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)»

Выполнила:
студентка группы 381906-1
Тырина А.А.

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

Нижний Новгород
2021

Оглавление

Введение

Численное интегрирование — вычисление значения определённого интеграла (как правило, приближённое). Под численным интегрированием понимают набор численных методов для нахождения значения определённого интеграла. Одним из таких методов является метод Симпсона (парабол).

Метод Симпсона заключается в интегрировании интерполяционного многочлена второй степени функции f(x) с узлами интерполяции a, b и m = (a+b)/2 — параболы p(x).Для повышения точности имеет смысл разбить отрезок интегрирования на N равных промежутков(по аналогии с методом трапеций), на каждом из которых применить метод Симпсона.

Постановка задачи

В данной лабораторной работе требуется реализовать последовательный алгоритм вычисления многомерных интегралов методом Симпсона и параллельный алгоритм с помощью библиотеки MPI.

Для оценки эффективности работы программы нужно произвести серию экспериментов, сравнивающих время выполнения последовательной и параллельной версии алгоритма, сравнить полученные результаты и сделать выводы.

Описание алгоритма

Метод Симпсона заключается в интегрировании интерполяционного многочлена второй степени функции f(x) с узлами интерполяции a, b и m = (a+b)/2 — параболы p(x).Для повышения точности имеет смысл разбить отрезок интегрирования на N равных промежутков(по аналогии с методом трапеций), на каждом из которых применить метод Симпсона. Площадь параболы может быть найдена суммированием площадей 6 прямоугольников равной ширины. Высота первого из них должна быть равна f(a), с третьего по пятый — f(m), шестого — f(m). Таким образом, приближение методом Симпсона находим по формуле: $\int_a^b f(x)\,dx \approx \int_a^b P_2(x)\,dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$.

Для вычисления многомерных интегралов воспользуемся многошаговой схемой.

Схема распараллеливания

Распараллеливание происходит путем параллельного вычисления локальных сумм по формуле Симпсона. Каждый процесс вычисляет значения подынтегральной функции для своего числа отрезков, суммирует эти значения в свою локальную сумму. Затем функцией Reduce все локальные суммы складываются в глобальную в процессе с рангом 0. Останется только умножить результат на высоты оснований каждого интервала.

Описание программной реализации

Программа состоит из заголовочного файла simpson.h и двух файлов исходного кода simpson.cpp и main.cpp.

В заголовочном файле находятся прототипы функций для последовательного и параллельного вычисления многомерных интегралов.

Функция для последовательного алгоритма:

```
double getSequentialSimpson(function<double(vector<double>)> func,
                            vector<pair<double, double>> a_b, int n);
```

Входными параметрами функции является подынтегральная функция, массив пределов интегрирования, количество разбиений

Функция для параллельного алгоритма:

```
double getParallelSimpson(function<double(vector<double>)> f,
                          vector<pair<double, double>> limits, int n);
```

Входные параметры данной функции совпадают с входными параметрами функции для последовательного алгоритма.

В файле исходного кода simpson.cpp содержится реализация функций, объявленных в заголовочном файле. В файле исходного кода main.cpp содержатся тесты для проверки корректности программы.

Подтверждение корректности

Для подтверждения корректности работы данной программы с помощью фрэймфорка Google Test была разработана серия тестов. В каждом из тестов вычисляется значение контрольного интеграла заданной размерности при помощи последовательного и параллельного алгоритмов, подсчитывается время работы обоих алгоритмов, находится ускорение делением времени работы последовательного алгоритма на время работы параллельного. Результаты вычисления интеграла последовательным и параллельным способом сравниваются между собой в пределах погрешности, после чего можно сделать выводы об эффективнсти программы.

Успешное прохождение разработанных мной тестов подтверждает корректность работы программы.

Результаты экспериментов

Вычислительные эксперименты для оценки эффективности работы параллельного алгоритма проводились на ПК со следующими характеристиками:

- Процессор: Intel Core i5-10210U, 3.5 ГГц, количество ядер: 8;
- Оперативная память: 8 ГБ (DDR4), 3200 МГц;
- Операционная система: Windows 10 Home.

Результаты экспериментов представлены в Таблице 1.

Таблица 1: Результаты вычислительных экспериментов в тесте 4

Количество процессов	Время работы последовательного алгоритма (в секундах)	Время работы параллельного алгоритма (в секундах)	Ускорение
1	1.508	1.415	1.066
2	1.403	0.822	1.707
3	1.478	0.584	2.530
4	1.494	0.459	3.254
8	1.616	0.285	5.669
16	1.599	0.353	4.524

По данным, полученным в результате экспериментов, можно сделать вывод о том, что параллельный алгоритм работает до 5 раз быстрее, чем последовательный.

Заключение

В ходе выполнения данной лабораторной работы были разработаны последовательный и параллельный алгоритмы вычисления многомерных интегралов методом Симпсона. После выполенения серии экспериментов можно сделать вывод о том, что параллельный алгоритм работает быстрее последовательного, что доказывает эффективность параллельного алгоритма по сравнению с последовательным.

Были разработаны и доведены до успешного выполнения тесты, с использованием GoogleC++ Testing Framework, которые подтвердиди корректность выполнения программы.

Литература

1. Гергель В. П. Теория и практика параллельных вычислений. – 2007.
2. Гергель В. П., Стронгин Р. Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. – 2003.

Приложение

simpson.cpp

```
// Copyright 2021 Tyrina Anastasia
#include "../../../modules/task_3/tyrina_a_simpson/simpson.h"

#include <mpi.h>

#include <cmath>

double getSequentialSimpson(function<double(vector<double>)> func,
                            vector<pair<double, double>> a_b, int n) {
  int integral_size = a_b.size();
  vector<double> h(integral_size);
  int num = 1;

  for (int i = 0; i < integral_size; ++i) {
    h[i] = (a_b[i].second - a_b[i].first) / n;
    num *= n;
  }

  double sum = 0.0;
  for (int i = 0; i < num; ++i) {
    vector<vector<double>> x(integral_size);
    int tmp = i;
    for (int j = 0; j < integral_size; ++j) {
      x[j].push_back(a_b[j].first + tmp % n * h[j]);
      for (int k = 0; k < 4; k++) {
        x[j].push_back(a_b[j].first + tmp % n * h[j] + h[j] / 2);
      }
      x[j].push_back(a_b[j].first + tmp % n * h[j] + h[j]);
      tmp /= n;
    }
    vector<double> comb;
    for (int i = 0; i < pow(6, integral_size); ++i) {
      int tmp = i;
      for (int j = 0; j < integral_size; ++j) {
        comb.push_back(x[j][tmp % 6]);
        tmp /= 6;
      }
      sum += func(comb);
      comb.clear();
    }
    x.clear();
  }
  for (int i = 0; i < integral_size; ++i) {
    sum *= h[i] / 6.0;
  }
  return sum;
}

double getParallelSimpson(function<double(vector<double>)> func,
                          vector<pair<double, double>> a_b, int n) {
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int integral_size = a_b.size();
```

```
  std::vector<double> h(integral_size);

  int global_num;
  if (rank == 0) {
    global_num = 1;
    for (int i = 0; i < integral_size; ++i) {
      h[i] = (a_b[i].second - a_b[i].first) / n;
      global_num *= n;
    }
  }
  MPI_Bcast(&global_num, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(h.data(), integral_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  int rem = global_num % size;
  int delta, start;
  if (rank == 0) {
    delta = global_num / size + rem;
    start = 0;
  } else {
    delta = global_num / size;
    start = rem + delta * rank;
  }

  double local_sum = 0.0;
  for (int i = start; i < delta + start; ++i) {
    vector<vector<double>> x(integral_size);
    int tmp = i;
    for (int j = 0; j < integral_size; ++j) {
      x[j].push_back(a_b[j].first + tmp % n * h[j]);
      for (int k = 0; k < 4; k++) {
        x[j].push_back(a_b[j].first + tmp % n * h[j] + h[j] / 2);
      }
      x[j].push_back(a_b[j].first + tmp % n * h[j] + h[j]);
      tmp /= n;
    }

    vector<double> comb;
    for (int i = 0; i < pow(6, integral_size); ++i) {
      int temp = i;
      for (int j = 0; j < integral_size; ++j) {
        comb.push_back(x[j][temp % 6]);
        temp /= 6;
      }
      local_sum += func(comb);
      comb.clear();
    }
    x.clear();
  }

  double global_sum = 0.0;
  MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);

  if (rank == 0) {
    for (int i = 0; i < integral_size; ++i) {
      global_sum *= h[i] / 6.0;
    }
  }
  return global_sum;
}
```

main.cpp

```cpp
// Copyright 2021 Tyrina Anastasia
#include <gtest/gtest.h>

#include <cmath>
#include <gtest-mpi-listener.hpp>

#include "./simpson.h"

const function<double(vector<double>)> func1 = [](vector<double> vec) {
  double x = vec[0];
  double y = vec[1];
  return x * x - 2 * y;
};

const double eps = 0.0001;

TEST(SIMPSON_METHOD_MPI, TEST_1) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  std::vector<std::pair<double, double>> limits({{4, 10}, {1, 2}});
  int n = 100;

  double start = MPI_Wtime();
  double result = getParallelSimpson(func1, limits, n);
  double end = MPI_Wtime();

  if (rank == 0) {
    double ptime = end - start;

    start = MPI_Wtime();
    double reference_result = getSequentialSimpson(func1, limits, n);
    end = MPI_Wtime();
    double stime = end - start;

    std::cout << "Sequential: " << stime << std::endl;
    std::cout << "Parallel: " << ptime << std::endl;
    std::cout << "Speedup: " << stime / ptime << std::endl;
    ASSERT_NEAR(result, reference_result, eps);
  }
}

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();
}
```