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 прямоугольников)»

Выполнил:
студент группы 381706-2
Банденков Д.В.

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

Нижний Новгород
2019

Содержание

Введение

Численное интегрирование — приближенное вычисление значения определенного интеграла. Допустим, нам дана функция ***f(x)***, определенная на отрезке или многомерном кубе, и возможность получать ее численное значение в любой из точек области определения за фиксированное время. Задача — вычислить определенный интеграл данной функции по заданной области и дать оценку погрешности вычисления.

Большинство методов численного интегрирования, объединенны общим принципом: область интегрирования разбивается по каждой из осей на равное количество частей. В каждой из полученных маленьких областей интеграл приближается простой функцией (например, линейной), значение которой вычисляется явно (путем вычисления подынтегрального выражения в одной или нескольких точках). Ввиду линейности оператора интегрирования по областям полученные значения суммируются и представляют собой результат интегрирования.

К данному типу относятся одномерные метод прямоугольников, метод трапеций, метод парабол (метод Симпсона). Перечисленные методы обобщается и для многомерного случая. Обобщенно все эти методы называются квадратурными. Также говорится, что перечисленные методы используют квадратурные формулы.

Постановка задачи

Цель данной работы – реализовать параллельный алгоритм метода прямоугольников и сравнить его с последовательным алгоритмом.

Данная цель предполагает решение следующих основных задач:

1. Изучение и реализация общей схемы численного интегрирования для многомерных случаев.
2. Написание последовательной и параллельной версии этих алгоритмов.
3. Тестирование работоспособности написанных алгоритмов посредствам тестов, написанных с использованием Google C++ Testing Framework.
4. Провести численные эксперименты, и сравнить результаты тестирования.

Метод решения

Пусть функция $y = f(x)$ непрерывна на отрезке $[a; b]$. Нам требуется вычислить определенный интеграл $\int_a^b f(x)dx$.

Обратимся к понятию определенного интеграла. Разобьем отрезок $[a;b]$ на n частей $[x_{i-1}; x_i]$, $i = 1, 2, \ldots, n$ точками $a = x_0 < x_1 < x_2 < \ldots < x_{n-1} < x_n = b$. Внутри каждого отрезка $[x_{i-1}; x_i]$, $i = 1, 2, \ldots, n$ выберем точку ξ_i. Так как по определению определенный интеграл есть предел интегральных сумм при бесконечном уменьшении длины элементарного отрезка разбиения $\lambda = \max\limits_{i = 1,2,\ldots,n} x_i - x_{i-1} \to 0$, то любая из интегральных сумм является приближенным значением интеграла $\int_a^b f(x)dx \approx \sum_{i=1}^n f(\xi_i)(x_i - x_{i-1})$.

Если разбить интегрируемый отрезок $[a;b]$ на одинаковые части точкой h, то получим $a=x_0$, $x_1=x_0+h$, $x_2=x_0+2h,\ldots$, $x_{n-1}=x_0+(n-1)h$, $x_n=x0+nh=$, то есть $h= x_i-x_{i-1} =(b-a)/n$, $i=1, 2,\ldots, n$. Серединами точек ξ_i выбираются элементарные отрезки $[x_{i-1}; x_i]$, $i = 1, 2, \ldots, n$, значит $\xi_i=x_{i-1} + h/2$, $i=1, 2,\ldots, n$.

Тогда приближенное значение $\int_a^b f(x)dx \approx \sum_{i=1}^n f(\xi_i)(x_i - x_{i-1})$ записывается таким образом $\int_a^b f(x)dx \approx h \cdot \sum_{i=1}^n f(\xi_i)(x_{i-1} + \frac{h}{2})$. Данная формула называется формулой метода средних прямоугольников.

Такое название метод получает из-за характера выбора точек ξ_i, где $h = \frac{b-a}{n}$ называют шагом разбиения отрезка $[a; b]$.

Основная суть метода прямоугольников заключается в том, что в качестве приближенного значения определенного интеграла берут интегральную сумму.

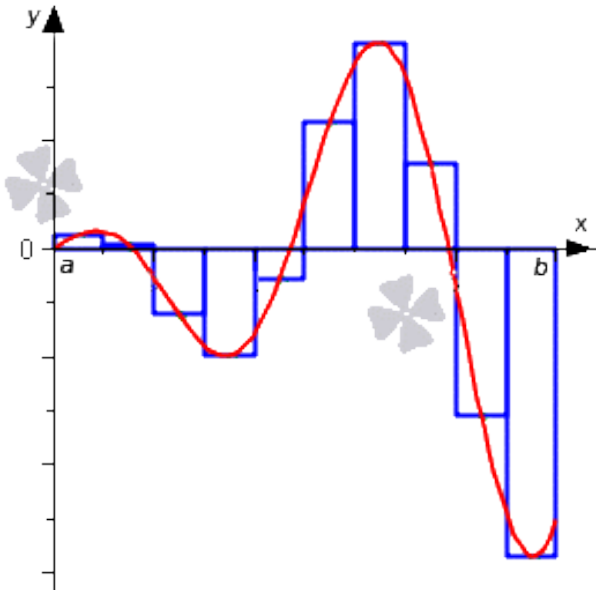

Рисунок 1. Геометрическое представление метода средних прямоугольников

Схема распараллеливания

В работе осуществляется распараллеливание интеграла на заданное число процессов. В начале имеется область интегрирование, заданная отрезками $[a_i; b_i]$, $i = 1, 2, \ldots, n$, количество которых совпадает с количеством интегралов n в многомерном интеграле. Также изначально задаётся шаг разбиения данного отрезка; так как для каждого примера имеется своё индивидуальное значение.

Один из наиболее простых подходов распараллеливания метода прямоугольников является геометрическая декомпозиция данных. Данный подход предполагает разделение данных на части и применение к ним одного и того же алгоритма. В численном интегрировании мы имеем дело с прямоугольной сеткой, в каждом узле которой вычисляется функция и умножается на квадрат шага. Вычисление функции в одном узле сетки не зависит от соседних узлов, таким образом, поделив сетку между процессами, можно получить параллельную версию,

Описание программной реализации

Рассмотрим основные моменты программной реализации.

Программа состоит из заголовочного файла `rectang_integ.h` в котором объявлены функции:

1. `double RectParall(double (*func)(std::vector<double>), std::vector <std::pair<double, double>> cordinate, const int leN);` - вычисление многомерного интеграла методом прямоугольников параллельная версия.
2. `double RectSequen(double (*func)(std::vector<double>), std::vector <std::pair<double, double>> cordinate, const int leN);` - вычисление многомерного интеграла методом прямоугольников последовательная версия.

В файле `rectang_integ.cpp` содержится реализация данных функций.

В файле `main.cpp` содержаться тесты для проверки корректности программы.

Подтверждение корректности

Для подтверждения корректности в программе реализован набор тестов с использованием библиотеки для модульного тестирования Google C++ Testing Framework:

Правильность получаемых результатов проверяется на пяти различных функциях разной сложности.

Примеры подынтегральных функций некоторых из них:

x + y * y * 7 + 12 - z * z * z

(log10(2 * x * x) + z + 5 * y

log10(2 * x * x) + sqrt(36) * 5 * sin(x-y) + sin(z * 12)

x + y * y * y

Все тесты проходят проверку, что является доказательством корректной работы программы

Результаты экспериментов

Эксперименты проводились на разном числе процессов для вычисления трехмерного интеграла для функции $F = log10(2 * x * x) + sqrt(36) * 5 * sin(x-y) + sin(z * 12)$ по области $x \in [1,117], y \in [-100,44], z \in [-29,172]$,с числом разбиений равным 200 для каждой переменной..

Вычисления проводились на ПК со следующими характеристиками:

- Процессор: AMD FX(tm) - 6300 CPU @ 3.5GHz
- Версия ОС: Windows 10 (64 бит)
- Оперативная память: 8104 МБ

Количество процессов	Время работы последовательного алгоритма, с.	Время работы параллельного алгоритма, с.	Ускорение
1	12.5308	14.0963	0,89
2	11.8551	7.65642	1,55
4	11.8738	4.6895	2.53
6	11.9258	3.7617	3.17
8	11.9242	3.637	3.28

Таблица 1. Время работы параллельной и последовательной версии алгоритма

Полученные данные демонстрируют разность во времени работы при последовательном и параллельном вычислениях. По результатам можно сделать вывод, что параллельное выполнение программы выигрывает во времени у последовательного во всех случаях, кроме случая с одним процессом, что логично, ведь в параллельном алгоритме, в этой ситуации довольно большие накладные расходы. Можно сделать вывод: чем больше процессов, тем быстрее работает программа.

Заключение

В результате выполнения лабораторной работы была разработана библиотека, реализующая параллельный метод интегрирования прямоугольниками, используя технологию MPI.

Для подтверждения корректности работы программы разработан и доведен до успешного выполнения набор тестов с использованием библиотеки модульного тестирования Google C++ Testing Framework.

По данным экспериментов удалось сравнить время работы параллельного и последовательного алгоритмов. Выявлено, что параллельный алгоритм интегрирования показывает более высокую эффективность на большем числе процессов.

Литература

Книги:

- Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. 184 с. ISBN 5-85746-602-4.

Internet-ресурсы:

- Национальный Открытый Университет «ИНТУИТ». Академия: Интернет-Университет Суперкомпьютерных Технологий. Курс: Теория и практика параллельных вычислений. Автор: Виктор Гергель. ISBN: 978-5-9556-0096-3. URL: https://www.intuit.ru/studies/courses/1156/190/info

Приложение

rectang_integ.h

```
// Copyright 2019 Bandenkov Daniil
#ifndef MODULES_TASK_3_BANDENKOV_D_RECTANG_INTEG_RECTANG_INTEG_H_
#define MODULES_TASK_3_BANDENKOV_D_RECTANG_INTEG_RECTANG_INTEG_H_
#include <vector>
#include <algorithm>
#include <utility>

double RectParall(double (*func)(std::vector<double>), std::vector <std::pair<double,
  double>> cordinate, const int leN);
double RectSequen(double (*func)(std::vector<double>), std::vector <std::pair<double,
  double>> cordinate, const int leN);

#endif  // MODULES_TASK_3_BANDENKOV_D_RECTANG_INTEG_RECTANG_INTEG_H_
```

rectang_integ.cpp

```
// Copyright 2019 Bandenkov Daniil
#include <mpi.h>
#include <vector>
#include <iostream>
#include <random>
#include <ctime>
#include <numeric>
#include <algorithm>
#include <stdexcept>
#include <utility>
#include "../../../modules/task_3/bandenkov_d_rectang_integ/rectang_integ.h"

using std::vector;
using std::pair;

double RectParall(double (*func)(vector<double>), vector <pair<double, double>> cordinate, const
int leN) {
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int n = cordinate.size();
  vector<double> step(n);
  vector <pair<double, double>> coord(n);
  int allElem;

  if (rank == 0) {
    allElem = 1;
    for (int i = 0; i < n; ++i) {
      step[i] = (cordinate[i].second - cordinate[i].first) / leN;
      coord[i] = cordinate[i];
    }
    int r = 0;
    while (r != n - 1) {
      allElem *= leN;
      r++;
    }
  }
  int ln = leN;
  int root = 0;
  MPI_Bcast(&allElem, 1, MPI_INT, root, MPI_COMM_WORLD);
  MPI_Bcast(&ln, n, MPI_INT, root, MPI_COMM_WORLD);
  MPI_Bcast(&step[0], n, MPI_DOUBLE, root, MPI_COMM_WORLD);
  MPI_Bcast(&coord[0], 2 * n, MPI_DOUBLE, root, MPI_COMM_WORLD);
```

```
  int ost = allElem % size;
  int count = allElem / size;
  if (rank < ost) {
    count += 1;
  }
  int temp = 0;
  if (rank < ost) {
    temp = rank * count;
  } else {
    temp = ost + rank * count;
  }

  vector<vector<double>> answVec(count);
  for (int i = 0; i < count; ++i) {
    int number = temp + i;
    for (int j = 0; j < n - 1; ++j) {
      answVec[i].push_back(coord[j].first + (number % leN + 0.5) * step[j]);
    }
  }
  double result = 0.0;
  for (int i = 0; i < count; ++i) {
    for (int j = 0; j < leN; ++j) {
      answVec[i].push_back(coord[n - 1].first + (j + 0.5) * step[n - 1]);
      result += func(answVec[i]);
      answVec[i].pop_back();
    }
  }
  double totalInteg = 0.0;
  MPI_Reduce(&result, &totalInteg, 1, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);
  for (int i = 0; i < n; ++i) {
    totalInteg *= step[i];
  }
  return totalInteg;
}


double RectSequen(double (*func)(vector<double>), vector <pair<double, double>> cordinate, const
int leN) {
  int  membs = 1;
  int razmer = cordinate.size();
  vector<double> h(razmer);
  for (int i = 0; i < razmer; ++i) {
    h[i] = (cordinate[i].second - cordinate[i].first) / leN;
    membs = membs * leN;
  }
  vector <double> answVec(razmer);
  double result = 0.0;
  for (int i = 0; i < membs; ++i) {
    for (int j = 0; j < razmer; ++j) {
      answVec[j] = cordinate[j].first + (i % leN + 0.5) * h[j];
    }
    result += func(answVec);
  }
  for (int i = 0; i < razmer; ++i) {
    result *= h[i];
  }
  return result;
}
```

main.cpp

```
// Copyright 2019 Bandenkov Daniil
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <mpi.h>
```

```cpp
#include <math.h>
#include <utility>
#include <vector>
#include "../../../modules/task_3/bandenkov_d_rectang_integ/rectang_integ.h"

using std::vector;
using std::pair;

double f1(vector<double> a) {
  double x = a[0];
  double y = a[1];;
  return (x + y * y * y);
}

double f2(vector<double> a) {
  double x = a[0];
  double y = a[1];
  double z = a[2];
  return (x + y * y * 7 + 12 - z * z * z);
}

double f3(vector<double> a) {
  double x = a[0];
  double y = a[1];
  double z = a[2];
  return (log10(2 * x * x) + z + 5 * y);
}

double f4(vector<double> a) {
  double x = a[0];
  double y = a[1];
  double z = a[2];
  return (log10(2 * x * x) + sqrt(36) * 5 * sin(x-y) + sin(z * 12));
}

double f5(vector<double> a) {
  double x = a[0];
  double y = a[1];
  return x + y;
}

TEST(RectangleIntegration, Double_Integral_0) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int n = 100;
  vector <pair<double, double>> a(2);
  a = { {7, 11}, {5, 6} };
  double result = RectParall(f5, a, n);
  if (rank == 0) {
    double error = 0.01;
    EXPECT_NEAR(result, RectSequen(f5, a, n), error);
  }
}
TEST(RectangleIntegration, Double_Integral_1) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int n = 100;
  vector <pair<double, double>> a(2);
  a = { {7, 19}, {12, 21} };
  double result = RectParall(f1, a, n);
  if (rank == 0) {
    double error = 0.01;
    EXPECT_NEAR(result, RectSequen(f1, a, n), error);
  }
}
```

```
TEST(RectangleIntegration, Triple_Integral_2) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int n = 100;
  vector <pair<double, double>> a(3);
  a = { {11, 17}, {29, 48}, {17, 21} };
  double result = RectParall(f2, a, n);
  if (rank == 0) {
    double error = 0.01;
    EXPECT_NEAR(result, RectSequen(f2, a, n), error);
  }
}

TEST(RectangleIntegration, Triple_Integral_3) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int n = 100;
  vector <pair<double, double>> a(3);
  a = { {54, 68}, {9, 18}, {7, 121} };
  double result = RectParall(f3, a, n);
  double error = 0.01;
  if (rank == 0) {
    EXPECT_NEAR(result, RectSequen(f3, a, n), error);
  }
}
TEST(RectangleIntegration, Triple_Integral_4) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int n = 200;
  vector<pair<double, double>> a(3);
  a = { {1, 117}, {-100, 44}, {-29, 172} };
  double l1, l2, dl;
  l1 = MPI_Wtime();
  double result = RectParall(f4, a, n);
  l2 = MPI_Wtime();
  dl = l2 - l1;
  double t1, t2, dt;
  if (rank == 0) {
    double error = 0.01;
    t1 = MPI_Wtime();
    EXPECT_NEAR(result, RectSequen(f4, a, n), error);
    t2 = MPI_Wtime();
    dt = t2 - t1;
    std::cout << "seq time " << dt << std::endl;
    std::cout << "par time " << dl << std::endl;

  }
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  MPI_Init(&argc, &argv);

  ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
  ::testing::TestEventListeners& listeners =
    ::testing::UnitTest::GetInstance()->listeners();

  listeners.Release(listeners.default_result_printer());
  listeners.Release(listeners.default_xml_generator());

  listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);

  return RUN_ALL_TESTS();
}
```