(seg_begin) + 4 * sum.first + 2 * sum.second - func(seg_end)) *
           seg_prod / (3.0 * steps_count);
}

double SimpsonMethod::parallel(
    const std::function<double(const std::vector<double>&)>& func,
    std::vector<double> seg_begin, std::vector<double> seg_end,
    int steps_count) {
    if (steps_count <= 0)
        throw std::runtime_error("Steps count must be positive");
```

```
    if (seg_begin.empty() || seg_end.empty())
        throw std::runtime_error("No segments");
    if (seg_begin.size() != seg_end.size())
        throw std::runtime_error("Invalid segments");
    size_t dim = seg_begin.size();
    std::vector<double> steps(dim), segments(dim);
    for (size_t i = 0; i < dim; i++) {
        steps[i] = (seg_end[i] - seg_begin[i]) / steps_count;
        segments[i] = seg_end[i] - seg_begin[i];
    }
    std::pair<double, double> sum = std::make_pair(0.0, 0.0);
    sum = tbb::parallel_reduce(
        tbb::blocked_range<int>(0, steps_count), std::make_pair(0.0, 0.0),
        [&func, &steps, &seg_begin, &dim](const tbb::blocked_range<int>& range,
                                          std::pair<double, double> sum) {
            int t_begin = range.begin();
            int t_end = range.end();
            std::vector<double> args(dim);
            for (size_t i = 0; i < dim; i++)
                args[i] = seg_begin[i] + steps[i] * t_begin;
            for (int i = t_begin; i < t_end; i++) {
                sumUp(&args, steps);
                if (i % 2 == 0)
                    sum.first += func(args);
                else
                    sum.second += func(args);
            }
            return sum;
        },
        [](const std::pair<double, double>& lhs,
           const std::pair<double, double>& rhs) {
            return std::make_pair(lhs.first + rhs.first,
                                  lhs.second + rhs.second);
        });
    double seg_prod = std::accumulate(segments.begin(), segments.end(), 1.0,
                                      [](double p, double s) { return p * s; });
    return (func(seg_begin) + 4 * sum.first + 2 * sum.second - func(seg_end)) *
           seg_prod / (3.0 * steps_count);
}
```

9. Библиотека с реализацией на oneTBB. Файл: main.cpp

```
// Copyright 2021 Vlasov Maksim

#include <gtest/gtest.h>
#include <tbb/tick_count.h>

#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>

#include "./simpson_method.h"

#define MULTIDIM_FUNC(FNAME, FVARCOUNT, FCOMP)                                 \
    double FNAME(const std::vector<double>& x) {                               \
```

```
        assert(x.size() == (FVARCOUNT));                                    \
        return (FCOMP);                                                     \
    }

MULTIDIM_FUNC(generic, 1, 0);
MULTIDIM_FUNC(parabola, 1, -x[0] * x[0] + 4);
MULTIDIM_FUNC(body, 2, x[0] * x[0] + x[1] * x[1]);
MULTIDIM_FUNC(super, 3, std::sin(x[0] + 3) - std::log(x[1]) + x[2] * x[2]);

// Performance test - for demo purposes, not for CI
/*TEST(TBB_SimpsonMethodTest, same_result_as_sequential) {
    std::vector<double> seg_begin = {0, 0};
    std::vector<double> seg_end = {1, 1};
    std::pair<tbb::tick_count, tbb::tick_count> time;
    time.first = tbb::tick_count::now();
    double seq = SimpsonMethod::sequential(body, seg_begin, seg_end, 10000000);
    time.second = tbb::tick_count::now();
    std::cout << "Sequential " << (time.second - time.first).seconds() << ' '
              << seq << std::endl;
    time.first = tbb::tick_count::now();
    double par = SimpsonMethod::parallel(body, seg_begin, seg_end, 10000000);
    time.second = tbb::tick_count::now();
    std::cout << "Parallel " << (time.second - time.first).seconds() << ' '
              << par << std::endl;
    ASSERT_NEAR(seq, par, 1e-6);
}*/

TEST(TBB_SimpsonMethodTest, can_integrate_2d_function) {
    std::vector<double> seg_begin = {0};
    std::vector<double> seg_end = {2};
    double square = SimpsonMethod::parallel(parabola, seg_begin, seg_end, 100);
    ASSERT_NEAR(16.0 / 3.0, square, 1e-6);
}

TEST(TBB_SimpsonMethodTest, can_integrate_3d_function) {
    std::vector<double> seg_begin = {0, 0};
    std::vector<double> seg_end = {1, 1};
    double volume = SimpsonMethod::parallel(body, seg_begin, seg_end, 100);
    ASSERT_NEAR(2.0 / 3.0, volume, 1e-6);
}

// Calculated by WolframAlpha with the following query:
// integrate (sin(x + 3) - ln(y) + z^2), x=[-2, 1], y=[1, 3], z=[0, 2]
TEST(TBB_SimpsonMethodTest, can_integrate_super_function) {
    std::vector<double> seg_begin = {-2, 1, 0};
    std::vector<double> seg_end = {1, 3, 2};
    double integral = SimpsonMethod::parallel(super, seg_begin, seg_end, 100);
    ASSERT_NEAR(13.0007625, integral, 1e-6);
}

TEST(TBB_SimpsonMethodTest, cannot_accept_empty_segment_vectors) {
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {}, {}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {0}, {}, 100));
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {}, {0}, 100));
}

TEST(TBB_SimpsonMethodTest, cannot_accept_invalid_segment_vectors) {
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {1, 2}, {1, 2, 3}, 100));
```

```
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {1, 2, 3}, {1, 2}, 100));
}

TEST(TBB_SimpsonMethodTest, cannot_accept_invalid_steps_count) {
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {0}, {0}, 0));
    ASSERT_ANY_THROW(SimpsonMethod::parallel(generic, {0}, {0}, -1));
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```