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 двумерного интеграла:

$$\iint_{a\ d}^{b\ c} f(x,y)dxdy = \frac{hx * hy}{3 * 3} \sum_{i=1,2}^{N-1} \sum_{j=1,2}^{N-1} \Big(f\big(x_{i-1}, y_{j-1}\big), f\big(x_{i-1}, y_j\big), f\big(x_{i-1}, y_{j+1}\big)\Big) + 4 * (f(x_i, y_{j-1}) f(x_i, y_{j-1}) f(x_i, y_{j-1})) + (f(x_{i+1}, y_{j-1}) f(x_{i+1}, y_{j-1}) f(x_{i+1}, y_{j-1}))$$

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

$$|E(f)| \leq \frac{(b-a)}{2880} h^4 \max\left|f^{(4)}(x)\right|, x\epsilon[a,b]$$

Суть метода парабол.

На каждом интервале подынтегральная функция приближается квадратичной параболой

$y = a_i x^2 + b_i + c_i$, проходящей через точки . Отсюда и название метода - метод парабол.

Это делается для того, чтобы в качестве приближенного значения определенного интеграла взять $\int (a_i x^2 + b_i + c_i) dx$, который мы можем вычислить по формуле Ньютона-Лейбница.

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.

Описание алгоритма решения.

Линейная версия

Предполагаемая функция интегрирования.

$$\int_{0}^{200}\int_{0}^{200} (x+y)dxdy$$

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

$$f(x_{i-1}, y_{i-1}) + 4 * f(x_i, y_i) + f(x_{i+1}, y_{i+1})$$

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.

```
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;
}
```

Линейное интегрирование двумерного интеграла

В качестве параметра, был взят двойной интеграл с количеством шагов 20000.

Параллельная версия OMP

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

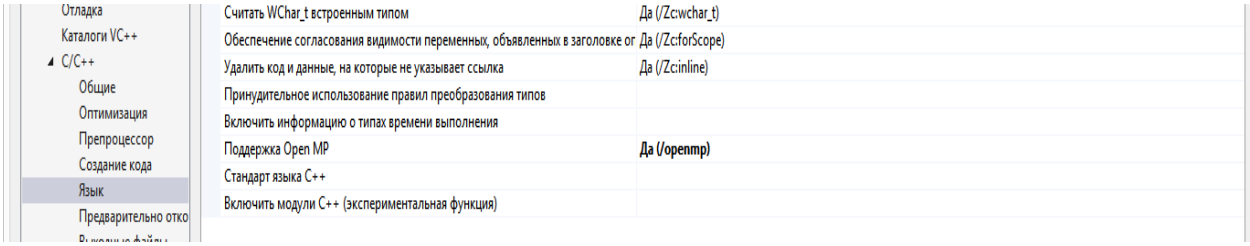
```
    #pragma omp parallel for reduction(+:sum)
```

Циклы, распараллеливаемые OMP:

```
#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]);
        }
    }
}
```

OMP-версия распараллеливания интеграла

Так же необходимо подключить в настройках поддержку OMP.



Установка поддержки OpenMP в VS2017

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.

Распараллеливание TBB:

```
    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>());
```

Одномерное итеграционнное пространство для распараллеливания циклов.

Результаты экспериментов

Эксперименты были проведены с характеристиками центрального процессора:

Intel® Core™ i7-4500U

Количество ядер: **2**

Логических процессоров: **4**

Результаты запусков:

Кол-во шагов	Кол-во потоков	1	2		4		8	
		Seq	OMP	TBB	OMP	TBB	OMP	TBB
400		0.005828	0.001525	0.002553	0.003054	0.003166	0.005620	0.009103
4000		0.164900	0.109148	0.109361	0.058253	0.100979	0.066016	0.084567
40000		14.682475	8.917438	10.212762	6.252804	6.861390	6.499271	7.377013
90000		72.430375	47.925228	48.663102	25.665090	30.504903	33.747248	37.424803

Таблица результатов(время работы в секундах)

Пример времени выполнения для 20000 шагов на 4 потоках

```
linear result is 8000000.000000 and time is 4.111615
tbb result is 8000000.000000 and time is 1.588832
omp result is 8000000.000000 and time is 1.423107
Для продолжения нажмите любую клавишу . . .
```

Пример ускорения для 20000 шагов на 4 потоках

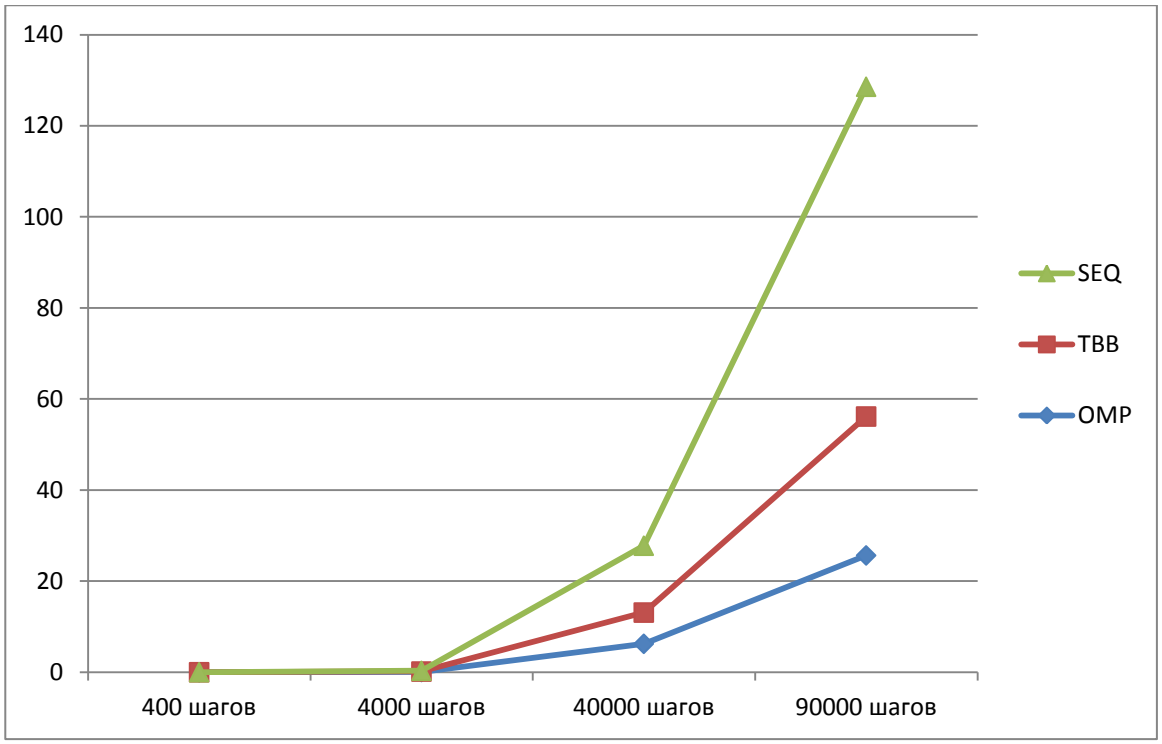
```
TBB boost is 2.23311
OMP boost is 2.55184
Для продолжения нажмите любую клавишу . . .
```

Пример эффективности для 20000 шагов на 4 потоках

```
TBB eff. is 0.426071
OMP eff. is 0.36697
Для продолжения нажмите любую клавишу . . .
```

Покажем на диаграмме результаты запусков программы на 4 потоках, при разном количестве шагов.

Таблица 1- Скорость выполнения программы (в секундах)



Вывод

При исследовании работы программы, ускорение TBB было меньше чем OMP, обусловлено это тем, что, в кеше хранились данные, а так же обусловлено самой задачей , так как библиотеку TBB рекомендуется использовать с более сложными задачами. TBB библиотека имеет более сложный алгоритм планирования, написан на объектно-ориентированном языке C++.

Заключение

В результате проделанной работы, было изучено численное интегрирование многомерных интегралов, реализована линейная версия, OMP-версия, TBB-версия, были проведены замеры времени выполнения версий, эффективность, что позволило оценить время выполнения параллельных версий.

Приложения

Приложение №1:

```
// Copyright 2019 Bolshakov Konstantin
#include <stdio.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(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;
}
int main(int argc, char* argv[]) {
    double result = integrate(func, 0, 0, 2, 2, 2000, 2000);
    std::cout << "result is " << std::fixed << result << std::endl;
    return 0;
}
```

Приложение №2:

```
// Copyright 2019 Bolshakov Konstantin
#include <omp.h>
#include <stdio.h>
#include <math.h>
#include <iostream>
double func1(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) + 2* 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,
    double h_x, double h_y, int x_n, int y_n) {
    h_x = (upper_bound_x - lower_bound_x) / x_n;
    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]);
        }
    }
    return sum * h_x * h_y / 9;
}
double integrate_parallel(double f(double, double),
    double lower_bound_x, double lower_bound_y,
    double upper_bound_x,
    double upper_bound_y, double h_x,
    double h_y, int x_n, int y_n) {
    h_x = (upper_bound_x - lower_bound_x) / x_n;
    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;
    }
    int main(int argc, char* argv[]) {
        double t1 = omp_get_wtime();
        double linear_result = integrate_linear
        (func1, 11, 22, 33, 44, 0.0001, 0.0001, 300, 300);
        double linear_time = omp_get_wtime() - t1;
        omp_set_num_threads(4);
        t1 = omp_get_wtime();
        double parallel_result = integrate_parallel
        (func1, 11, 22, 33, 44, 0.0001, 0.0001, 300, 300);
        double parallel_time = omp_get_wtime() - t1;
        double boost = linear_time / parallel_time;
        std::cout << "linear time is " << linear_time<< " and result is "
        << linear_result << std::endl;
        std::cout << "parallel time is " << parallel_time << " and result is "
        << parallel_result << std::endl;
        std::cout << "boost is " << boost << std::endl;
        return 0;
    }
```

Приложение №3

```
// Copyright 2019 Bolshakov Konstantin
#include <tbb/tbb.h>
#include <stdio.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_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, 2, 2, 2000, 2000);
    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, 2, 2, 2000, 2000);
    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 << "boost is " << linear_time/tbb_time <<std::endl;
    return 0;
}
```

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.