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)»

Выполнил:
студент группы 381906-2
Шелепин Н.А.

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

Нижний Новгород
2021

Оглавление

Введение 3

Постановка задачи 4

Описание алгоритма 5

Схема распараллеливания 6

Описание программной реализации 7

Подтверждение корректности 8

Результаты экспериментов 9

Заключение 10

Литература 11

Приложение 12

Введение

Задачи численного интегрирования - распространённая задача, имеющая множество видов решения. Под численным интегрированием понимают набор численных методов для нахождения значения определённого интеграла. Основная идея большинства методов численного интегрирования состоит в замене подынтегральной функции на более простую, интеграл от которой легко вычисляется аналитически. Есть методы численного интегрирования, позволяющие получить значение определенного интеграла с требуемой степенью точности. Одним из таких методов является метод Симпсона (его еще называют методом парабол).

Формула Симпсона относится к приёмам численного интегрирования. Суть метода заключается в приближении подынтегральной функции на отрезке [a,b] интерполяционным многочленом второй степени, то есть приближение графика функции на отрезке параболой.

Постановка задачи

В данной лабораторной работе требуется реализовать последовательный и параллельные алгоритмы вычисления многомерных интегралов методом Симпсона.

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

Параллельный алгоритм должен быть реализован при помощи технологии MPI.

Описание алгоритма

Формулой Симпсона называется интеграл от интерполяционного многочлена второй степени на отрезке [a,b] $\int_a^b f(x)\,dx \approx \int_a^b P_2(x)\,dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$,

где $f(a)$, $f((a+b)/2)$ и $f(b)$ — значения функции в соответствующих точках (на концах отрезка и в его середине).

Для более точного вычисления интеграла, интервал [a,b] разбивают на N=2n элементарных отрезков одинаковой длины и применяют формулу Симпсона на составных отрезках. Каждый составной отрезок состоит из соседней пары элементарных отрезков. Значение исходного интеграла является суммой результатов интегрирования на составных отрезках:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_N)\right],$$

где $h = \frac{b-a}{N}$ — величина шага, а $x_j = a + jh$ — чередующиеся границы и середины составных отрезков, на которых применяется формула Симпсона.

Для вычисления многмерных интегралов воспользуемся многошаговой схемой.

Схема распараллеливания

Основная идея реализации параллельного алгоритма состоит в параллельном вычислении локальных сумм по формуле Симпсона. Заданное количество интервалов распределяется между определенным количеством процессов. Каждый процесс вычисляет значения подынтегральной функции для определенного числа отрезков, суммирует эти значения и отправляет получившиеся локальные суммы в процесс с рангом 0. В 0-ом процессе все локальные суммы сначала складываются в глобальную сумму. После результат будет умножаться на шаги разбиения отрезков интегрирования. В итоге получится приближенное значение искомого интеграла

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:

```
const std::function<double(std::vector<double>)>& f,
  const std::vector<std::pair<double, double>>& limits,
  const std::vector<int>& n
```

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

Функция для параллельного алгоритма:

```
double getParallelSimpson(const std::function<double(std::vector<double>)>& f,
                          const std::vector<std::pair<double, double>>& limits,
                          const std::vector<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: AMD FX-8320E, 3.5 ГГц, количество ядер: 8;
- Оперативная память: 16 ГБ (DDR4), 1600 МГц;
- Операционная система: Windows 10 Pro.

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	3.3475	3.76676	0.888696
2	3.35775	2.3388	1.43567
3	3.34642	1.89855	1.76261
4	3.35924	1.68735	1.99084
5	3.34809	1.57535	2.1253
6	3.35504	1.51344	2.21683
7	3.39476	1.51552	2.24
9	3.37919	1.5558	2.17199
11	3.41202	1.66682	2.04703
33	3.37404	1.99409	1.69202

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

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 Shelepin Nikita
#include <gtest/gtest.h>

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

#include "./simpson.h"

TEST(SIMPSON_METHOD, TEST_FUNCTION_1) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    printf("  f(x,y) = x * x - 2 * y\n");
    fflush(stdout);
  }

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

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

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

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

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

    std::cout << "  Sequential result: " << reference_result << std::endl;
    std::cout << "  Sequential time: " << stime << std::endl;
    std::cout << "  Parallel result: " << result << std::endl;
    std::cout << "  Parallel time: " << ptime << std::endl;
    std::cout << "  Speedup: " << stime / ptime << std::endl;

    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_2) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
  if (rank == 0) {
    printf("  f(x,y,z) = log10(2 * x * x) + sqrt(z) + 5 * y\n");
    fflush(stdout);
  }

  const std::function<double(std::vector<double>)> f =
      [](std::vector<double> vec) {
        double x = vec[0];
        double y = vec[1];
        double z = vec[2];
        return log10(2 * x * x) + sqrt(z) + 5 * y;
      };

  std::vector<std::pair<double, double>> limits({{4, 10}, {1, 2}, {2, 5}});
  std::vector<int> n({10, 10, 10});

  double result = getParallelSimpson(f, limits, n);

  if (rank == 0) {
    double reference_result = getSequentialSimpson(f, limits, n);
    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_3) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    printf("  f(x,y,z,t) = x + y + z + t\n");
    fflush(stdout);
  }

  const std::function<double(std::vector<double>)> f =
      [](std::vector<double> vec) {
        double x = vec[0];
        double y = vec[1];
        double z = vec[2];
        return x * y * z;
      };

  std::vector<std::pair<double, double>> limits({{4, 10}, {1, 2}, {4, 5}});
  std::vector<int> n({10, 5, 5});

  double result = getParallelSimpson(f, limits, n);

  if (rank == 0) {
    double reference_result = getSequentialSimpson(f, limits, n);
    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_4) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    printf("  f(x,y,z) = exp(x) - sqrt(10) * 5 * sin(y) + cos(-2 * z * z)\n");
```

```
    fflush(stdout);
  }

  const std::function<double(std::vector<double>)> f =
      [](std::vector<double> vec) {
        double x = vec[0];
        double y = vec[1];
        double z = vec[2];
        return exp(x) - sqrt(10) * 5 * sin(y) + cos(-2 * z * z);
      };

  std::vector<std::pair<double, double>> limits({{4, 10}, {1, 2}, {0, 5}});
  std::vector<int> n({10, 5, 5});

  double result = getParallelSimpson(f, limits, n);

  if (rank == 0) {
    double reference_result = getSequentialSimpson(f, limits, n);
    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_5) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    printf("  f(x,y,z,t) = cos(5 * x) + exp(y) + 2.9 * sin(z) - t * t\n");
    fflush(stdout);
  }

  const std::function<double(std::vector<double>)> f =
      [](std::vector<double> vec) {
        double x = vec[0];
        double y = vec[1];
        double z = vec[2];
        double t = vec[3];
        return cos(5 * x) + exp(y) + 2.9 * sin(z) - t * t;
      };

  std::vector<std::pair<double, double>> limits(
      {{4, 10}, {1, 2}, {1, 5}, {6, 10}});
  std::vector<int> n({5, 5, 3, 2});

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

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

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

    std::cout << "  Sequential result: " << reference_result << std::endl;
    std::cout << "  Sequential time: " << stime << std::endl;
    std::cout << "  Parallel result: " << result << std::endl;
```

```
    std::cout << "␣␣Parallel␣time:␣" << ptime << std::endl;
    std::cout << "␣␣Speedup:␣" << stime / ptime << std::endl;

    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

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();
}
```

main.cpp

```
// Copyright 2021 Shelepin Nikita
#include <gtest/gtest.h>

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

#include "./simpson.h"

TEST(SIMPSON_METHOD, TEST_FUNCTION_1) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    printf("␣␣f(x,y)␣=␣x␣*␣x␣-␣2␣*␣y\n");
    fflush(stdout);
  }

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

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

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

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

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

    std::cout << "  Sequential result: " << reference_result << std::endl;
    std::cout << "  Sequential time: " << stime << std::endl;
    std::cout << "  Parallel result: " << result << std::endl;
    std::cout << "  Parallel time: " << ptime << std::endl;
    std::cout << "  Speedup: " << stime / ptime << std::endl;

    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_2) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    printf("  f(x,y,z) = log10(2 * x * x) + sqrt(z) + 5 * y\n");
    fflush(stdout);
  }

  const std::function<double(std::vector<double>)> f =
      [](std::vector<double> vec) {
        double x = vec[0];
        double y = vec[1];
        double z = vec[2];
        return log10(2 * x * x) + sqrt(z) + 5 * y;
      };

  std::vector<std::pair<double, double>> limits({{4, 10}, {1, 2}, {2, 5}});
  std::vector<int> n({10, 10, 10});

  double result = getParallelSimpson(f, limits, n);

  if (rank == 0) {
    double reference_result = getSequentialSimpson(f, limits, n);
    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_3) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    printf("  f(x,y,z,t) = x + y + z + t\n");
    fflush(stdout);
  }

  const std::function<double(std::vector<double>)> f =
      [](std::vector<double> vec) {
        double x = vec[0];
        double y = vec[1];
        double z = vec[2];
        return x * y * z;
```

```
        };

    std::vector<std::pair<double, double>> limits({{4, 10}, {1, 2}, {4, 5}});
    std::vector<int> n({10, 5, 5});

    double result = getParallelSimpson(f, limits, n);

    if (rank == 0) {
      double reference_result = getSequentialSimpson(f, limits, n);
      double error = 0.0001;
      ASSERT_NEAR(result, reference_result, error);
    }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_4) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
      printf("  f(x,y,z) = exp(x) - sqrt(10) * 5 * sin(y) + cos(-2 * z * z)\n");
      fflush(stdout);
    }

    const std::function<double(std::vector<double>)> f =
        [](std::vector<double> vec) {
          double x = vec[0];
          double y = vec[1];
          double z = vec[2];
          return exp(x) - sqrt(10) * 5 * sin(y) + cos(-2 * z * z);
        };

    std::vector<std::pair<double, double>> limits({{4, 10}, {1, 2}, {0, 5}});
    std::vector<int> n({10, 5, 5});

    double result = getParallelSimpson(f, limits, n);

    if (rank == 0) {
      double reference_result = getSequentialSimpson(f, limits, n);
      double error = 0.0001;
      ASSERT_NEAR(result, reference_result, error);
    }
}

TEST(SIMPSON_METHOD, TEST_FUNCTION_5) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
      printf("  f(x,y,z,t) = cos(5 * x) + exp(y) + 2.9 * sin(z) - t * t\n");
      fflush(stdout);
    }

    const std::function<double(std::vector<double>)> f =
        [](std::vector<double> vec) {
          double x = vec[0];
          double y = vec[1];
          double z = vec[2];
          double t = vec[3];
          return cos(5 * x) + exp(y) + 2.9 * sin(z) - t * t;
        };
```

```
  std::vector<std::pair<double, double>> limits(
      {{4, 10}, {1, 2}, {1, 5}, {6, 10}});
  std::vector<int> n({5, 5, 3, 2});

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

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

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

    std::cout << "  Sequential result: " << reference_result << std::endl;
    std::cout << "  Sequential time: " << stime << std::endl;
    std::cout << "  Parallel result: " << result << std::endl;
    std::cout << "  Parallel time: " << ptime << std::endl;
    std::cout << "  Speedup: " << stime / ptime << std::endl;

    double error = 0.0001;
    ASSERT_NEAR(result, reference_result, error);
  }
}

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();
}
```