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

Решение многомерных интегралов методом Симпсона.

Выполнил: Студент ИИТММ гр. 381608

Большаков К.

Проверил: Доцент кафедры МОиСТ,

кандидат технических наук

Сысоев А. В.

Нижний Новгород 2019 г.

Оглавление

Введение

Формула Симпсона (также **Ньютона-Симпсона**[1]) относится к приёмам численного интегрирования.

Суть метода заключается в приближении подынтегральной функции на отрезке интерполяционным многочленом второй степени , то есть приближение графика функции на отрезке параболой. Метод Симпсона имеет порядок погрешности 4 и алгебраический порядок точности 3.

Сама формула для интеграла:

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

Также формулу можно записать используя только известные значения функции, то есть значения в узлах:

$$\int_a^b f(x)dx \approx \frac{h}{3} \cdot \sum_{k=1,2}^{N-1} [f(x_{k-1}) + 4f(x_k) + f(x_{k+1})]$$

Общая погрешность при интегрировании по отрезку [a,b] с шагом h определяется по формуле:

$$|E(f)| \leqslant \frac{(b-a)}{2880} h^4 \max_{x \in [a,b]} |f^{(4)}(x)|.$$

Постановка задачи

Задачей практической работы является разработка программы, реализующей последовательный и параллельный алгоритм применения к многомерным интегралам методом Симпсона**.**

Так как двумерный интеграл является частным случаем многомерного, то можно принять его за решение.

Ход работы:

- Реализация линейной версии алгоритма.
- Реализация параллельной версии с использованием OMP.
- Реализация параллельной версии с использованием TBB.
- Замерить время, оценить ускорение и эффективность каждой из версий.

Линейная версия

Предполагаемая функция интегрирования.

```
double func(double x, double y) {
    return x + y;
}
```

В качестве параметров функции Симпсона передается интегрируемая функция, значения отрезков на сетке.

```
double func_simpson(double f(double, double),
    double xp, double yp, double x, double y,
    double xn, double yn) {
    return (f(xp, yp) + 4 * f(x, y) + f(xn, yn));
}
```

В качестве параметров функции передается функция, которую предполагается интегрировать, нижняя граница по x, по y, так же передаются верхние границы интегрирования.
Заключительными двумя параметрами является шаги разбиения по x, y.

Определяется размер шага разбиения, выделение памяти и инициализация.

В переменной sum хранится вычисления на отрезках.

Вернем sum * h_x * h_y / 9 так как предполагается формула для двойного интеграла.

```
double integrate_linear(double f(double, double),
```

```
    double lower_bound_x, double lower_bound_y, double upper_bound_x, double
upper_bound_y, int y_n, int x_n) {
    double h_x = (upper_bound_x - lower_bound_x) / x_n;
    double h_y = (upper_bound_y - lower_bound_x) / y_n;
    double* x = new double[x_n + 1];
    double* y = new double[y_n + 1];
    double sum = 0;
    for (int i = 0; i <= x_n; i++) {
        x[i] = lower_bound_x + i * h_x;
    }
    for (int i = 0; i <= y_n; i++) {
        y[i] = lower_bound_y + i * h_y;
    }
    for (int i = 1; i < x_n; i += 2) {
        for (int j = 1; j < y_n; j += 2) {
            sum += func_simpson(f,
                x[i - 1], y[j - 1], x[i - 1], y[j], x[i - 1], y[j + 1]) +
                4 * func_simpson(f,
                    x[i], y[j - 1], x[i], y[j], x[i], y[j + 1]) +
                func_simpson(f,
                    x[i + 1], y[j - 1], x[i + 1], y[j], x[i + 1], y[j + 1]);
        }
    }
    delete[] x;
    delete[] y;
    return sum * h_x * h_y / 9;
}
```

В качестве параметров, был взят двойной интеграл от 0 до 200 с количеством шагов 20000.

```
double linear_result = integrate_linear(func, 0, 0, 200, 200, 20000, 20000);
```

Параллельная версия OMP

Так как OMP поддерживает директивы, достаточно использовать директиву редукции перед началом интегрирования.

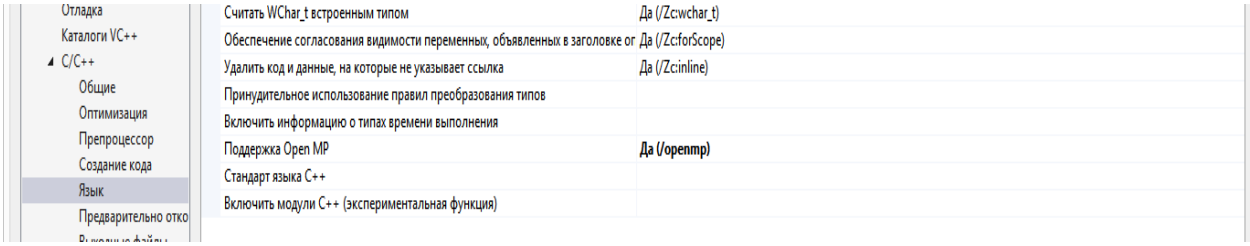
```
#pragma omp parallel for reduction(+:sum)
```

Код параллельной области:

```
double integrate_parallel_omp(double f(double, double),
    double lower_bound_x, double lower_bound_y, double upper_bound_x, double
upper_bound_y, int x_n, int y_n) {
    double h_x = (upper_bound_x - lower_bound_x) / x_n;
    double h_y = (upper_bound_y - lower_bound_x) / y_n;
    double* x = new double[x_n + 1];
    double* y = new double[y_n + 1];
    double sum = 0;
    for (int i = 0; i <= x_n; i++) {
        x[i] = lower_bound_x + i * h_x;
    }
    for (int i = 0; i <= y_n; i++) {
        y[i] = lower_bound_y + i * h_y;
    }
```

```
#pragma omp parallel for reduction(+:sum)
    for (int i = 1; i < x_n; i += 2) {
        for (int j = 1; j < y_n; j += 2) {
            sum += func_simpson(f,
                x[i - 1], y[j - 1], x[i - 1], y[j], x[i - 1], y[j + 1]) +
                4 * func_simpson(f,
                    x[i], y[j - 1], x[i], y[j], x[i], y[j + 1]) +
                func_simpson(f,
                    x[i + 1], y[j - 1], x[i + 1], y[j], x[i + 1], y[j + 1]);
        }
    }
    return sum * h_x * h_y / 9;
    delete[] x;
    delete[] y;
}
```

Так же необходимо подключить в настройках поддержку OMP.



Параллельная версия TBB

Функция библиотеки TBB parallel_reduse имеет параметры: итерационное пространство, функтор.

В TBB версии было использовано одномерное итерационное пространство, так как хранение переменных распределено не динамически.

Пространство является классом библиотеки TBB, реализовать можно с помощью лямбда выражений.

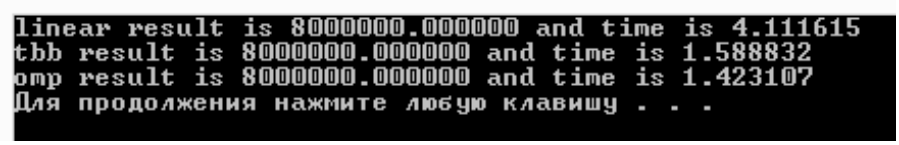
```
Размер разбиения блока
    size_t g_size = x_n / 8;

Будем хранить в переменной sum вычисления функции reduse
    double sum = tbb::parallel_reduce(

0, x_n – границы итерационного пространства, 0.0 начальное значение partSum
        tbb::blocked_range<size_t>(0, x_n, g_size), 0.0,

В функтор передаются по ссылке само пространство и переменная, в которой будет хранится
часть результатов
        [&](const tbb::blocked_range<size_t> &range, double partSum) -> double {

Инициализация итерационного пространства
        size_t begin = range.begin(), end = range.end();

Вычисления
        for (size_t i = begin + 1; i < end; i += 2) {
            for (int j = 1; j < y_n; j += 2) {
                partSum += func_simpson(f, x[i - 1], y[j - 1], x[i - 1], y[j], x[i - 1],
y[j + 1])
                    + func_simpson(f, x[i], y[j - 1], x[i], y[j], x[i], y[j + 1]) * 4
                    + func_simpson(f, x[i + 1], y[j - 1], x[i + 1], y[j], x[i + 1], y[j +
1]);
            }
        }
        return partSum;

Суммирование проинтегрированных частей
    }, std::plus<double>());
```

Код TBB версии:

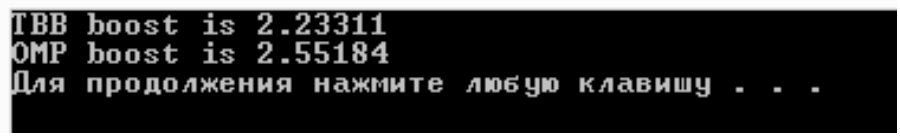
```
double integrate_parallel_tbb(double f(double, double),
    double lower_bound_x, double lower_bound_y, double upper_bound_x, double
upper_bound_y, int x_n, int y_n) {
    double h_x = (upper_bound_x - lower_bound_x) / x_n;
    double h_y = (upper_bound_y - lower_bound_x) / y_n;
    double* x = new double[x_n + 1];
    double* y = new double[y_n + 1];
    for (int i = 0; i <= x_n; i++) {
        x[i] = lower_bound_x + i * h_x;
    }
    for (int i = 0; i <= y_n; i++) {
        y[i] = lower_bound_y + i * h_y;
    }
    size_t g_size = x_n / 8;
    double sum = tbb::parallel_reduce(
        tbb::blocked_range<size_t>(0, x_n, g_size), 0.0,
        [&](const tbb::blocked_range<size_t> &range, double partSum) -> double {
        size_t begin = range.begin(), end = range.end();
        for (size_t i = begin + 1; i < end; i += 2) {
            for (int j = 1; j < y_n; j += 2) {
                partSum += func_simpson(f, x[i - 1], y[j - 1], x[i - 1], y[j], x[i - 1],
y[j + 1])
                    + func_simpson(f, x[i], y[j - 1], x[i], y[j], x[i], y[j + 1]) * 4
                    + func_simpson(f, x[i + 1], y[j - 1], x[i + 1], y[j], x[i + 1], y[j +
1]);
            }
        }
        return partSum;
    }, std::plus<double>());
    delete[]x;
    delete[]y;
    return sum * h_x * h_y / 9;
}
```

Результаты

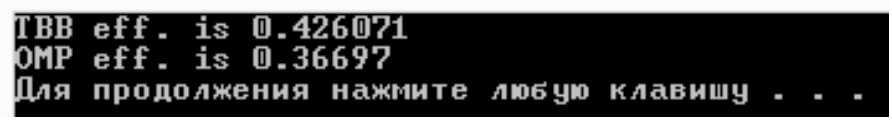
Время выполнения:



Ускорение:



Эффективность:



Код программы:

```
#include <tbb/tbb.h>
#include <stdio.h>
#include <omp.h>
#include <math.h>
#include <iostream>
#include <functional>

double func(double x, double y) {
    return x + y;
}
double func_simpson(double f(double, double),
    double xp, double yp, double x, double y,
    double xn, double yn) {
    return (f(xp, yp) + 4 * f(x, y) + f(xn, yn));
}
double integrate_linear(double f(double, double),
    double lower_bound_x, double lower_bound_y, double upper_bound_x, double
upper_bound_y, int y_n, int x_n) {
    double h_x = (upper_bound_x - lower_bound_x) / x_n;
    double h_y = (upper_bound_y - lower_bound_x) / y_n;
    double* x = new double[x_n + 1];
    double* y = new double[y_n + 1];
    double sum = 0;
    for (int i = 0; i <= x_n; i++) {
        x[i] = lower_bound_x + i * h_x;
    }
    for (int i = 0; i <= y_n; i++) {
        y[i] = lower_bound_y + i * h_y;
    }
    for (int i = 1; i < x_n; i += 2) {
        for (int j = 1; j < y_n; j += 2) {
            sum += func_simpson(f,
                x[i - 1], y[j - 1], x[i - 1], y[j], x[i - 1], y[j + 1]) +
                4 * func_simpson(f,
```

```
                    x[i], y[j - 1], x[i], y[j], x[i], y[j + 1]) +
                func_simpson(f,
                    x[i + 1], y[j - 1], x[i + 1], y[j], x[i + 1], y[j + 1]);
        }
    }
    delete[] x;
    delete[] y;
    return sum * h_x * h_y / 9;
}
double integrate_parallel_omp(double f(double, double),
    double lower_bound_x, double lower_bound_y, double upper_bound_x, double
upper_bound_y, int x_n, int y_n) {
    double h_x = (upper_bound_x - lower_bound_x) / x_n;
    double h_y = (upper_bound_y - lower_bound_x) / y_n;
    double* x = new double[x_n + 1];
    double* y = new double[y_n + 1];
    double sum = 0;
    for (int i = 0; i <= x_n; i++) {
        x[i] = lower_bound_x + i * h_x;
    }
    for (int i = 0; i <= y_n; i++) {
        y[i] = lower_bound_y + i * h_y;
    }
#pragma omp parallel for reduction(+:sum)
    for (int i = 1; i < x_n; i += 2) {
        for (int j = 1; j < y_n; j += 2) {
            sum += func_simpson(f,
                x[i - 1], y[j - 1], x[i - 1], y[j], x[i - 1], y[j + 1]) +
                4 * func_simpson(f,
                    x[i], y[j - 1], x[i], y[j], x[i], y[j + 1]) +
                func_simpson(f,
                    x[i + 1], y[j - 1], x[i + 1], y[j], x[i + 1], y[j + 1]);
        }
    }
    return sum * h_x * h_y / 9;
    delete[] x;
    delete[] y;
}
double integrate_parallel_tbb(double f(double, double),
    double lower_bound_x, double lower_bound_y, double upper_bound_x, double
upper_bound_y, int x_n, int y_n) {
    double h_x = (upper_bound_x - lower_bound_x) / x_n;
    double h_y = (upper_bound_y - lower_bound_x) / y_n;
    double* x = new double[x_n + 1];
    double* y = new double[y_n + 1];
    for (int i = 0; i <= x_n; i++) {
        x[i] = lower_bound_x + i * h_x;
    }
    for (int i = 0; i <= y_n; i++) {
        y[i] = lower_bound_y + i * h_y;
    }
    size_t g_size = x_n / 8;
    double sum = tbb::parallel_reduce(
        tbb::blocked_range<size_t>(0, x_n, g_size), 0.0,
        [&](const tbb::blocked_range<size_t> &range, double partSum) -> double {
        size_t begin = range.begin(), end = range.end();
        for (size_t i = begin + 1; i < end; i += 2) {
            for (int j = 1; j < y_n; j += 2) {
                partSum += func_simpson(f, x[i - 1], y[j - 1], x[i - 1], y[j], x[i - 1],
y[j + 1])
                    + func_simpson(f, x[i], y[j - 1], x[i], y[j], x[i], y[j + 1]) * 4
                    + func_simpson(f, x[i + 1], y[j - 1], x[i + 1], y[j], x[i + 1], y[j +
1]);
            }
```

```
        }
        return partSum;
    }, std::plus<double>());
    delete[]x;
    delete[]y;
    return sum * h_x * h_y / 9;
}

int main(int argc, char* argv[]) {
    tbb::tick_count linear_t1 = tbb::tick_count::now();
    double linear_result = integrate_linear(func, 0, 0, 200, 200, 20000, 20000);
    tbb::tick_count linear_t2 = tbb::tick_count::now();
    double linear_time = (linear_t2 - linear_t1).seconds();
    std::cout << "linear result is " << std::fixed << linear_result << " and time is " <<
linear_time << std::endl;

    tbb::tick_count tbb1 = tbb::tick_count::now();
    double tbb_result = integrate_parallel_tbb(func, 0, 0, 200, 200, 20000, 20000);
    tbb::tick_count tbb2 = tbb::tick_count::now();
    double tbb_time = (tbb2 - tbb1).seconds();
    std::cout << "tbb result is " << std::fixed << tbb_result << " and time is " <<
tbb_time << std::endl;
    std::cout << "TBB boost is " << linear_time / tbb_time << std::endl;
    std::cout << "TBB eff. is " << tbb_time / 4<<std::endl;

    double p_time1 = omp_get_wtime();
    double omp_result = integrate_parallel_omp(func, 0, 0, 200, 200, 20000, 20000);
    double p_time2 = omp_get_wtime() - p_time1;
    std::cout << "OMP eff. is " << p_time2 / 4<<std::endl;
    std::cout << "OMP boost is " << linear_time / p_time2<<std::endl;
    std::cout << "omp result is " << omp_result << " and time is " << p_time2 <<
std::endl;
    system("pause");
    return 0;
}
```

Вывод

В результате проделанной работы, было изучено численное интегрирование многомерных интегралов, реализована линейная версия, OMP-версия с помощью директив, TBB-версия с помощью итерационного пространства лямбда выражения, были проведены замеры времени выполнения версий, эффективность. TBB в данной задаче уступает по скорости OMP, но при достаточно больших значениях показатели выравниваются.

Литература

1. *Костомаров Д. П., Фаворский А. П.* Вводные лекции по численным методам. М.: Логос, 2004. 184 с. ISBN 5-94010-286-7
2. *Петров И. Б., Лобанов А. И.* Лекции по вычислительной математике. М.: Интуит, Бином, 2006. 523 с. ISBN 5-94774-542-9
3. *Richard Gerber.* Начало работы с OpenMP*. Intel Software Network (5 июня 2009 года). Дата обращения 11 февраля 2010. Архивировано 3 марта 2012 года.
4. *Richard Gerber.* Эффективное распределение нагрузки между потоками с помощью OpenMP*. Intel Software Network (5 июня 2009 года). Дата обращения 11 февраля 2010. Архивировано 3 марта 2012 года.
5. *Andrey Karpov.* Кратко о технологии OpenMP. Intel Software Network (5 января 2010 года). Дата обращения 11 февраля 2010. Архивировано 3 марта 2012 года.
6. 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.
7. Voss, M. (2006, October). «Demystify Scalable Parallelism with Intel Threading Building Blocks' Generic Parallel Algorithms.»
8. Voss, M. (2006, December). «Enable Safe, Scalable Parallelism with Intel Threading Building Blocks' Concurrent Containers.»
9. Hudson, R. L., B. Saha, et al. (2006, June). «McRT-Malloc: a scalable transactional memory allocator.» Proceedings of the 2006 International Symposium on Memory Management. New York: ACM Press, pp. 74–83.