МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»

студента 2 курса, 18209 группы

Большим Максима Антоновича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Матвеев А.С.

Новосибирск 2020

Оглавление

Цель

Написать параллельную программу, которая реализует метод простой итерации для решения СЛАУ вида Ax=b.

Задание

Написать программу на языке C++, которая реализует метод простой итерации для решения СЛАУ вида Ax=b, где A – матрица размером N×N, x и b – векторы длины N. Тип элементов – double. Распараллелить алгоритм решения СЛАУ посредством стандарта параллельного программирования OpenMP. После реализации программы запустить код на вычислительном сервере nusc.nsu.ru на 8 вычислительных ядрах, используя последовательно 1, 2, 4, 8, 16 и 32 потока для вычисления алгоритма в программе. Проанализировать полученные результаты и сделать вывод, при каких настройках алгоритм будет выполняется быстрее.

Алгоритм решения СЛАУ

Для решения СЛАУ использовался метод простой итерации, суть которого сводится к следующей формуле:

$$x^{x+1} = x^n - \tau\left(Ax^n - b\right)$$

Где x^n - вектор решения уравнения на *n*-й итерации. Мы вычисляем итерационно формулу вплоть до тех пор, пока не будет выполняться критерий сходимости решения СЛАУ:

$$\frac{||Ax^n - b||_2}{||b||_2} < \varepsilon$$

Где ε = 10^{-6} .

Листинг программы

```
    #include <cstdlib>
    #include <string>
    #include <sstream>

    namespace patch
    {
        template <typename T>
        std::string to_string(const T& n)
        {
            std::ostringstream stm;
            stm << n;
            return stm.str();
        }
    }

    #include <iostream>
    #include "stdio.h"
    #include "math.h"
    #include "omp.h"

    using namespace std;

    double *getSolution(const double *matrix, const double *right, long size);
    double *generateMatrix(double **right, long size);
    double  *fillMatrix(double  **right,  const  string&  fileName,  const  string&
ansName);

    int main(int argc, char *argv[])
    {
        if (argc == 1)
        {
            cout << "No arg!" << endl;
            return 0;
        }
        if(argc > 5)
        {
            for (int i = 0; i < argc; ++i)
            {
                cout << argv[i] << endl;
            }
            cout << "Too many args!" << endl;
            return 0;
        }

        if(argc != 5 && argc != 3)
        {
            cout << "Invalid arg count!" << endl;
            return 0;
        }

        double *matrix = NULL;
        double *right = NULL;

        try
        {
            if(argc == 5)
                matrix = fillMatrix(&right, (string)argv[1], (string)argv[2]);
            else
                matrix = generateMatrix(&right, atoi(argv[1]));
        } catch (exception &e)
        {
            cout << "lol.. " << e.what() << endl;
            return 0;
        }
```

```
          double start_time = omp_get_wtime();
          if(argc == 5) //TODO FILE Tester Section
          {
                double *sol = getSolution(matrix, right, 2500);//TODO 2500 - size of
matrix
              if(sol)
              {
                  double end_time = omp_get_wtime();
                  FILE* ifsAns = fopen(argv[3], "r");
                  FILE* ans    = fopen(argv[4], "w");
                  float a = 0;

                  for (int i = 0; i < 2500; ++i)
                  {
                      fread(((void*)&a), sizeof(float), 1, ifsAns);
                      fwrite(((void*)&a), sizeof(float), 1, ans);
                  }
                  cout << end_time - start_time << endl << "--------" << endl;
                  fclose(ifsAns);
                  fclose(ans);
                  free(sol);
              } else
                  cout << "Haven't solution!!!" << endl;
          } else
          {
              double *sol = getSolution(matrix, right, atoi(argv[1]));
              FILE* ans    = fopen(argv[4], "w");

              if(sol)
              {
                  double end_time = omp_get_wtime();
                  string a;
                  for (int i = 0; i < atoi(argv[1]); ++i)
                  {
                       a = "x" + patch::to_string(i) + " = " + patch::to_string(sol[i])
+ "\n";
                      fwrite(((void*)a.c_str()), sizeof(char), a.length(), ans);
                  }
                  cout << end_time - start_time << endl << "--------" << endl;
                  free(sol);
                  fclose(ans);
              } else
                  cout << "Haven't solution!!!" << endl;
          }

          free(matrix);
          free(right);
          return 0;
      }

      double *fillMatrix(double **right, const string& fileName, const string& ansName)
      {
          double* matrix = new double[6250000];
          (*right)  = new double[2500];

          FILE* ifsMatrix = fopen(fileName.c_str(), "r");
          FILE* ifsAns= fopen(ansName.c_str(), "r");
          float a = 0;

          for (int i = 0; i < 6250000; ++i)
          {
              fread(((void*)&a), sizeof(float), 1, ifsMatrix);
              matrix[i] = a;
          }

          for (int i = 0; i < 2500; ++i)
          {
              fread(((void*)&a), sizeof(float), 1, ifsAns);
              (*right)[i] = a;
```

```
        }

        fclose(ifsMatrix);
        fclose(ifsAns);

        return matrix;
    }

    double *generateMatrix(double **right, long size)
    {
        double* matrix = new double[size * size];
        (*right) = new double[size];

        for (int i = 0; i < size; i++)
        {
            (*right)[i] = (((double) rand() / (RAND_MAX)) + 1) * 3.0 * size - 1.0;
            for (int j = 0; j < size; j++)
                if (i == j)
                    matrix[i * size + i] = (((double) rand() / (RAND_MAX)) + 1) * 2.0
* size;
                else
                      matrix[i * size + j] = (((double) rand() / (RAND_MAX)) + 1) *
1.0;
        }

        return matrix;
    }

    double *getSolution(const double *matrix, const double *right, long size)
    {
        double*  solution = new double[size];
        double*  solutionBuffer = new double[size];

        double tau = 0.015;
        bool diverge0 = false;
        bool diverge1 = false;
        bool flag = true;
        bool setOld = false;
        int divergeCount = 0;
        double oldValue = 0;

        double norm0 = 0, norm1 = 0;

         #pragma omp parallel shared(norm0, norm1, tau, size, right, divergeCount,
oldValue, matrix, diverge0, diverge1, setOld, flag, solution, solutionBuffer, cout)
default(none)
        {
            #pragma omp for reduction(+:norm0)
                for (int i = 0; i < size; ++i)
                    norm0 += right[i] * right[i];

            #pragma omp single
            {
                norm0 = sqrt(norm0);
            }

            while (flag)
            {
                #pragma omp for reduction(+:norm1)
                    for (int i = 0; i < size; ++i)
                    {
                        double valueX = 0;

                        const double *m = matrix + i * size;
                        for (int j = 0; j < size; ++j) //Ax
                            valueX += m[j] * solution[j];

                        valueX -= right[i]; //Ax - b
                        solutionBuffer[i] = solution[i] - valueX * tau;
```

```
                    norm1 += valueX * valueX;
                }
            #pragma omp single
            {
                swap(solution, solutionBuffer);

                norm1 = sqrt(norm1);
                flag = (norm1 / norm0) > 1e-6;

                if (!setOld)
                    setOld = true;
                else if (oldValue < norm1 / norm0)
                    divergeCount++;

                oldValue = norm1 / norm0;
                norm1 = 0;

                if (divergeCount > 50)
                {
                    if (diverge0)
                    {
                        diverge1 = true;
                        flag = false;
                    } else
                    {
                        diverge0 = true;
                        tau *= -1;
                        divergeCount = 0;
                    }
                }
            }
        }
    }

    free(solutionBuffer);

    if (diverge0 && diverge1)
    {
        free(solution);
        return NULL;
    } else
        return solution;
}
```

Код непараллельной программы будет выглядеть так же, но без директив `#pragma omp …`

Производительность программы

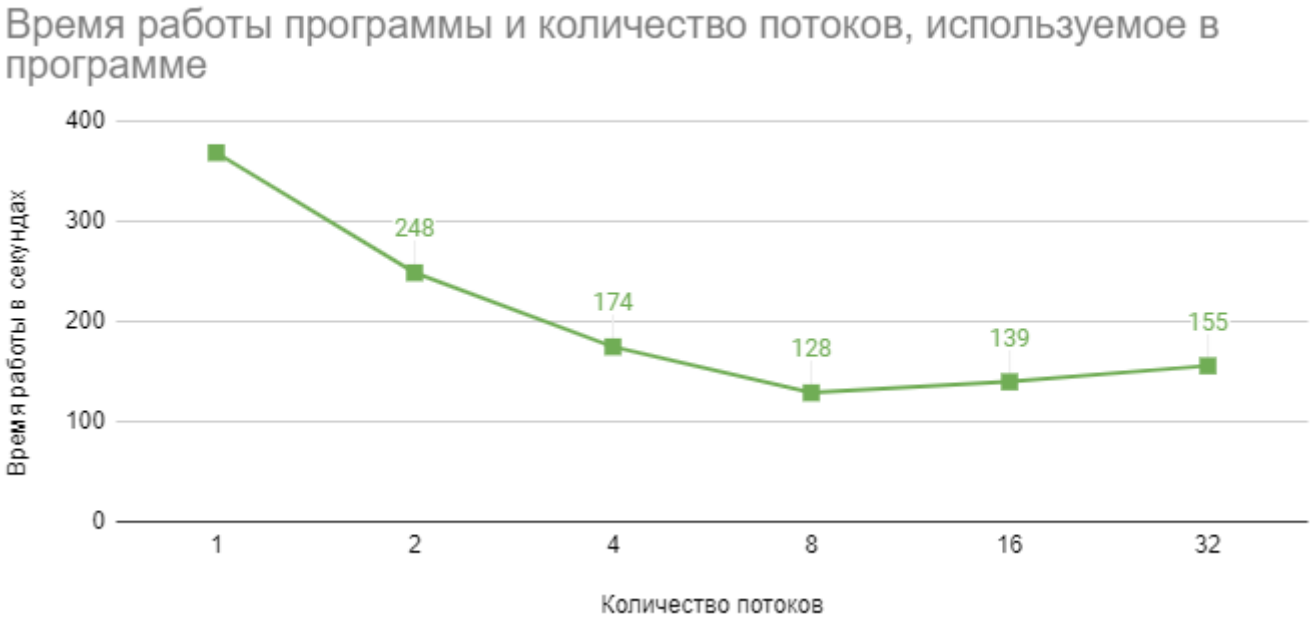

Таблица эффективности

Кол-во потоков	1	2	4	8	16	32
Эффективность	1	0.74	0.52	0.35	0.16	0.07

На графике показана производительность программы при решении СЛАУ, размер матрицы которой был 2500*2500 элементов.

На графике мы можем заметить, что программа с количеством потоков более 8 начинает замедляться при исполнении. Это связано с тем, что количество ядер, выделенных на сервере для вычисления программы, было всего 8. Когда количество потоков было равно 8, программа задействовала вся свободные для него ядра. После, когда мы начали исполнять 16 потоков, каждое отдельное ядро стало исполнять по 2 потока, что начало снижать производительность, т.к. увеличение количества потоков одного процесса на ядро никак не даёт выигрыш в производительности, а даёт накладной расход в обработке и исполнении самих потоков. Поэтому целесообразно использовать количество потоков, равное количеству свободных для вычислений ядер.

Стоить отметить, что время работы кода без директив будет целиком зависеть от процессора, который его исполняет, а также от уровня оптимизации при компиляции кода. В случае компиляции программы с ключом –O3 компилятором GCC код исполнялся около 200 секунд. Без

оптимизаций время было почти таким же, как время работы программы на одном ядре – 368 секунд.

Заключение

В ходе лабораторной программы была освоена методика разбития программу на языке C++ на потоки, эффективно используя ресурсы процессора. Проанализировав результаты работы, можно сделать выбор, что огромное влияние на время исполнения программы влияют следующие факторы:

1) Объём обрабатываемых данных, в нашем случае – размер матрицы;
2) Количество ядер на машине, где выполнялся код программы.

На основе графика я пришёл к выводу, что на сервере, где исполнялся код, приемлемое количество потоков для вычисления алгоритма равно самому количеству ядер, выделенных под процесс самой программы. При дальнейшем возрастании количества потоков программа только замедлялась, так как одно ядро вычисляло уже 2 потока вместо одного, что накладывало расходы в производительности. Похожие показатели были и на 8-ядерном ноутбуке автора отчёта, что подтверждает выдвинутую гипотезу. Однопоточное программирование приложения для вычисления решения СЛАУ в данном случае крайне нецелесообразно, поскольку велась постоянная работа с матрицами и требовалась работа нескольких ядер.