МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Нижегородский государственный университет им. Н.И. Лобачевского»
Институт информационных технологий, математики и механики

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Решение многомерных интегралов методом Симпсона»

Выполнила:
студентка группы 381706-2
Савосина Александра Дмитриевна
_____ Подпись

Научный руководитель:
доцент кафедры МОСТ
Сысоев А.В.
_____ Подпись

Нижний Новгород
2019

Оглавление

Введение

В современном мире многие вычисления уже не производятся вручную – для этого существует специальная вычислительная техника, ведь проводимые эксперименты становятся сложнее: объём выборки и трудоёмкость алгоритмов уже не позволяют производить расчёты на бумаге – такой объём работы может оказаться просто не под силу даже группе людей, к тому же в таком случае нельзя гарантировать правильность подсчётов наверняка, так как имеет место быть человеческих фактор. Для больших объёмов вычислений уже давно используют специальную технику и имеющиеся алгоритмы, однако даже при использовании производящих подсчёты устройств может пройти достаточно много времени до получения результата. А если это не научный эксперимент, результаты которого в дальнейшем станут почвой для размышлений больших умов, а программа, работающая в реальном времени, то вопрос скорости получения ответа встаёт более остро. Здесь нам на помощь приходит понятие оптимизации.

Оптимизация — процесс максимизации выгодных характеристик, соотношений (например, оптимизация производственных процессов и производства), и минимизации расходов.

В нашем случае речь идёт о минимизации времени вычислений путём распараллеливания программы на некоторое количество процессов, так как время в контексте поставленной задачи – самый ценный ресурс.

Реализация метода Симпсона включает в себя рекурсивный алгоритм, необходимый для осуществления погружения на уровни при вычислении многомерного интеграла, являющийся частью системы распараллеливания интеграла первого уровня на заданное число процессов.

Постановка задачи

Основной задачей проекта является изучение метода Симпсона для решения множественных интегралов различной сложности, а также разработка программы, решающей подобные определённые интегралы.

Декомпозиция основной задачи:

1. Реализация последовательного алгоритма метода Симпсона для получения численного решения множественного определённого интеграла
2. Реализация параллельного алгоритма метода Симпсона для получения численного решения множественного определённого интеграла. Данный алгоритм подразумевает распараллеливание задачи на заданное число процессов с целью увеличения производительности
3. Тестирование работоспособности написанных алгоритмов посредствам тестов, написанных с использованием Google C++ Testing Framework
4. Проведение анализа и сравнение времени работы последовательного и параллельного алгоритмов, осуществление оценки программы на основе произведённых экспериментов с помощью расчёта ускорения.

Описание алгоритма

Суть метода Симпсона заключается в приближении подынтегральной функции на отрезке **[a; b]** интерполяционным многочленом второй степени $\mathbf{p_2(x)}$, то есть приближение графика функции на отрезке параболой.

Пусть функция **y = f(x)** непрерывна на отрезке **[a; b]** и нам требуется вычислить определенный интеграл $\int_a^b \mathbf{f(x)\,dx}$.

Разобьем отрезок **[a; b]** на **n** элементарных отрезков $\mathbf{[x_{2i-2}; x_{2i}]}$, **i = 1, 2, …, n** длины **2h = (b - a) / n** точками $\mathbf{a = x_0 < x_2 < \ldots < x_{2n-2} < x_n = b}$.

Пусть точки $\mathbf{x_{2n-1}}$, **i = 1, 2, …, n** являются серединами отрезков $\mathbf{[x_{2i-2}; x_{2i}]}$, **i = 1, 2, …, n** соответственно.

В этом случае все "узлы" определяются из равенства $\mathbf{x_i = a + ih}$, **i = 0, 1, 2, …, n**

На каждом интервале $\mathbf{[x_{2i-2}; x_{2i}]}$, **i = 1, 2, ..., n** подынтегральная функция приближается квадратичной параболой $\mathbf{y = a_i x^2 + b_i x + c_i}$, проходящей через точки $\mathbf{(x_{2i-2};f(x_{2i-2}))}$, $\mathbf{(x_{2i-1};f(x_{2i-1}))}$, $\mathbf{(x_{2i};f(x_{2i}))}$. Отсюда и название метода - метод парабол.

Это делается для того, чтобы в качестве приближенного значения определенного интеграла $\int_{x2i-2}^{x2i} \mathbf{f(x)\,dx}$ взять $\int_{x2i-2}^{x2i} \mathbf{ai\ x2 + bi\ x + ci\ dx}$, который мы можем вычислить по формуле Ньютона-Лейбница. В этом и заключается суть метода парабол.

Геометрически это выглядит так:

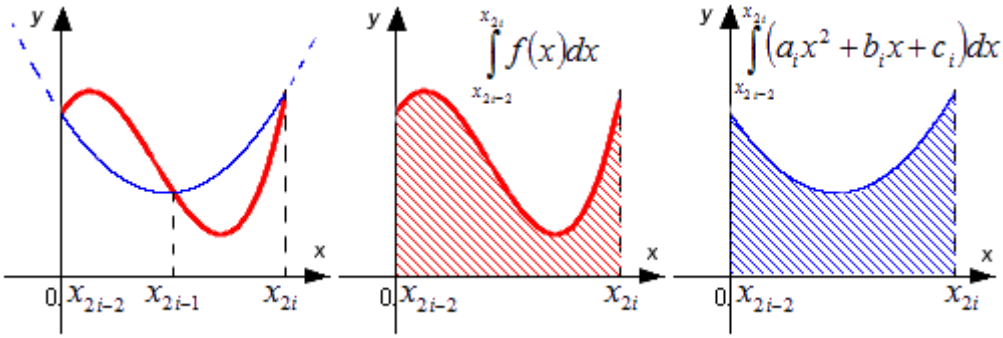

Рисунок 1. Геометрическое представление метода парабол.

Схема распараллеливания

В лабораторной работе осуществляется распараллеливание интеграла первого уровня на заданное число процессов. В начале имеется область интегрирование, заданная отрезками $\mathbf{[a_i; b_i], i = 1, \ldots, n}$, количество которых совпадает с количеством интегралов **n** в многомерном интеграле. Также в примере изначально задаётся параметр, определяющий разбиение данного отрезка; эти данные в алгоритм передаются в алгоритм виде литеры и не имеют отдельной переменной для хранения, так как для каждого примера имеется своё индивидуальное значение.

Далее в зависимости от количества процессов данные о разбиении первого интеграла передаются в разные процессы для вычисления значений на заданных промежутках, после чего данные суммируются и становятся итоговым ответом. На уровнях интеграла > 2 разбиение не производится, поскольку предоставленный алгоритм уже обеспечивает оптимизацию решения по времени работы программы. Погружение производится за счёт рекурсии, которая позволяет производить вычисления для множественного интеграла. Остановка программы и возвращение результата осуществляются при прохождении процессом всех уровней интегрирования и завершения рекурсии.

Описание программной реализации

Для решения поставленной задачи используется три основные функции и одна вспомогательная:

- double calculateIntegral(const std::vector<std::pair<double, double>>& _scope, const size_t& _n, const std::function<double(const std::vector<double>&)>& f, const size_t& _nRec = 2)
  – параллельное вычисление интеграла
- double calculateIntegralSequential(const std::vector<std::pair<double, double>>& _scope, const size_t& _n, const std::function<double(const std::vector<double>&)>& f, const size_t& _nRec = 2)
  – последовательное вычисление интеграла
- double calculateRecusionLevel(std::vector<double> _fixedVar, size_t _levelNumber, const std::vector<std::pair<double, double>>& _scope, const std::function<double(const std::vector<double>&)>& f, const size_t& _n)
  – функция, вычисляющая уровень погружения (или уровень интеграла, на котором производятся вычисления)
- void scopeCheck(const std::vector<std::pair<double, double>>& scope)
  – проверяет правильность границ интегрирования (чтобы верхняя граница не оказалась меньше нижней)

Корректность

Основным инструментом работоспособности программы являются тесты, разработанные при использовании Google C++ Testing Framework.

Правильность получаемых результатов проверяется на пяти различных функциях разной сложности.

Примеры подынтегральных функций некоторых из них:

sin(x + y)dxdy, x - (0, pi/2), y - (0, pi/4)

sin(x)ycos(z)dxdydz, x - (0, 1), y - (-2, 2), z - (3, 7)

x^2+y+z+c^2+s^3, x - (0, 1), y - (1, 2), z - (-1, 1), c - (-2, 2), s - (0, 1)

Все тесты проходят проверку, что является доказательством корректной работы программы

Результаты экспериментов

В таблицах приведены результаты времени вычислений в зависимости от количества процессов и сложности примеров.

Пример 1	Число разбиений = 2048			(0, pi/2), (0, pi/4)		**sin(x + y) dx dy**	
	последовательно	1 попытка	2 попытка	3 попытка	4 попытка	5 попытка	среднее
4 процесса	3.01043	1.59008	1.60491	1.57133	1.6008	1.59154	**1.591732**
3 процесса	3.01043	2.5741	2.55274	2.55274	2.55274	2.57073	**2.050062**
2 процесса	3.01043	2.94741	3.02116	2.98953	3.03401	2.99584	**2.99759**

Таблица 1. Данные по первому примеру в секундах

Пример 2	Число разбиений = 200			(0, 1), (-2, 2), (3, 7)		**sin(x) y cos(z)dxdydz**	
	последовательно	1 попытка	2 попытка	3 попытка	4 попытка	5 попытка	среднее
2 процесса	7.14103	6.46929	6.90064	6.87955	6.85982	6.87668	**6.79796**
3 процесса	7.14103	4.28247	4.49798	4.49941	4.5008	4.4975	**4.455632**
4 процесса	7.14103	3.43169	3.38789	3.38789	3.57926	3.47251	**3.451846**

Таблица 2. Данные по второму примеру в секундах

Пример 3	Число разбиений = 16		(0, 1), (1, 2), (-1, 1), (-2, 2), (0, 1)			**x^2+y+z+c^2+s^3**	
	последовательно	1 попытка	2 попытка	3 попытка	4 попытка	5 попытка	среднее
2 процесса	9.93028	5.49439	5.47534	5.50913	5.50732	5.54199	**5.505634**
3 процесса	9.93028	5.9047	5.88409	5.88613	5.88279	5.88328	**5.888198**
4 процесса	9.93028	4.41494	4.3233	4.25916	4.48253	4.36258	**4.368502**

Таблица 3. Данные по третьему примеру в секундах

Функция из таблицы	Посл. время	Параллельное время					
		2		3		4	
		время	ускорение	время	ускорение	время	ускорение
3	9.93028	5.505634	**1.803658**	5.888198	**1.686472**	4.368502	**2.273155**
2	7.14103	6.79796	**1.050466**	4.455632	**1.602697**	3.451846	**2.068756**
1	3.01043	2.99759	**1.004283**	2.050062	**1.468458**	1.591732	**1.891292**

Таблица 4. Сравнение данных и расчёт ускорения: время в секундах, ускорение в условных единицах

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 уравнений. Причём чем больше процессов, тем быстрее работает параллельная программа.

Вывод

Результатом лабораторной работы стала реализация метода Симпсона для решения множественного интеграла в виде параллельных и последовательных вычислений, в том числе выполнение всех поставленных подзадач.

Основным результатом работы программы стали данные, подтверждающие превосходство параллельных вычислений, а также зависимость скорости работы программы от количества запущенных процессов. С увеличением числа процессов время работы параллельной программы сильнее отличается от времени работы последовательной. Это демонстрирует ускорение, формула, для вычисления которого – время работы последовательного алгоритма, делённое на время работы параллельного. Таблица 4 наглядно демонстрирует увеличение значения ускорения с увеличением числа процессов. Это обусловлено распределением данных между процессами и уменьшением объёма работы каждого процесса в отдельности, что при учёте одновременной их работы, сокращает время ожидания получения ответа при параллельных вычислениях.

Время работы программы также зависит от сложности заданной функции, количества разбиений и уровней интегралов (количестве поставленных интегралов в задании). Однако анализ работы проводился при константных перечисленных параметров с изменением лишь числа процессов.

Корректность получаемых результатов проверяется на пяти заданных примерах с помощью тестов, написанных при использовании Google C++ Testing Framework. Это свидетельствует о правильной реализации алгоритма метода Симпсона.

Литература

1. Самарский А.А.
   «Введение в численные методы»
   Книга написана на основе курса лекций, читавшихся автором па
   факультете вычислительной математики и кибернетики МГУ
   1982 год
   272 страницы
2. Самарский А. А., Гулин А. В.
   «Численные методы»
   Москва «Наука»
   Главная редакция физико-математической литературы
   1989 год
   432 страницы

Приложение

simpson_method.h

```
// Copyright 2019 Savosina
#ifndef MODULES_TASK_3_SAVOSINA_A_SIMPSON_METHOD_SIMPSON_METHOD_H_
#define MODULES_TASK_3_SAVOSINA_A_SIMPSON_METHOD_SIMPSON_METHOD_H_

#include <vector>
#include <utility>
#include <functional>

double calculateIntegral(const std::vector<std::pair<double, double>>& _scope, const size_t& _n,
const std::function<double(const std::vector<double>&)>& f, const size_t& _nRec = 2);

double calculateIntegralSequential(const std::vector<std::pair<double, double>>& _scope, const size_t& _n,
const std::function<double(const std::vector<double>&)>& f, const size_t& _nRec = 2);

double calculateRecusionLevel(std::vector<double> _fixedVar, size_t _levelNumber,
const std::vector<std::pair<double, double>>& _scope,
const std::function<double(const std::vector<double>&)>& f, const size_t& _n);

void scopeCheck(const std::vector<std::pair<double, double>>& scope);

#endif  // MODULES_TASK_3_SAVOSINA_A_SIMPSON_METHOD_SIMPSON_METHOD_H_
```

main.cpp

```
// Copyright 2019 Savosina
#define _USE_MATH_DEFINES

#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <cmath>
#include <utility>
#include <vector>
#include "./simpson_method.h"

TEST(Simpson_Method_MPI, Test_First_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::function<double(const std::vector<double>&)> func = [](const std::vector<double>& vec){
        return sin(vec[0]+vec[1]);};
    std::vector<std::pair<double, double>> scope = { {0, M_PI_2}, {0, M_PI_4}};  // expected 1.00028
    scopeCheck(scope);
    double res = calculateIntegral(scope, 100, func, 100);
    if (rank == 0) {
        ASSERT_LE(std::abs(res - 1.0), 0.01);
    }
}
```

```
TEST(Simpson_Method_MPI, Test_Second_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::function<double(const std::vector<double>&)> func = [](const std::vector<double>& vec){
        return sin(vec[0])*vec[1]*cos(vec[2]);};
    std::vector<std::pair<double, double>> scope = { {0, 1}, {-2, 2}, {3, 7} };  // expected ~ 0
    scopeCheck(scope);
    double res = calculateIntegral(scope, 100, func, 100);
    if (rank == 0) {
        ASSERT_LE(std::abs(res - 0.0), 0.01);
    }
}

TEST(Simpson_Method_MPI, Test_Third_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::function<double(const std::vector<double>&)> func = [](const std::vector<double>& vec){
        return std::sqrt(1 + 2*std::pow(vec[0], 2) - std::pow(vec[0], 3));};
    std::vector<std::pair<double, double>> scope = { {1.2, 2} };  // expected 1.09
    scopeCheck(scope);
    double res = calculateIntegral(scope, 100, func, 100);
    if (rank == 0) {
        ASSERT_LE(std::abs(res - 1.09), 0.01);
    }
}
TEST(Simpson_Method_MPI, Test_Fourth_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::function<double(const std::vector<double>&)> func = [](const std::vector<double>& vec){
    return vec[0] / (std::pow(vec[0], 4) + 4);};
    std::vector<std::pair<double, double>> scope = { {0, 5} };  // expected 0.377
    scopeCheck(scope);
    double res = calculateIntegral(scope, 100, func, 100);
    if (rank == 0) {
        ASSERT_LE(std::abs(res - 0.377), 0.01);
    }
}

TEST(Simpson_Method_MPI, Test_Fifth_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::function<double(const std::vector<double>&)> func = [](const std::vector<double>& vec){
        return std::pow(vec[0], 2) + vec[1] + vec[2] + std::pow(vec[3], 2) + std::pow(vec[4], 3);};
    std::vector<std::pair<double, double>> scope = { {0, 1}, {1, 2}, {-1, 1}, {-2, 2}, {0, 1} };  // expected ~ 27
    scopeCheck(scope);
    double res = calculateIntegral(scope, 5, func, 5);
    if (rank == 0) {
        ASSERT_LE(std::abs(res - 27), 1);
    }
}
```

```
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();
}
```

simpson_method.cpp

```
// Copyright 2019 Savosina
#include <mpi.h>
#include <utility>
#include <vector>
#include <stdexcept>
#include "../../../modules/task_3/savosina_a_simpson_method/simpson_method.h"

double calculateIntegral(const std::vector<std::pair<double, double>>& _scope, const size_t& _n,
const std::function<double(const std::vector<double>&)>& f, const size_t& _nRec) {
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double res = 0;
    if (size == 1) {
        res = calculateIntegralSequential(_scope, _n, f, _nRec);
    } else {
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Status status;
        int levelNumber = 0;
        if (rank == 0) {
            double h = (_scope[levelNumber].second - _scope[levelNumber].first) / (2 * _n);
            std::vector<double> funcResults;
            int rankItr = 1;
            double mes;
            if (size > static_cast<int>(_n * 2)) {
                for (double i = _scope[levelNumber].first; i <= _scope[levelNumber].second; i = i + h) {
                    MPI_Send(&i, 1, MPI_DOUBLE, rankItr, 1, MPI_COMM_WORLD);
                    rankItr++;
                }
                for (int i = 1; i < rankItr; i++) {
```

```
            MPI_Recv(&mes, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, &status);
            funcResults.push_back(mes);
        }
    } else {
        int needToSolve = _n * 2 / size;
        double iter = _scope[levelNumber].first;
        iter += needToSolve * h;
        for (rankItr = 1; rankItr < size; rankItr++)
            for (int i = 0; i < needToSolve; i++) {
                MPI_Send(&iter, 1, MPI_DOUBLE, rankItr, 1, MPI_COMM_WORLD);
                iter += h;
            }
        for (rankItr = 1; rankItr <= static_cast<int>(_n * 2 % size); rankItr++) {
            MPI_Send(&iter, 1, MPI_DOUBLE, rankItr, 1, MPI_COMM_WORLD);
            iter += h;
        }
        for (double i = _scope[levelNumber].first; i < _scope[levelNumber].first + needToSolve * h; i = i + h) {
            std::vector<double> fixedVar = { i };
            if (static_cast<int>(_scope.size()) == levelNumber + 1) {
                funcResults.push_back(f(fixedVar));
            } else {
                funcResults.push_back(calculateRecusionLevel(fixedVar, levelNumber + 1, _scope, f, _nRec));
            }
        }
        for (rankItr = 1; rankItr < size; rankItr++)
            for (int i = 0; i < needToSolve; i++) {
                MPI_Recv(&mes, 1, MPI_DOUBLE, rankItr, 1, MPI_COMM_WORLD, &status);
                funcResults.push_back(mes);
            }
        for (rankItr = 1; rankItr <= static_cast<int>(_n * 2 % size); rankItr++) {


                    MPI_Recv(&mes, 1, MPI_DOUBLE, rankItr, 1, MPI_COMM_WORLD, &status);
                    funcResults.push_back(mes);
                }
            }
            mes = -1;
            for (rankItr = 1; rankItr < size; rankItr++) {
                MPI_Send(&mes, 1, MPI_DOUBLE, rankItr, 2, MPI_COMM_WORLD);
            }
            double tempRes = funcResults[0];
            res = tempRes;
            tempRes = 0;
            for (size_t i = 1; i <= _n; i++)
                tempRes += funcResults[2*i - 1];
            res += 4 * tempRes;
            tempRes = 0;
            for (size_t i = 1; i <= _n - 1; i++)
                tempRes += funcResults[2*i];
            res += 2 * tempRes;
            res += funcResults[2 * _n - 1];
            res = res * h / 3;
        } else {
```

```
            bool terminate = false;
            while (!terminate) {
                double mes;
                MPI_Recv(&mes, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                if (status.MPI_TAG == 2) {
                    terminate = true;
                } else {
                    std::vector<double> fixedVar = { mes };
                    if (static_cast<int>(_scope.size()) == levelNumber + 1) {
                        res = f(fixedVar);
                    } else {
                        res = calculateRecusionLevel(fixedVar, levelNumber + 1, _scope, f, _nRec);
                    }
                    MPI_Send(&res, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
                }
            }
        }
    }
    return res;
}

double calculateIntegralSequential(const std::vector<std::pair<double, double>>& _scope, const size_t& _n,
const std::function<double(const std::vector<double>&)>& f, const size_t& _nRec) {
    int levelNumber = 0;
    double h = (_scope[levelNumber].second - _scope[levelNumber].first) / (2 * _n);
    std::vector<double> funcResults;
    for (double i = _scope[levelNumber].first; i <= _scope[levelNumber].second; i = i + h) {
        std::vector<double> fixedVar = { i };
        if (static_cast<int>(_scope.size()) == levelNumber + 1) {
            funcResults.push_back(f(fixedVar));
        } else {
            funcResults.push_back(calculateRecusionLevel(fixedVar, levelNumber + 1, _scope, f, _nRec));
        }
    }
    double tempRes = funcResults[0];
    double res = tempRes;
    tempRes = 0;
    for (size_t i = 1; i <= _n; i++)
        tempRes += funcResults[2*i - 1];
    res += 4 * tempRes;
    tempRes = 0;
    for (size_t i = 1; i <= _n - 1; i++)
        tempRes += funcResults[2*i];
    res += 2 * tempRes;
    res += funcResults[2 * _n - 1];
    res = res * h / 3;
    return res;
}

void scopeCheck(const std::vector<std::pair<double, double>>& scope) {
    for (auto iter = scope.begin(); iter != scope.end(); iter++) {
        if (iter->first > iter->second)
            throw std::runtime_error("Invalide scope");
    }
}
```