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 проектно-технологической практике

«Параллельные версии волновой схемы решения задачи Дирихле для уравнения Пуассона»

Выполнил:
студент группы 381908-4
Китаев П.И.

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

Нижний Новгород
2022

Оглавление

Введение

Задача Дирихле для уравнения Пуассона является одной из классических задач математической физики. Для решения уравнений с частными производными, как правило, используются сеточные методы. На практике широко применяются итерационные методы, при которых вычислительная схема описывает, как следующее состояние сетки зависит от предыдущего. В результате счета на компьютере получается приближенное решение уравнений с частными производными.

Решение задачи Дирихле для уравнения Пуассона сопровождается большим размером сетки и, зачастую, большим количеством итераций, в связи с этим последовательные алгоритмы выполняются достаточно долго. Данную проблему можно решить с помощью параллельных алгоритмов.

Постановка задачи

Необходимо реализовать последовательный метод, параллельный метод с использованием технологии Intel TBB и параллельный метод с использованием технологии std::thread для волновой схемы решения задачи Дирихле для уравнения Пуассона, определяемая как задача нахождения функции *u=u(x,y)*, удовлетворяющей в области определения *D* уравнению:

$$\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = f(x,y), g(x,y) \in D$$

и принимающей значения *g(x,y)* на границе D^0 области *D* (*f* и *g* являются функциями, задаваемыми при постановке задачи).

В качестве области задания функции будет использоваться единичный квадрат:

$$D = \{(x,y) \in R^2 : 0 \leq x, u \leq 1\}$$

С помощью метода Гаусса-Зейделя для проведения итераций уточнения следует использовать правило:

$$u^k_{i,j} = 0.25(u^k_{i-1,j} + u^{k-1}_{i+1,j} + u^k_{i,j-1} + u^k_{i,j+1} - h^2 f_{i,j})$$

по которому очередное *k-ое* приближение значения u_{ij} вычисляется по последнему *k-ому* приближению значений $u_{i-1,j}$ и $u_{i,j-1}$ и предпоследнему *(k-1)-ому* приближению значений $u_{i+1,j}$ и $u_{i,j+1}$.

Выполнение итераций продолжается до тех пор, пока получаемые в результате итераций изменения значений не станут меньше некоторой заданной величины (требуемой точности вычислений).

При разработке программ полагается, что функция *f* тождественно равна нулю, т.е. $f(x,y) \equiv 0$

Также следует использовать модульное тестирование Google Test, систему сборки проектов CMake и распределенную систему контроля версий Git.

Описание алгоритма

Функция (не зависимо от типа алгоритма) принимает в качестве входных параметров указатель на линейную матрицу *MxM* с начальными значениями, требуемую точность и размер матрицы. Затем, производятся следующие действия:

1. Объявляются необходимые переменные и выделяется память для массива, который предназначен для хранения погрешностей в пределах одной итерации,
2. Производятся вычисления нарастающей волной с фиксацией погрешностей,
3. Производятся вычисления затухающей волной с фиксацией погрешностей,
4. Из массива погрешностей извлекается погрешность с максимально большим значением,
5. Пункты 2, 3, 4 повторяются до тех пор, пока погрешность не станет равной, либо меньше требуемой.

Параллельные алгоритмы не отличаются от последовательного, за исключением пунктов 2 и 3, которые реализованы с использованием параллельных технологий Intel TBB и std::thread.

Описание схемы распараллеливания

Основные участки алгоритма, где распараллеливание приносит результат - это вычисления с нарастанием волны и затуханием.

При использовании технологии Intel TBB объявляется параллельная секция с указанием итерационного пространства. Используемая библиотека сама распределяет количество итераций между потоками.

В случае с std::thread за распределение элементов матрицы между потоками ответственен программист. Перед тем, как запустить поток для вычисления, происходит расчет количества итераций (индекс начала, и конца) для каждого потока на основании общего количества потоков. В ситуации, если не получается распределить элементы матрицы на равное количество между потоками, первые потоки получают больше элементов для вычисления.

Описание программной реализации

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

В заголовочном файле объявлены прототипы функций для последовательного метода, методов на основе Intel TBB, std::thread, а так же прототип функции для заполнения матрицы начальными значениями.

Прототип функции для заполнения матрицы начальными значениями:

```
void FillingTheMatrix(double* matrix, int size);
```

Принимает указатель на линейный массив квадратной матрицы и ее размер.

Прототип последовательного метода:

```
void SequentialAlg(double* matrix, int size, double eps);
```

Прототип параллельного метода с использованием технологии Intel TBB:

```
void ParallelAlgTBB(double* matrix, int size, double eps);
```

Прототип параллельного метода с использованием технологии std::thread:

```
void ParallelAlgSTD(double* matrix, int size, double eps);
```

Все основные функции принимают в качестве входных параметров указатель на массив с начальными элементами линейной матрицы, размер матрицы и требуемую точность.

Подтверждение корректности

Каждый реализованный метод подвергался комплексному тестированию, в том числе с использованием модульной системы Google Test. В частности были реализованы такие тесты, как:

- создание матриц с начальными элементами различных размеров для дальнейшего использование их в качестве входного параметра основных функций,
- использование последовательного алгоритма с различными размерами матриц и заданной точностью,
- использование параллельного алгоритма на основе Intel TBB с различными размерами матриц и заданной точностью,
- использование параллельного алгоритма на основе std::thread с различными размерами матриц и заданной точностью.

В конечном итоге на основании полученных результатов тестов сделан вывод о работоспособности и корректности реализации всех методов.

Результаты экспериментов

Тестирование всех методов и вычислительные эксперименты для сравнения эффективности работ параллельных алгоритмов относительно последовательного метода проводились на ЭВМ с характеристиками:

- Процессор: Intel Core i5-10210U 1.6 GHz, имеющий 8 потоков,
- Оперативная память: 8 ГБ,
- Операционная система: Windows 10 Home x64.

Эксперименты с фиксацией времени выполнения проводились на 8 потоках и в конфигурации решения Release.

Результаты экспериментов, средние значения за 10 измерений:

Алгоритм	Время работы, в секундах	Разница, в секундах
Sequential	0.133	
Parallel TBB	0.153	-0.02
Parallel std::thread	7.993	-7.86

Таблица 1: Результаты вычислительных экспериментов с матрицей 100x100, с точностью 0.01

Алгоритм	Время работы, в секундах	Разница, в секундах
Sequential	18.914	
Parallel TBB	10.785	8.129
Parallel std::thread	18.415	0.499

Таблица 2: Результаты вычислительных экспериментов с матрицей 1000x1000, с точностью 0.01

Алгоритм	Время работы, в секундах	Разница, в секундах
Sequential	42.628	
Parallel TBB	31.771	10.858
Parallel std::thread	34.262	8.366

Таблица 3: Результаты вычислительных экспериментов с матрицей 1500x1500, с точностью 0.01

Алгоритм	Время работы, в секундах	Разница, в секундах
Sequential	81.464	
Parallel TBB	59.032	22.432
Parallel std::thread	52.447	29.017

Таблица 4: Результаты вычислительных экспериментов с матрицей 2000x2000, с точностью 0.01

Выводы из результатов экспериментов

На основе результатов экспериментов можно сделать вывод о том, что параллельные реализации эффективны только тогда, когда сэкономленное время на вычислениях превышает затраты на накладные расходы, например такие, как передача данных между потоками, сбор данных в главный поток. Другими словами, не имеет смысла распараллеливать участки кода, где не предвидятся масштабные вычисления.

В случае с матрицами, размерами 1500x1500 элементов и более, время на выполнение вычислений значительно сокращается. При этом, в случаях с достаточно большим размером матрицы, параллельный метод на основе Intel TBB уступает методу, реализованному с использованием std::thread.

Заключение

В процессе выполнения данной лабораторной работы были изучены и применены на практике такие методы распараллеливания программ, как Intel TBB и std::thread. Получены практические навыки использования системы сборки проектов CMake и системы контроля версий Git.

Были разработаны и реализованы последовательный и параллельные алгоритмы решения задачи Дирихле для уравнения Пуассона. Разработанные тесты и проведенные вычислительные эксперименты подтвердили корректность реализованной программы.

Литература

1. Учебный курс "Введение в методы параллельного программирования". Раздел "Принципы разработки параллельных методов"/ сост.: В.П. Гергель - Нижний Новгород, 2007
2. А.В. Сысоев, И.Б. Мееров, А.А. Сиднев «Средства разработки параллельных программ для систем с общей памятью. Библиотека Intel Threading Building Blocks». Нижний Новгород, 2007, 128 с.
3. А.В. Сысоев, И.Б. Мееров, А.Н. Свистунов, А.Л. Курылев, А.В. Сенин, А.В. Шишков, К.В. Корняков, А.А. Сиднев «Параллельное программирование в системах с общей памятью. Инструментальная поддержка». Нижний Новгород, 2007, 110 с.
4. Учебный курс «Технологии разработки параллельных программ» Раздел «Создание параллельной программы» Библиотека Intel Threading Building Blocks – краткое описание / сост.: А.А. Сиднев, А.В. Сысоев, И.Б. Мееров - Нижний Новгород, 2007

Приложение

Реализация основной программы. Файл dirichle.h

```
// Copyright 2022 Kitaev Pavel

#ifndef MODULES_VER_1_DIRICHLE_DIRICHLE_H_
#define MODULES_VER_1_DIRICHLE_DIRICHLE_H_

void PrintMatrix(double* matrix, int size);
void FillingTheMatrix(double* matrix, int size);
void SequentialAlg(double* matrix, int size, double eps);
void ParallelAlgTBB(double* matrix, int size, double eps);
void ParallelAlgSTD(double* matrix, int size, double eps);

#endif  // MODULES_VER_1_DIRICHLE_DIRICHLE_H_
```

Файл dirichle.cpp

```
// Copyright 2022 Kitaev Pavel

#include <tbb/tbb.h>
#include <iostream>
#include <cmath>
#include "../../../3rdparty/unapproved/unapproved.h"
#include "../../../modules/ver_1/dirichle/dirichle.h"

void PrintMatrix(double* matrix, int size) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      std::cout << matrix[i * size + j] << "␣";
    }

    std::cout << std::endl;
  }
}

void FillingTheMatrix(double* matrix, int size) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      if ((i == 0) || (i == size - 1) || (j == 0) || (j == size - 1)) {
        matrix[i * size + j] = 100;
      } else {
        matrix[i * size + j] = 0;
      }
    }
  }
}

void SequentialAlg(double* matrix, int size, double eps) {
  double* dm = new double[size]{ 0 };
  double dmax, temp, d;
  const int N = size - 2;

  do {
    dmax = 0;

    for (int wave = 1; wave < N + 1; wave++) {
      dm[wave-1] = 0;
      for (int i = 1; i < wave + 1; i++) {
        int j = wave + 1 - i;
        temp = matrix[size * i + j];

        matrix[size * i + j] = 0.25 * (matrix[size * i + j + 1] +
          matrix[size * i + j - 1] + matrix[size * (i + 1) + j] +
          matrix[size * (i - 1) + j]);

        d = fabs(temp - matrix[size * i + j]);
        if (dm[i - 1] < d) {
          dm[i - 1] = d;
        }
      }
    }

    for (int wave = N - 1; wave > 0; wave--) {
      for (int i = N - wave + 1; i < N + 1; i++) {
        int j = 2 * N - wave - i + 1;
```

```
        temp = matrix[size * i + j];

        matrix[size * i + j] = 0.25 * (matrix[size * i + j + 1] +
          matrix[size * i + j - 1] + matrix[size * (i + 1) + j] +
          matrix[size * (i - 1) + j]);

        d = fabs(temp - matrix[size * i + j]);
        if (dm[i - 1] < d) {
          dm[i - 1] = d;
        }
      }
    }

    for (int i = 0; i < size; i++) {
      if (dmax < dm[i]) {
        dmax = dm[i];
      }
    }
  } while (dmax > eps);

  delete[] dm;
}

void ParallelAlgTBB(double* matrix, int size, double eps) {
  double* dm = new double[size] { 0 };
  double dmax;
  const int N = size - 2;

  do {
    dmax = 0;

    tbb::parallel_for(tbb::blocked_range<int>(1, N + 1, 1),
      [&](const tbb::blocked_range<int>& range) {
        for (int wave = range.begin(); wave < range.end(); wave++) {
          dm[wave - 1] = 0;
          for (int i = 1; i < wave + 1; i++) {
            int j = wave + 1 - i;
            double temp = matrix[size * i + j];

            matrix[size * i + j] = 0.25 * (matrix[size * i + j + 1] +
              matrix[size * i + j - 1] + matrix[size * (i + 1) + j] +
              matrix[size * (i - 1) + j]);

            double d = fabs(temp - matrix[size * i + j]);

            if (dm[i - 1] < d) {
              dm[i - 1] = d;
            }
          }
        }
      });

    tbb::parallel_for(tbb::blocked_range<int>(0, N - 1, 1),
      [&](const tbb::blocked_range<int>& range) {
        for (int wave = range.end(); wave > range.begin(); wave--) {
          for (int i = N - wave + 1; i < N + 1; i++) {
            int j = 2 * N - wave - i + 1;
            double temp = matrix[size * i + j];

            matrix[size * i + j] = 0.25 * (matrix[size * i + j + 1] +
```

```
            matrix[size * i + j - 1] + matrix[size * (i + 1) + j] +
            matrix[size * (i - 1) + j]);

          double d = fabs(temp - matrix[size * i + j]);

          if (dm[i - 1] < d) {
            dm[i - 1] = d;
          }
        }
      }
    });

  for (int i = 0; i < size; i++) {
    if (dmax < dm[i]) {
      dmax = dm[i];
    }
  }
} while (dmax > eps);

delete[] dm;
}

void ParallelAlgSTD(double* matrix, int size, double eps) {
  const int th_num = std::thread::hardware_concurrency();
  std::thread* threads = new std::thread[th_num];

  double* dm = new double[size] { 0 };
  double dmax;

  const int delta = size / th_num;
  int residue = size % th_num;

  int i_start_inc, i_end_inc;
  int i_start_dec, i_end_dec;

  do {
    dmax = 0;
    i_start_inc = 1, i_end_inc = 0;
    i_start_dec = (size - 2) - 1, i_end_dec = (size - 2) - 1;

    for (int k = 0; k < th_num; k++) {
      i_end_inc += (delta);
      i_end_dec -= (delta);

      if (residue > 0) {
        i_end_dec--;

        i_end_inc++;
        residue--;
      }

      if (k == th_num - 1) {
        i_end_inc = size - 1;
        i_end_dec = size - 1;
      }

      threads[k] = std::thread([&](int i_start_inc, int i_end_inc,
        int i_start_dec, int i_end_dec) {
          for (int wave = i_start_inc; wave < i_end_inc; wave++) {
            dm[wave - 1] = 0;
```

```
            for (int i = 1; i < wave + 1; i++) {
              int j = wave + 1 - i;
              double temp = matrix[size * i + j];

              matrix[size * i + j] = 0.25 * (matrix[size * i + j + 1] +
                matrix[size * i + j - 1] + matrix[size * (i + 1) + j] +
                matrix[size * (i - 1) + j]);

              double d = fabs(temp - matrix[size * i + j]);

              if (dm[i - 1] < d) {
                dm[i - 1] = d;
              }
            }
          }

          for (int wave = i_start_dec; wave > i_end_dec; wave--) {
            for (int i = (size - 2) - wave + 1; i < (size - 2) + 1; i++) {
              int j = 2 * (size - 2) - wave - i + 1;

              double temp = matrix[size * i + j];

              matrix[size * i + j] = 0.25 * (matrix[size * i + j + 1] +
                matrix[size * i + j - 1] + matrix[size * (i + 1) + j] +
                matrix[size * (i - 1) + j]);

              double d = fabs(temp - matrix[size * i + j]);
              if (dm[i - 1] < d) {
                dm[i - 1] = d;
              }
            }
          }
        }, i_start_inc, i_end_inc, i_start_dec, i_end_dec);
      i_start_inc = i_end_inc;
      i_start_dec = i_end_dec;
    }

    for (int i = 0; i < size; i++) {
      if (dmax < dm[i]) {
        dmax = dm[i];
      }
    }

    for (int i = 0; i < th_num; i++) {
      threads[i].join();
    }
  } while (dmax > eps);

  delete[] threads;
  delete[] dm;
}
```

Файл main.cpp

```
// Copyright 2022 Kitaev Pavel

#include <gtest/gtest.h>
#include <tbb/tbb.h>
#include "../../../modules/ver_1/dirichle/dirichle.h"

TEST(Sequential, Test_Sequential_Alg_10x10) {
  int size = 10;
  double eps = 0.01;
  double* matrix = new double[size * size];

  FillingTheMatrix(matrix, size);
  ASSERT_NO_THROW(SequentialAlg(matrix, size, eps));

  delete[] matrix;
}

TEST(Sequential, Test_Sequential_Alg_100x100) {
  int size = 100;
  double eps = 0.01;
  double* matrix = new double[size * size];

  FillingTheMatrix(matrix, size);
  ASSERT_NO_THROW(SequentialAlg(matrix, size, eps));

  delete[] matrix;
}

TEST(TBB, Test_TBB_Alg_10x10) {
  int size = 10;
  double eps = 0.01;
  double* matrix = new double[size * size];

  FillingTheMatrix(matrix, size);
  ASSERT_NO_THROW(ParallelAlgTBB(matrix, size, eps));

  delete[] matrix;
}

TEST(TBB, Test_TBB_Alg_100x100) {
  int size = 100;
  double eps = 0.01;
  double* matrix = new double[size * size];

  FillingTheMatrix(matrix, size);
  ASSERT_NO_THROW(ParallelAlgTBB(matrix, size, eps));

  delete[] matrix;
}

TEST(STD, Test_STD_Alg_10x10) {
  int size = 10;
  double eps = 0.01;
  double* matrix = new double[size * size];

  FillingTheMatrix(matrix, size);
  ASSERT_NO_THROW(ParallelAlgSTD(matrix, size, eps));
```

```
  delete[] matrix;
}

TEST(STD, Test_STD_Alg_100x100) {
  int size = 100;
  double eps = 0.01;
  double* matrix = new double[size * size];

  FillingTheMatrix(matrix, size);
  ASSERT_NO_THROW(ParallelAlgSTD(matrix, size, eps));

  delete[] matrix;
}

/*
TEST(Time_test_1500_1500, Test_SEQ_TBB_STD_Alg) {
  int size = 1000;
  double eps = 0.01;

  double* matrix_seq = new double[size * size];
  FillingTheMatrix(matrix_seq, size);
  double start_seq = clock();
  SequentialAlg(matrix_seq, size, eps);
  double end_seq = clock();
  double time_seq =
  static_cast<float>(end_seq - start_seq) / CLOCKS_PER_SEC;
  delete[] matrix_seq;

  double* matrix_tbb = new double[size * size];
  FillingTheMatrix(matrix_tbb, size);
  double start_tbb = clock();
  ParallelAlgTBB(matrix_tbb, size, eps);
  double end_tbb = clock();
  double time_tbb =
  static_cast<float>(end_tbb - start_tbb) / CLOCKS_PER_SEC;
  delete[] matrix_tbb;

  double* matrix_std = new double[size * size];
  FillingTheMatrix(matrix_std, size);
  double start_std = clock();
  ParallelAlgSTD(matrix_std, size, eps);
  double end_std = clock();
  double time_std =
  static_cast<float>(end_std - start_std) / CLOCKS_PER_SEC;
  delete[] matrix_std;

  std::cout << "SEQ: " << time_seq << std::endl;
  std::cout << "TBB: " << time_tbb << std::endl;
  std::cout << "STD: " << time_std << std::endl;

  SUCCEED();
}
*/
```