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 ПРАКТИЧЕСКОЙ РАБОТЫ

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

студента 2 курса, группы 21209

Панас Матвей Алексеевич

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Мичуров М.А.

Новосибирск 2023

ЦЕЛЬ

Ознакомиться со средством параллельного программирования OpenMP, освоить основные принципы многопоточного программирования.

ЗАДАНИЕ

Последовательную программу из лабораторной работы 1, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP. Реализовать два варианта программы:

- Вариант 1: для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for,

- Вариант 2: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.

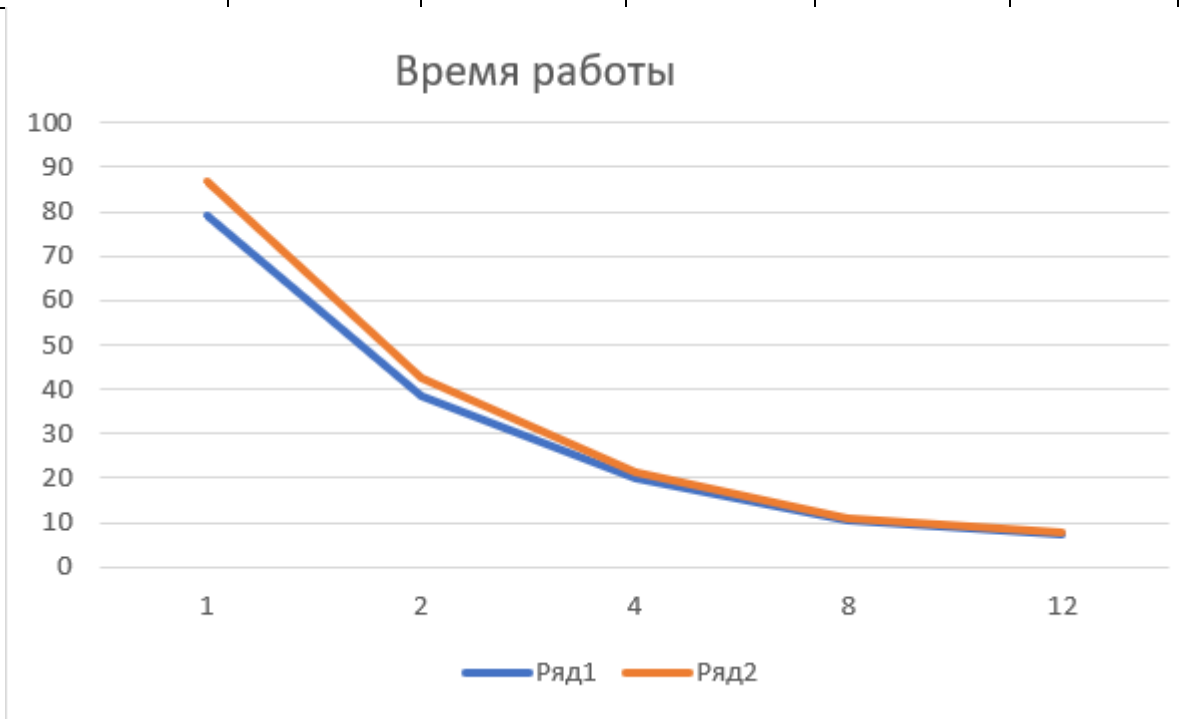
2. Замерить время работы программ при использовании различного числа процессорных ядер. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.

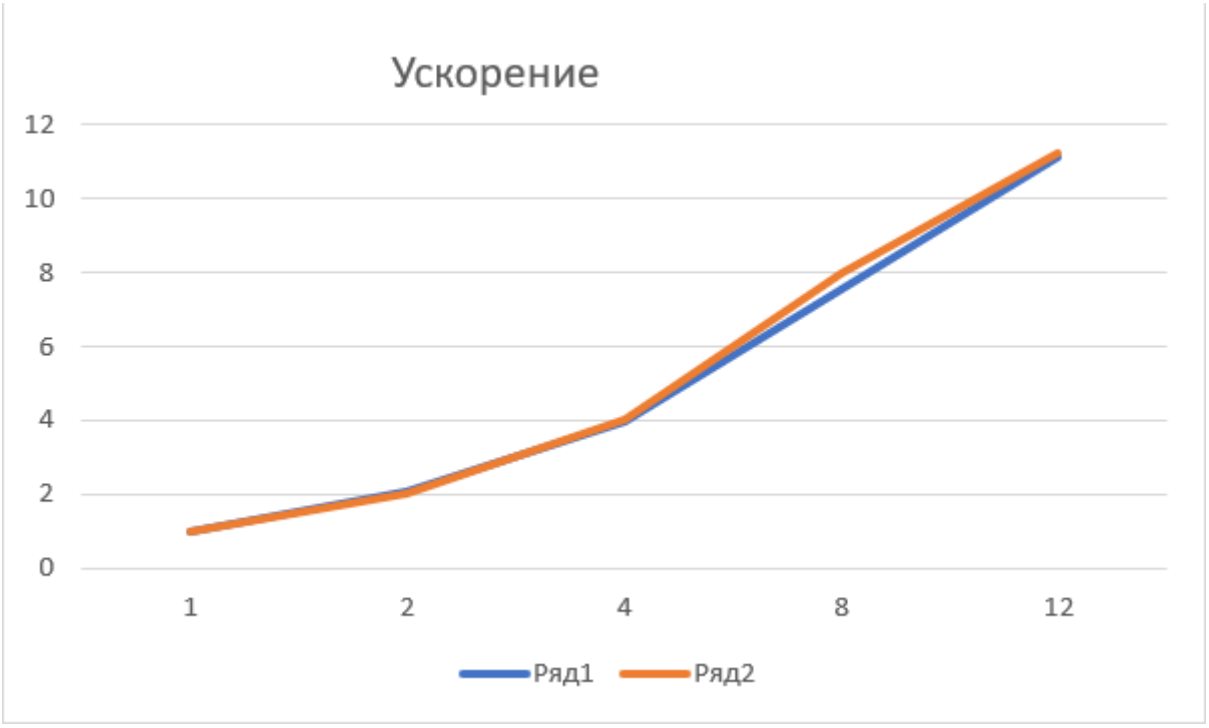
3. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.

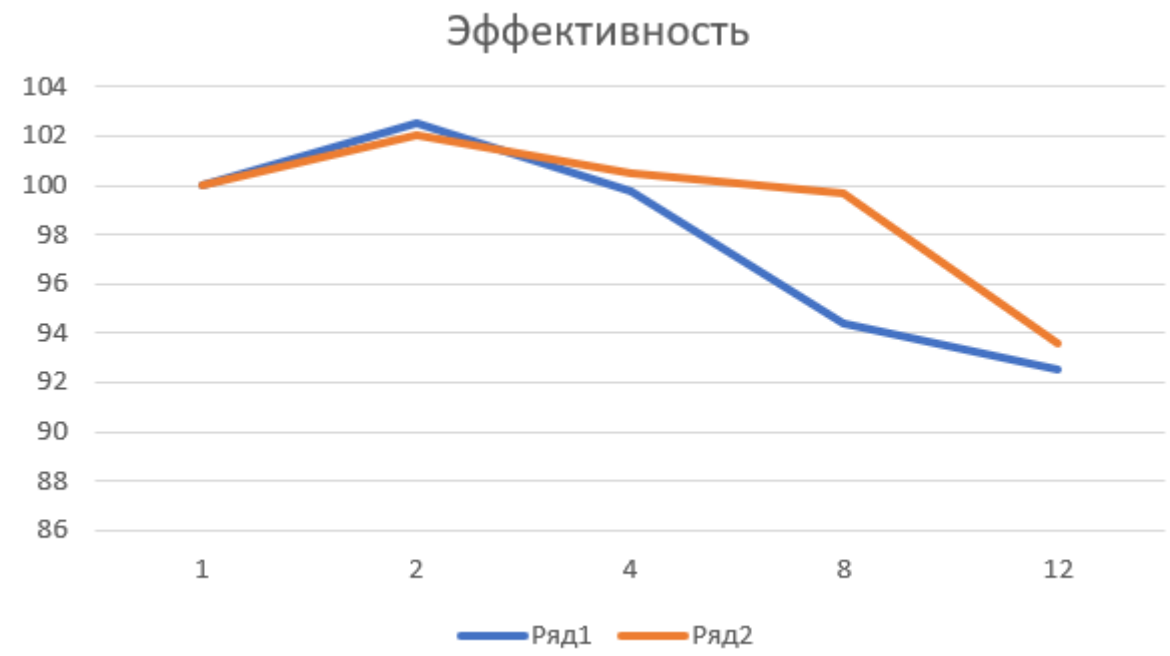
ОПИСАНИЕ РАБОТЫ

Были написана программа, реализующая метод простых итераций для решения системы линейных алгебраических уравнений. Затем она была распараллелена с использованием средства многопоточного программирования OpenMP. Размер матрицы 8500*8500.
Замерено время работы параллельных программ при использовании 2, 4, 8 и 12 потоков.
Вычислены ускорение и эффективность параллельных программ. Получены следующие результаты:

Кол-во потоков	1	2	4	8	12
Время 1, с	79.3	38.657	19.831	10.497	7.137
Ускорение 1, раз	1	2.05	3.99	7.55	11.1
Эффективность 1, %	100	102.5	99.75	94.37	92.5
Время 2, с	86.634	42.373	21.516	10.854	7.706
Ускорение 2, раз	1	2.04	4.02	7.98	11.24
Эффективность 2, %	100	102	100.5	99.7	93.6




Ускорение
12
10
8
6
4
2
0
1
2
4
8
12
Ряд1
Ряд2


Эффективность
104
102
100
98
96
94
92
90
88
86
1
2
4
8
12
Ряд1
Ряд2

Как видно из графиков, эффективность программы при увеличении числа потоков уменьшается. Это связано с увеличением времени, необходимого на распределение работы между потоками перед исполнением каждого цикла.

Проведено исследование на определение оптимальных параметров директивы #pragma omp for schedule(...) при том же размере задачи и количестве потоков, равном 8.
Получены следующие результаты:

Параметры	Время работы
static, 10	11.82
static, 100	11.445
dynamic, 1	35.295
dynamic, 20	11.803
guided, 1	11.531
guided, 20	11.501
auto	11.425

Статическое распределение на равные части (размер матрицы – $8500 \approx 1062 * 8$) даёт оптимальное время работы, примерно совпадающее с вариантом программы с директивой #pragma omp for schedule(auto), из чего можно сделать предположение, что распределение работы по умолчанию происходит примерно таким же образом.

При динамическом распределении для каждой порции итераций необходимо определять поток, который их исполнит, что, очевидно, долго, особенно при малом размере этих порций.

Управляемое распределение разделяет работу между потоками на большие по размеру части, чем динамическое, благодаря чему быстрее его, но вынуждено вычислять этот размер, отчего медленнее оптимального статического.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы обнаружено, что запуск распараллеленной программы на небольшом (2-4) числе потоков даёт пропорциональное этому числу ускорение работы программы. На большем числе ядер эффективность незначительно снижается, в отличии от MPI программ. Задание параметров для распределения работы между потоками на такой задаче, где все итерации в каждом цикле одинаковы по вычислительной сложности, не оптимизировало время работы программы, распределение по умолчанию оказалось наиболее эффективным.

Параллельная программа вариант 1

```c
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 10000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 100000

void generate_A(double* A, int size) {
#pragma omp parallel for
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; ++j) {
      A[i * size + j] = 1;
    }

    A[i * size + i] = 2;
  }
}

void generate_x(double* x, int size) {
#pragma omp parallel for
  for (int i = 0; i < size; i++) {
    x[i] = 0;
  }
}

void generate_b(double* b, int size) {
#pragma omp parallel for
  for (int i = 0; i < size; i++) {
    b[i] = N + 1;
  }
}

double calc_norm_square(const double* vector, int size) {
  double norm_square = 0.0;
#pragma omp parallel for reduction(+ : norm_square)
  for (int i = 0; i < size; ++i) {
    norm_square += vector[i] * vector[i];
  }

  return norm_square;
}

void calc_Axb(const double* A, const double* x, const double* b, double* Axb, int size) {
#pragma omp parallel for
  for (int i = 0; i < size; ++i) {
    Axb[i] = -b[i];
    for (int j = 0; j < N; ++j) {
      Axb[i] += A[i * N + j] * x[j];
    }
  }
}

void calc_next_x(const double* Axb, double* x, double tau, int size) {
  for (int i = 0; i < size; ++i) {
    x[i] -= tau * Axb[i];
  }
}

int check_print_error(double* a, const char* message) {
  if (!a) {
    fprintf(stream: stderr, format: "%s", message);
    return 1;
  }
  return 0;
```

```
69
70  double* allocate_matrix(size_t n) {
71    double* matrix = malloc(size: n * n * sizeof(double));
72    check_print_error(matrix, message: "Failed to allocate memory to matrix\n");
73    return matrix;
74  }
75
76  double* allocate_vector(size_t n) {
77    double* vector = malloc(size: n * sizeof(double));
78    check_print_error(a: vector, message: "Failed to allocate memory to vector\n");
79    return vector;
80  }
81
82  void print_vector(const double* vector, size_t n) {
83    for (size_t i = 0; i < n; ++i) {
84      printf(format: "%lf ", vector[i]);
85    }
86    printf(format: "\n");
87  }
88
89  int main() {
90    double* x;
91    double* A;
92    double* Axb;
93    double* b;
94    size_t iters_count;
95    double start_time = 0.0, finish_time = 0.0;
96    double b_norm = 0.0;
97    double accuracy = EPSILON + 1;
98
99    x = allocate_vector(n: N);
100   b = allocate_vector(n: N);
101   A = allocate_matrix(n: N);
102   generate_x(x, size: N);
103   generate_b(b, size: N);
104   generate_A(A, size: N);
105
106   b_norm = sqrt(x: calc_norm_square(vector: b, size: N));
107   Axb = allocate_vector(n: N);
108
109   start_time = omp_get_wtime();
110
111   for (iters_count = 0; iters_count < MAX_ITERATION_COUNT && accuracy > EPSILON; ++iters_count) {
112     calc_Axb(A, x, b, Axb, size: N);
113     calc_next_x(Axb, x, tau: TAU, size: N);
114     accuracy = sqrt(x: calc_norm_square(vector: Axb, size: N)) / b_norm;
115   }
116
117   finish_time = omp_get_wtime();
118
119   if (iters_count == MAX_ITERATION_COUNT) {
120     printf(format: "Too many iterations\n");
121   } else {
122     printf(format: "Time: %lf sec\n", finish_time - start_time);
123   }
124
125   free(ptr: x);
126   free(ptr: b);
127   free(ptr: A);
128   free(ptr: Axb);
129
130   return 0;
131 }
```

Параллельная программа вариант 2

```c
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 5000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 100000
double res = 0.0;

void generate_A(double* A, int size) {
#pragma omp for
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; ++j) {
      A[i * size + j] = 1;
    }

    A[i * size + i] = 2;
  }
}

void generate_x(double* x, int size) {
#pragma omp for
  for (int i = 0; i < size; i++) {
    x[i] = 0;
  }
}

void generate_b(double* b, int size) {
#pragma omp for
  for (int i = 0; i < size; i++) {
    b[i] = N + 1;
  }
}

double calc_norm_square(const double* vector, int size) {
#pragma omp single
  res = 0.0;
  double local_norm = 0.0;
#pragma omp for
  for (int i = 0; i < size; ++i) {
    local_norm += vector[i] * vector[i];
  }
#pragma omp atomic update
  res += local_norm;
#pragma omp barrier
  return res;
}

void calc_Axb(const double* A, const double* x, const double* b, double* Axb, int size) {
#pragma omp for
  for (int i = 0; i < size; ++i) {
    Axb[i] = -b[i];
    for (int j = 0; j < N; ++j) Axb[i] += A[i * N + j] * x[j];
  }
}

void calc_next_x(const double* Axb, double* x, double tau, int size) {
  for (int i = 0; i < size; ++i) {
    x[i] -= tau * Axb[i];
  }
}

int check_print_error(double* a, const char* message) {
  if (!a) {
    fprintf(stream: stderr, format: "%s", message);
```

```
    return 1;
  }
  return 0;
}

double* allocate_matrix(size_t n) {
  double* matrix = malloc(size: n * n * sizeof(double));
  check_print_error(matrix, message: "Failed to allocate memory to matrix\n");
  return matrix;
}

double* allocate_vector(size_t n) {
  double* vector = malloc(size: n * sizeof(double));
  check_print_error(a: vector, message: "Failed to allocate memory to vector\n");
  return vector;
}

void print_vector(const double* vector, size_t n) {
  for (size_t i = 0; i < n; ++i) {
    printf(format: "%lf ", vector[i]);
  }
  printf(format: "\n");
}

int main(void) {
  double* x;
  double* A;
  double* Axb;
  double* b;
  size_t iters_count;
  double start_time = 0.0, finish_time = 0.0;

  double b_norm = 0.0;
  double accuracy = EPSILON + 1;

  x = allocate_vector(n: N);
  b = allocate_vector(n: N);
  A = allocate_matrix(n: N);
#pragma omp parallel shared(res)
  generate_x(x, size: N);
  generate_b(b, size: N);
  generate_A(A, size: N);
  b_norm = calc_norm_square(vector: b, size: N);

  b_norm = sqrt(x: b_norm);

  Axb = allocate_vector(n: N);

  start_time = omp_get_wtime();

  for (iters_count = 0; iters_count < MAX_ITERATION_COUNT && accuracy > EPSILON; ++iters_count) {
    calc_Axb(A, x, b, Axb, size: N);
    calc_next_x(Axb, x, tau: TAU, size: N);
    accuracy = calc_norm_square(vector: Axb, size: N);

    accuracy = sqrt(x: accuracy);
    accuracy /= b_norm;
  }

  finish_time = omp_get_wtime();

  if (iters_count == MAX_ITERATION_COUNT) {
    printf(format: "Too many iterations\n");
```

```
    if (iters_count == MAX_ITERATION_COUNT) {
        printf(format: "Too many iterations\n");
    } else {
        printf(format: "Time: %lf sec\n", finish_time - start_time);
        print_vector(vector: x, n: N);
    }

    free(ptr: x);
    free(ptr: b);
    free(ptr: A);
    free(ptr: Axb);

    return 0;
}
```