МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

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

(ННГУ)

Отчёт по лабораторной работе

Вычисление многомерных интегралов с использованием многошаговой схемы (метод прямоугольников)

Выполнил:
студент гр. 381706-4
Епремян Н.Г

Проверил:
Доцент кафедры МОСТ
Сысоев. А.В.

Нижний Новгород
2020

Оглавление

Введение

Метод прямоугольников — метод численного интегрирования функции, заключающийся в замене подынтегральной функции на многочлен нулевой степени, то есть константу, на каждом элементарном отрезке. Если рассмотреть график подынтегральной функции, то метод будет заключаться в приближённом вычислении площади (кратность 1) / объема (кратность 2) и т.д., под графиком суммированием конечного числа прямоугольников (кратность 1) / прямоугольных параллелепипедов (кратность 2) и т.д. , ширина/площадь которых будет определяться расстоянием между соответствующими соседними узлами интегрирования, а высота — значением подынтегральной функции в этих узлах.

Постановка задачи

Разработать и реализовать программу для работы с алгоритмом вычисления многомерных интегралов с использованием многошаговой схемы (метода прямоугольников). Распараллелить вычисления с использованием технологий OpenMP и TBB.

Данная программа должна поддерживать следующие возможности:

- Ввод границ интегрирования
- Подсчет времени исполнения последовательной и параллельной версии

Метод решения

Алгоритм работает по общей формуле для приближенного вычисления интеграла (или интеграла по геометрическому смыслу) первого порядка. Переносится алгоритм на многомерный случай интуитивно.

Алгоритм заключается в следующем: двигаясь с конкретным наперед известным шагом мы на каждом шаге имеем текущее значение (значение переменной до куда мы добрались своими шагами) и ширину шага. Для вычисления мы суммируем произведения высоты (значение функции в текущей точке) и ширину шага (так получаем площадь прямоугольника, отсюда и название). Стоить заметить, что известны три популярных метода-правых, левых и средних. Разница лишь в текущем значении, он либо на левом, либо на правом краю шага или вовсе в середине. При усложнении задачи с учетом мерности ситуация сильно не меняется. Только дальше у нас будет не ширина, а площадь, потом объем. Дальше в силу ограниченности человеческих возможностей ясную фигуру мы назвать не сможем, но мера будет такая же как и кратность (м, м^2, м^3, м^4). В общем виде формула получится такая:

$$\int_{n_1}^{m_1} di_1 \int_{n_2}^{m_2} di_2 \dots \int_{n_k}^{m_k} \varphi(i_1, \dots, i_k) di_k = \sum_{i_1=n_1}^{m_1} \sum_{i_2=n_2}^{m_2} \dots \sum_{i_k=n_k}^{m_k} \varphi(i_1, , \dots i_k) * \prod i$$

Схема распараллеливания

При последовательной версии мы вызываем функцию интегрирования, которая полностью обходит все области с заданной нами точностью (шагом).

Параллельная версия будет отличаться тем, что мы делим область интегрирования первого интеграла на количество потоков. Каждый поток будет обрабатывать свой отрезок. По сути распараллеливаться будут только циклы в которых считается интегральная сумма.

Описание программной реализации для однопоточной версии

Для упрощения решения этой задачи зафиксируем количество аргументов подынтегральной функции. Возьмем кратность интеграла равной двум.

Первым делом для всех версий задали функцию, для которой будем считать интеграл:

```
5  double function(double x, double y)
6  {
7    return sin(x + y) * x * x * cos(x - 2 * y);
8  }
9
```

Создаем переменные для расчета времени работы.

```
12
13    clock_t start_time;
14    clock_t end_time;
15
```

Берем входные данные из аргументов командной строки. Передаем только пределы интегрирования.

```
25    //check
26    const double step = 0.01;
27    double Sum = 0;
28    const double x_start = atof(argv[1]);
29    const double x_end= atof(argv[2]);
30    const double y_start = atof(argv[3]);
31    const double y_end = atof(argv[4]);
```

Следуя алгоритму в двух циклах обходим кратный интеграл.

```
34  for (int k = 0; k < 5; k++)
35    Sum = 0;
36    for (double i = x_start; i < x_end; i += step)
37      for (double j = y_start; j < y_end; j += step)
38        Sum += step*step*function(i, j);
```

Описание программной реализации для OpenMP версии

Первым отличием OpenMP версии будет передача количества создаваемых потоков через командную строку.

```
25  //check
26  const double step = 0.0001;
27  double Sum = 0;
28  const double x_start = atof(argv[1]);
29  const double x_end = atof(argv[2]);
30  const double y_start = atof(argv[3]);
31  const double y_end = atof(argv[4]);
32  const int threads = atoi(argv[5]);
33
```

Далее проинициализируем количество потоков выполняющих вычисления.

```
36  omp_set_num_threads(threads);
```

После чего распараллеливаем цикл используя директиву OpenMP.

#pragma omp parallel for reduction(+:Sum) schedule(dynamic) говорит о том что распараллеливается цикл, использующий суммирование с динамическим распределением порций. Это значит, что порции будут выдаваться по очереди каждому первому свободному потоку.

```
39  #pragma omp parallel for reduction(+:Sum) schedule(dynamic)
40  for (int i = 0; i < x_counter; i++)
41  {
42    x_current = x_start + i * step;
43    for (double y = y_start; y < y_end; y += step)
44      Sum += step * step * function(x_current, y);
45  }
```

Описание программной реализации для TBB версии

На первых этапах реализация с TBB не сильно отличается от OpenMP. Передача количества потоков также происходит через командную строку. Для tbb вызывается функция *init(количество потоков).*

```
28    const double step = 0.0001;
29    const double x_start = atof(argv[1]);
30    const double x_end = atof(argv[2]);
31    const double y_start = atof(argv[3]);
32    const double y_end = atof(argv[4]);
33    const int threads = atoi(argv[5]);
34    int x_counter = int((x_end - x_start) / step);
35
36    tbb::task_scheduler_init init(threads);
37    tbb::tick_count start_time = tbb::tick_count::now();
38
```

Для параллельного вычисления вызывает функцию *parallel_reduce.* В качестве параметров ей передается объект созданный на основе границ интегрирования, функция, результат которой будет суммироваться. В данном случае это лямбда выражение.

```
43  double x_current;
44  auto totl = tbb::parallel_reduce(
45    tbb::blocked_range<int>(0, x_counter),
46    0.0,
47    [&](tbb::blocked_range<int> r, double running_total)
48    {
49      for (int i = r.begin(); i < r.end(); ++i)
50      {
51        x_current = x_start + i * step;
52        for (double y = y_start; y <= y_end; y += step)
53        {
54          running_total += step * step * function(x_current, y);
55        }
56      }
57
58      return running_total;
59    }, std::plus<double>());
60
```

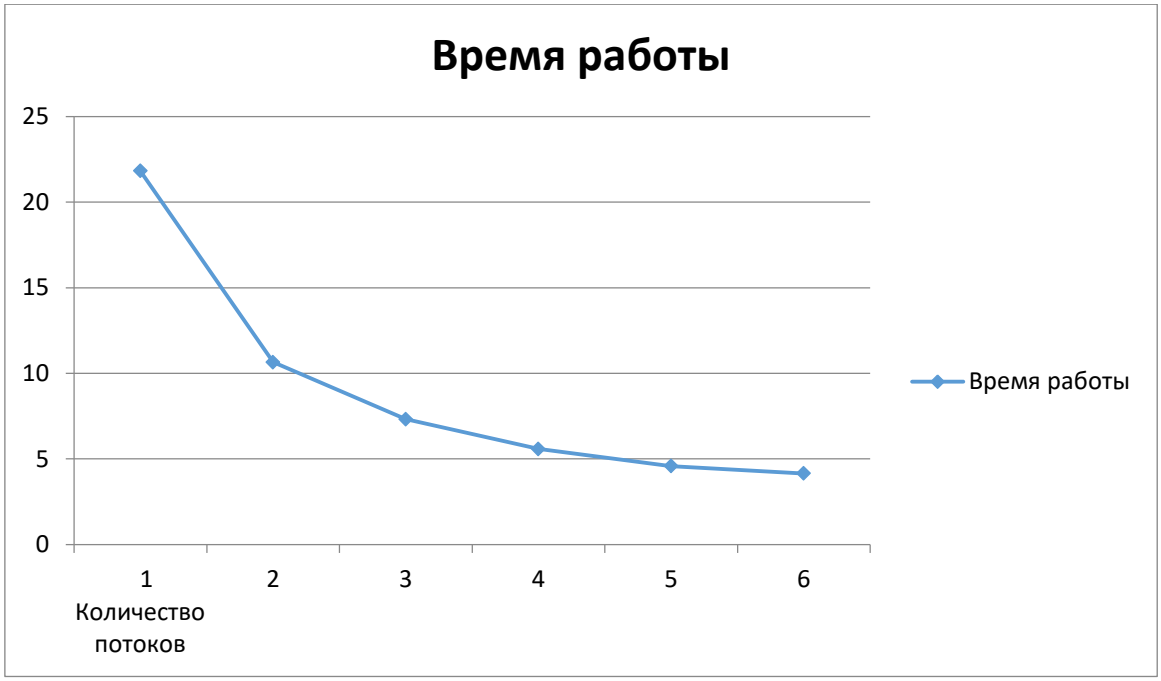
Результаты экспериментов по оценке масштабируемости

Для данного теста использовались расчеты следующих интегралов: Кратность=2, Пределы интегрирования:(0,2)(0,2), Шаг: 0.0001

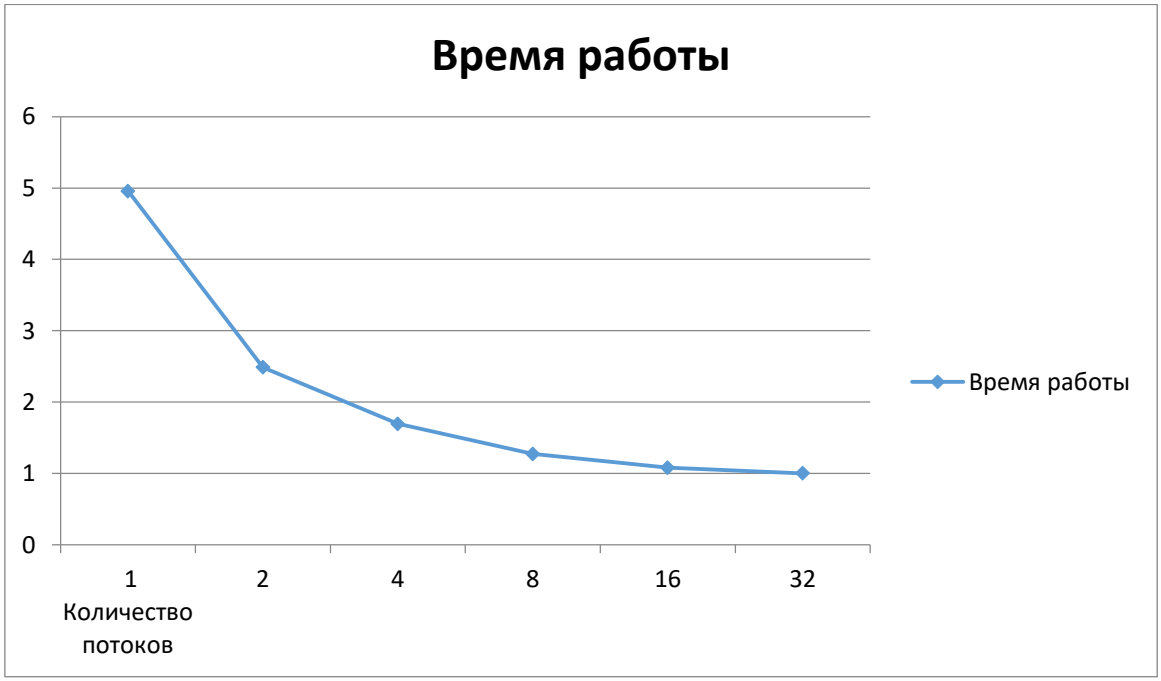
Один поток.

Время работы: 2,1549 сек

OpenMP.



TBB.



Заключение

В ходе выполнения лабораторной работы я разработал три реализации программы, решающей численно кратные интегралы. Одна из них однопоточная. Остальные распараллеливают вычисления с помощью технологий OpenMP и TBB. Получил прирост производительности на многопоточных системах. Ознакомился и разобрался с применением функций и возможностей OpenMP и TBB. А также убедился в необходимости параллельных вычислений.

Приложение.

1.Программный код однопоточной реализации

```
#include <iostream>
#include <ctime>
// function

double function(double x, double y)
{
	return sin(x + y) * x * x * cos(x - 2 * y);
}



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

	clock_t start_time;
	clock_t end_time;

	//test not enough  arguments
	if (argc<5) {
			std::cout << "NOT ENOUGH ARGUMENTS!" << std::endl;
			return 0;
	}
	else if (argc>5) {
			std::cout << "TOO MUCH ARGUMENTS!" << std::endl;
			return 0;
		}
	//check
	const double step = 0.01;
```

```
	double Sum = 0;
	const double x_start = atof(argv[1]);
	const double x_end= atof(argv[2]);
	const double y_start = atof(argv[3]);
	const double y_end = atof(argv[4]);

	start_time = clock();
	for (int k = 0; k < 5; k++)
		Sum = 0;
		for (double i = x_start; i < x_end; i += step)
			for (double j = y_start; j < y_end; j += step)
				Sum += step*step*function(i, j);
	end_time = clock();
	std::cout << "Result:" << Sum << std::endl;
	std::cout << "Time:" << ((double(end_time-start_time))/5*CLOCKS_PER_SEC) << std::endl;

	return 0;
}
```

2.Программный код реализации с OpenMP

```
#include <omp.h>
#include <iostream>
// function

double function(double x, double y)
{
	return sin(x + y) * x * x * cos(x - 2 * y);
}


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

```
double start_time;
double end_time;

//test not enough  arguments
if (argc < 6) {
	std::cout << "NOT ENOUGH ARGUMENTS!" << std::endl;
	return 0;
}
else if (argc > 6) {
	std::cout << "TOO MUCH ARGUMENTS!" << std::endl;
	return 0;
}
//check
const double step = 0.0001;
double Sum = 0;
const double x_start = atof(argv[1]);
const double x_end = atof(argv[2]);
const double y_start = atof(argv[3]);
const double y_end = atof(argv[4]);
const int threads = atoi(argv[5]);

int x_counter = int((x_end - x_start) / step);
double x_current = x_start;
omp_set_num_threads(threads);
start_time = omp_get_wtime();

#pragma omp parallel for reduction(+:Sum) schedule(dynamic)
for (int i = 0; i < x_counter; i++)
{
	x_current = x_start + i * step;
	for (double y = y_start; y < y_end; y += step)
		Sum += step * step * function(x_current, y);
}
```

```
	end_time = omp_get_wtime();

	std::cout << "Result:" << Sum << std::endl;
	std::cout << "Time:" << end_time - start_time << std::endl;

	return 0;
}
```

3. Программный код реализации с TBB

```
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/tick_count.h"
#include "tbb/parallel_reduce.h"

#include <iostream>

// function
double function(double x, double y)
{
	return sin(x + y) * x * x * cos(x - 2 * y);
}

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

	//test not enough  arguments
	if (argc < 6) {
		std::cout << "NOT ENOUGH ARGUMENTS!" << std::endl;
		return 0;
	}
	else if (argc > 6) {
		std::cout << "TOO MUCH ARGUMENTS!" << std::endl;
		return 0;
```

```
	}

	//init
	const double step = 0.0001;
	const double x_start = atof(argv[1]);
	const double x_end = atof(argv[2]);
	const double y_start = atof(argv[3]);
	const double y_end = atof(argv[4]);
	const int threads = atoi(argv[5]);
	int x_counter = int((x_end - x_start) / step);

	tbb::task_scheduler_init init(threads);
	tbb::tick_count start_time = tbb::tick_count::now();

	// create portion
	int portion = 1;
	if ((x_counter / threads) > 2) portion = (int)(x_counter / threads);

	double x_current;
	auto totl = tbb::parallel_reduce(
		tbb::blocked_range<int>(0, x_counter),
		0.0,
		[&](tbb::blocked_range<int> r, double running_total)
		{
			for (int i = r.begin(); i < r.end(); ++i)
			{
				x_current = x_start + i * step;
				for (double y = y_start; y <= y_end; y += step)
				{
					running_total += step * step * function(x_current,
y);
				}
			}
```

```
            return running_total;
        }, std::plus<double>());


tbb::tick_count end_time= tbb::tick_count::now();

//output
std::cout << "Result:" << totl << std::endl;
std::cout << "Time:" << (end_time- start_time).seconds() << std::endl;

init.terminate();
return 0;
```