Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа №3

по курсу «Разработка параллельных и распределенных программ»

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

Студент группы ИУ9-51Б Киселев К.А.

Преподаватель Царев А.С.

Москва 2023

Содержание

1 Постановка задачи

1. Написать программу, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A – матрица размером N×N, x и b – векторы длины N. Тип элементов – double.

2. Программу распараллелить с помощью MP с разрезанием матрицы A по строкам на близкие по размеру, возможно не одинаковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. Реализовать два варианта программы: 1: векторы x и b дублируются в каждом потоке, 2: векторы x и b разрезаются между потоками аналогично матрице A. (только для сдающих после срока) Уделить внимание тому, чтобы при запуске программы на различном числе потоков решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

3. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1,2, 4, 8, 16. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные, параметры N и ε подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Также параметр N разрешено подобрать таким образом, чтобы он нацело делился на на 1,2,4,8 и 16.

4. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы.

2 Практическая реализация

Код программы для решение СЛАУ методом простых итераций с делением матрицы A между потоками

```
#include "solver.hpp"
#include "boost/numeric/ublas/matrix.hpp"
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/matrix_proxy.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <cmath>
#include <omp.h>

using namespace boost::numeric;

const double EPSILON = 1e-4;

Solver::Solver(size_t size, ublas::matrix<double> &coeffs,
               ublas::vector<double> &free_coeffs)
    : size(size) {
  this->coeffs = ublas::matrix<double>(coeffs);
  this->free_coeffs = ublas::vector<double>(free_coeffs);
}

ublas::vector<double> Solver::FindSolution() {

  ublas::vector<double> result(size);

  double tau = double(1) / this->size;
  double metric = std::numeric_limits<double>::max();
  double prev = metric;

  while (metric >= EPSILON) {
    prev = metric;

#pragma omp parallel shared(result)
    {
      size_t thread_num = omp_get_thread_num();
      size_t chunk_size = this->size / omp_get_max_threads();
      size_t start_index = thread_num * chunk_size;
      size_t end_index = (thread_num == omp_get_max_threads() - 1)
                             ? this->size
                             : start_index + chunk_size;

      ublas::vector<double> chunk = this->iterate_solution(result, tau);

#pragma omp barrier
      for (size_t j = start_index; j < end_index; ++j) {
```

```
45          result(j) -= chunk(j - start_index);
46        }
47
48  #pragma omp barrier
49      }
50
51      metric = this->calc_metric(result);
52
53      if (prev < metric) {
54        tau *= -1;
55      }
56    }
57
58    return result;
59  }
60
61  ublas::vector<double> Solver::iterate_solution(ublas::vector<double> &sol,
62                                                 double tau) {
63    ublas::vector<double> tmp_free = ublas::subslice(
64        this->free_coeffs,
65        omp_get_thread_num() * (this->size / omp_get_max_threads()), 1,
66        this->size / omp_get_max_threads());
67
68    ublas::matrix<double> tmp_coeffs = ublas::subslice(
69        this->coeffs, omp_get_thread_num() * (this->size / omp_get_max_threads()),
70        1, this->size / omp_get_max_threads(), 0, 1, this->size);
71    ublas::vector<double> tmp = ublas::prod(tmp_coeffs, sol);
72    tmp -= tmp_free;
73    tmp *= tau;
74    return tmp;
75  }
76
77  double Solver::calc_metric(ublas::vector<double> &sol) {
78
79    double numerator = 0;
80  #pragma omp parallel shared(numerator)
81    {
82      ublas::vector<double> tmp_free = ublas::subslice(
83          this->free_coeffs,
84          omp_get_thread_num() * (this->size / omp_get_max_threads()), 1,
85          this->size / omp_get_max_threads());
86
87      ublas::matrix<double> tmp_coeffs = ublas::subslice(
88          this->coeffs,
89          omp_get_thread_num() * (this->size / omp_get_max_threads()), 1,
90          this->size / omp_get_max_threads(), 0, 1, this->size);
91      ublas::vector<double> tmp = ublas::prod(tmp_coeffs, sol);
92      tmp -= tmp_free;
93
94      double sum = 0;
```

```
95      std::for_each(tmp.begin(), tmp.end(), [&sum](double d) { sum += d * d; });
96  #pragma omp atomic
97      numerator += sum;
98    }
99
100   return std::sqrt(numerator) / ublas::norm_2(this->free_coeffs);
101 }
```

3 Результаты сравнения производительности

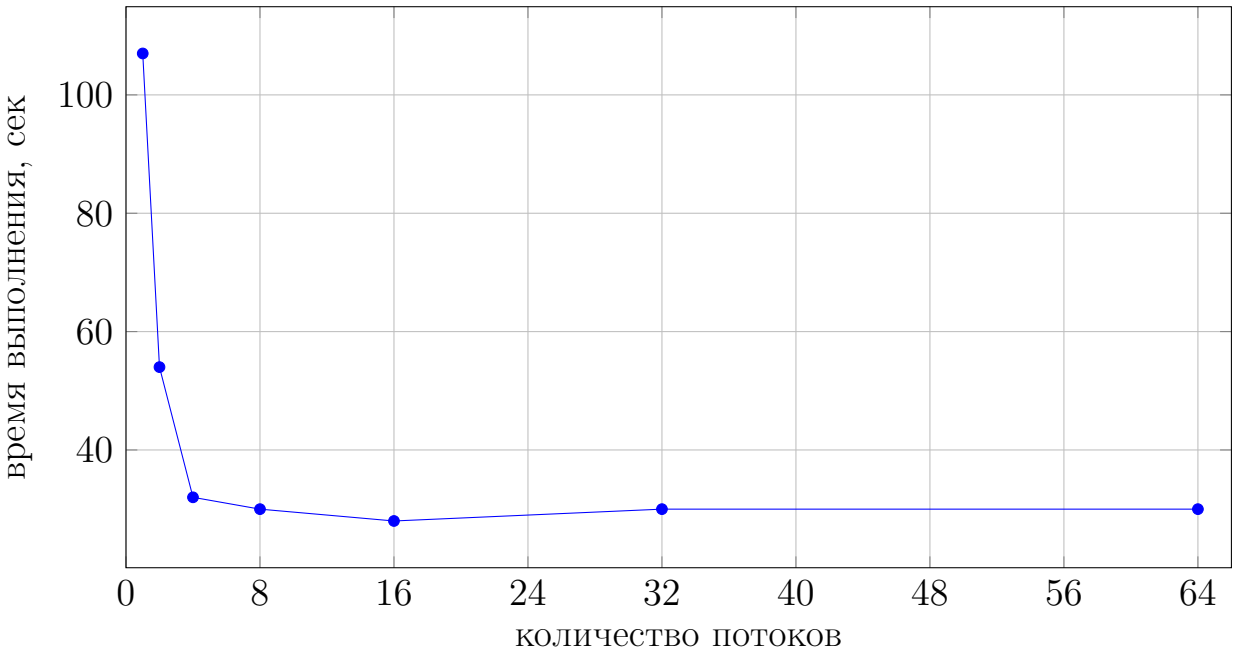

Рис. 1: зависимость времени выполнения от количества потоков