

Министерство образования и науки Российской Федерации

Новосибирский национальный исследовательский государственный  
университет

# Основы параллельного программирования

## Отчет по лабораторной работе № 1

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

Студент: Зверев Андрей, гр. 23206

Преподаватель: Мичуров Михаил Антонович

Новосибирск, 2025 г.

# 1. Цель работы

1. Последовательную программу, 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.

Уделить внимание тому, чтобы при запуске программы на различном числе OpenMP-потоков решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

2. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: от 1 до числа доступных в узле.

Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.

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

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

## 2. Описание работы

### 1) Реализация программы решения СЛАУ

Для решения СЛАУ я выбрал метод простой итерации, в котором тау я выбирал динамически, в зависимости от  $N$ . Для проверки работоспособности и правильности работы я выполнил первый из предложенных вариантов - модельную задачу с заданным решением. Проверил правильность работы на маленьких данных (размерах  $N$ ), выводя полностью вектор решений. Затем уменьшил вывод до 4-х значений из вектора решений  $X$  по индексам 0,  $N/3$ ,  $N/2$ ,  $N-1$ . Листинг этой программы предоставлен Приложении 1.

### 2) Распараллеливание программы двумя методами

Для распараллеливания я использовал программу, реализованную в п.1 и OpenMP. В первом варианте параллельной программы я распараллелил каждый цикл с помощью `#pragma omp parallel for`, заранее установив количество потоков в начале программы (вводится пользователем). Во втором варианте программы я распараллелил только главный цикл программы в функции `isgoodSolve`, также использовал `#pragma omp single` (что указывает на выполнение блока кода только одним процессом) на перерасчет данных (поиск  $x(n+1)$ ). Также пришлось изменить функцию `isGoodSolve` так, чтобы распараллеливание было эффективно (в моем варианте программы одно ядро выполняло все задачи, что приводило к простаиванию остальных и увеличению времени работы). Листинги программ представлены в Приложениях 2, 3 соответственно.

### 3) Использование сервера и замер времени, сбор данных

Для проведения измерений я использовал кафедральный сервер и его вычислительные мощности. Для компиляции файлов я написал bash-скрипт, компилирующий с помощью g++ сразу 3 файла. (Приложение 4). Для построения графиков использовал программу на python. (Приложение 5)

Начальные данные  $N=4000$ ,  $T=1$ . Время выполнения для них составило 32 секунды. Далее количество потоков увеличивалось по

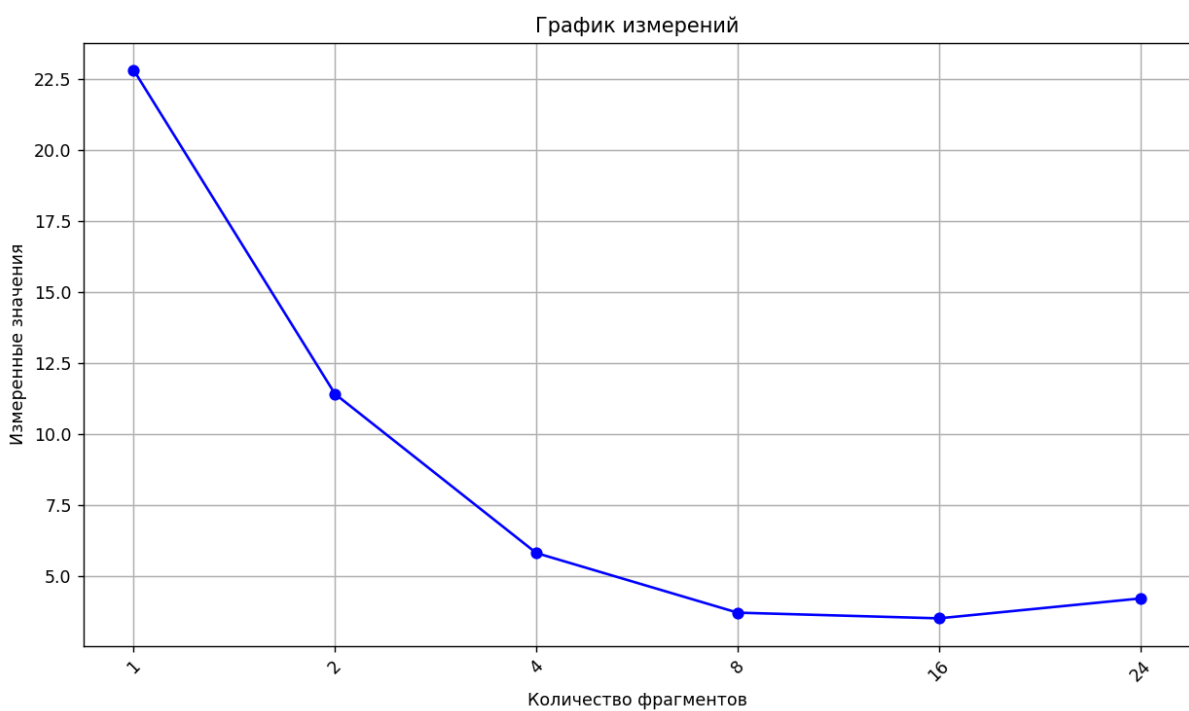
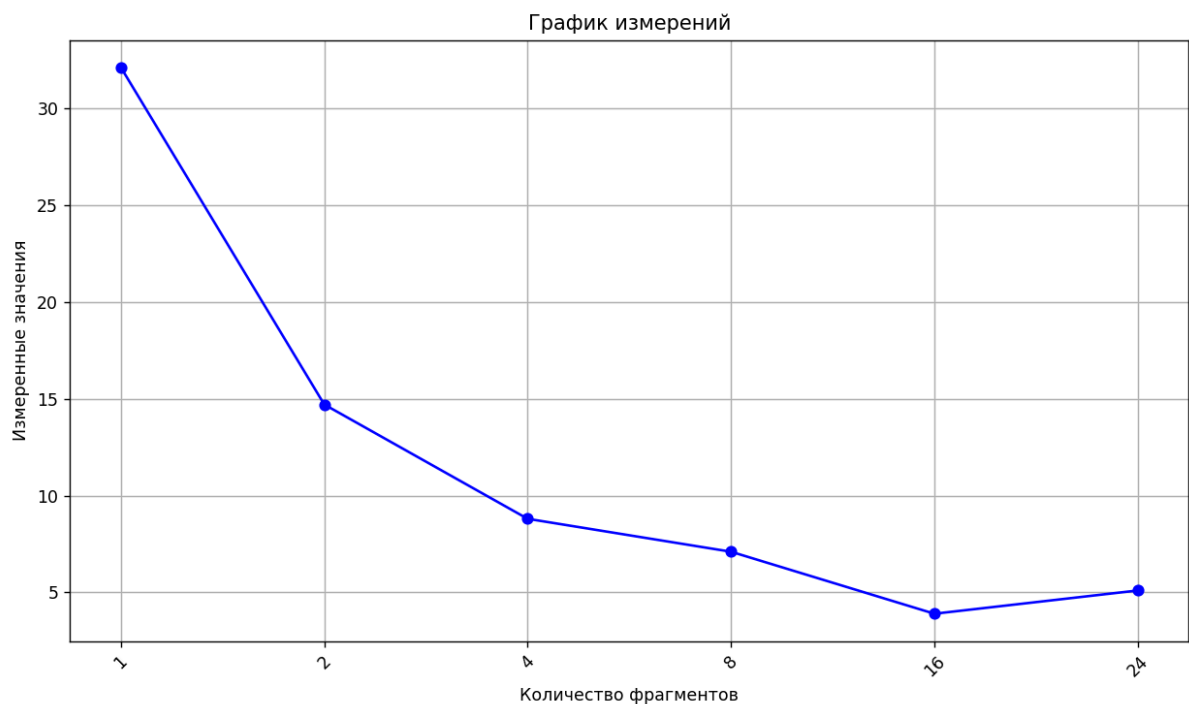
степеням двойки (1,2,4,8,16,24). 24 - максимально доступное количество.

Ниже представлены графики и данные замеров.

Для первого варианта программы: 32.1 14.7 8.8 7.1 3.9 5.1

Для второго варианта программы: 22.8 11.4 5.8 3.7 3.5 4.2

## Графики:



#### *4) Поиск наилучших параметров schedule*

Для поиска наилучших параметров была немного переписана распараллеленная программа первого варианта, чтобы при компиляции было удобно подставлять переменные окружения. (Приложение 6) За исходные данные выбраны  $N=3000$   $T=4$ . SHUNK\_SIZE были выбраны: 10, 50, 100. Полученные результаты:

*static: 5.02 4.65 5.17*

*dynamic: 5.94 7.26 6.04*

*guided: 6.4 5.99 4.99*

*auto: 5.1 6.9 5.39*

*runtime: 6.06 4.67 4.68*

Из них можно сделать вывод, что параметры static, 50 наилучшие для моей программы.

### 3. Приложения

Все программы также находятся на GitHub:

[https://github.com/whoitandrei/OPP/tree/master/lab\\_1](https://github.com/whoitandrei/OPP/tree/master/lab_1)

#### Приложение 1. Листинг программы решения СЛАУ

```
#include <iostream>
#include <vector>
#include <cmath>
#include <ctime>

#define EPS 0.00001
#define TAU (1.9 / (size + 1))

/* math with vectors funcs */
std::vector<double> mulMatrixVector(const
std::vector<std::vector<double>>& matrix, const std::vector<double>&
vec, int size) {
    std::vector<double> result(size, 0.0);

    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            result[i] += matrix[i][j] * vec[j];
        }
    }
    return result;
}

std::vector<double> subVector(const std::vector<double>& vec1, const
std::vector<double>& vec2, int size) {
    std::vector<double> result(size);

    for (int i = 0; i < size; ++i) {
        result[i] = vec1[i] - vec2[i];
    }
    return result;
}

void mulConstVector(std::vector<double>& vec, double num, int size) {
    for (int i = 0; i < size; ++i) {
        vec[i] *= num;
    }
}
```

```

}

/* funcs with matrices (init etc.) */
void initMatrix(std::vector<std::vector<double>>>* matrix, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            if (i == j) (*matrix)[i][j] = 2;
            else (*matrix)[i][j] = 1;
        }
    }
}

void initVectorB(std::vector<double>* B, int size) {
    for (int i = 0; i < size; ++i) {
        (*B)[i] = size + 1;
    }
}

void initVectorX(std::vector<double>* X, int size) {
    for (int i = 0; i < size; ++i) {
        (*X)[i] = 0.;
    }
}

/* funcs for solve approximating to EPS */
double normOfVector(const std::vector<double>& vec, int size) {
    double result = 0;
    for (int i = 0; i < size; ++i) {
        result += (vec[i] * vec[i]);
    }
    return sqrt(result);
}

bool isGoodSolve(const std::vector<double>& AX_B, const
std::vector<double>& B, int size) {
    return (normOfVector(AX_B, size) / normOfVector(B, size)) < EPS ?
true : false;
}

// (main loop of program)
void findGoodSolve(const std::vector<std::vector<double>>& A,
std::vector<double>* X, const std::vector<double>& B, int size) {

```

```

        std::vector<double> AX_B = subVector(mulMatrixVector(A, *X, size),
B, size);

        while (!isGoodSolve(AX_B, B, size)) {
            mulConstVector(AX_B, TAU, size);
            *X = subVector(*X, AX_B, size);
            AX_B = subVector(mulMatrixVector(A, *X, size), B, size);
        }
    }

/* main func */

/* main func */
int main() {
    int size;
    std::cout << "input N (size of matrix): " << std::endl;
    std::cin >> size;

    std::vector<std::vector<double>> A(size,
std::vector<double>(size));
    std::vector<double> X(size);
    std::vector<double> B(size);

    initMatrix(&A, size);
    initVectorB(&B, size);
    initVectorX(&X, size);

    findGoodSolve(A, &X, B, size);

    std::cout << "answer (vector X):" << std::endl;
    std::cout << X[0] << " " << X[size/3] << " " << X[size/2] << " " <<
X[size-1] << "\n";

    return 0;
}

```



## Приложение 2. Листинг распараллеленной программы (вариант 1)

```
#include <iostream>
#include <vector>
#include <cmath>
#include <ctime>
#include <omp.h>

#define EPS 0.00001
#define TAU (1.9 / (size + 1))

/* math with vectors funcs */
std::vector<double> mulMatrixVector(const
std::vector<std::vector<double>>& matrix, const std::vector<double>&
vec, int size) {
    std::vector<double> result(size, 0.0);

    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            result[i] += matrix[i][j] * vec[j];
        }
    }
    return result;
}

std::vector<double> subVector(const std::vector<double>& vec1, const
std::vector<double>& vec2, int size) {
    std::vector<double> result(size);

    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        result[i] = vec1[i] - vec2[i];
    }
    return result;
}

void mulConstVector(std::vector<double>& vec, double num, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        vec[i] *= num;
    }
}
```

```

}

/* funcs with matrices (init etc.) */
void initMatrix(std::vector<std::vector<double>>>* matrix, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            if (i == j) (*matrix)[i][j] = 2;
            else (*matrix)[i][j] = 1;
        }
    }
}

void initVectorB(std::vector<double>* B, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        (*B)[i] = size + 1;
    }
}

void initVectorX(std::vector<double>* X, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        (*X)[i] = 0.;
    }
}

/* funcs for solve approximating to EPS */
double normOfVector(const std::vector<double>& vec, int size) {
    double result = 0;
    #pragma omp parallel for reduction(+:result)
    for (int i = 0; i < size; ++i) {
        result += (vec[i] * vec[i]);
    }
    return sqrt(result);
}

bool isGoodSolve(const std::vector<double>& AX_B, const
std::vector<double>& B, int size) {
    return (normOfVector(AX_B, size) / normOfVector(B, size)) < EPS;
}

// (main loop of program)

```

```

void findGoodSolve(const std::vector<std::vector<double>>& A,
std::vector<double>* X, const std::vector<double>& B, int size) {
    std::vector<double> AX_B = subVector(mulMatrixVector(A, *X, size),
B, size);

    while (!isGoodSolve(AX_B, B, size)) {
        mulConstVector(AX_B, TAU, size);
        *X = subVector(*X, AX_B, size);
        AX_B = subVector(mulMatrixVector(A, *X, size), B, size);
    }
}

/* main func */
int main() {
    int size;
    std::cout << "input N (size of matrix): ";
    std::cin >> size;

    int T;
    std::cout << "input T (num of threads): ";
    std::cin >> T;

    omp_set_num_threads(T);

    std::vector<std::vector<double>> A(size,
std::vector<double>(size));
    std::vector<double> X(size);
    std::vector<double> B(size);

    initMatrix(&A, size);
    initVectorB(&B, size);
    initVectorX(&X, size);

    double start = omp_get_wtime();
    findGoodSolve(A, &X, B, size);
    double end = omp_get_wtime();

    std::cout << "\nTime: " << end - start << " seconds" << std::endl
<< std::endl;
    std::cout << "Answer (some nums from vector X):" << std::endl;
    std::cout << X[0] << " " << X[size/3] << " " << X[size/2] << " " <<
X[size-1] << std::endl;

```

```
    return 0;  
}
```

### Приложение 3. Листинг распараллеленной программы (вариант 2)

```
#include <iostream>
#include <vector>
#include <cmath>
#include <ctime>
#include <omp.h>

#define EPS 0.00001
#define TAU (1.9 / (size + 1))

/* math with vectors funcs */
std::vector<double> mulMatrixVector(const
std::vector<std::vector<double>>& matrix, const std::vector<double>&
vec, int size) {
    std::vector<double> result(size, 0.0);

    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            result[i] += matrix[i][j] * vec[j];
        }
    }
    return result;
}

std::vector<double> subVector(const std::vector<double>& vec1, const
std::vector<double>& vec2, int size) {
    std::vector<double> result(size);

    for (int i = 0; i < size; ++i) {
        result[i] = vec1[i] - vec2[i];
    }
    return result;
}

void mulConstVector(std::vector<double>& vec, double num, int size) {
    for (int i = 0; i < size; ++i) {
        vec[i] *= num;
    }
}
```

```

/* funcs with matrices (init etc.) */
void initMatrix(std::vector<std::vector<double>>* matrix, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            if (i == j) (*matrix)[i][j] = 2;
            else (*matrix)[i][j] = 1;
        }
    }
}

void initVectorB(std::vector<double>* B, int size) {
    for (int i = 0; i < size; ++i) {
        (*B)[i] = size + 1;
    }
}

void initVectorX(std::vector<double>* X, int size) {
    for (int i = 0; i < size; ++i) {
        (*X)[i] = 0.;
    }
}

/* funcs for solve approximating to EPS */
double normOfVector(const std::vector<double>& vec, int size) {
    double result = 0;
    for (int i = 0; i < size; ++i) {
        result += (vec[i] * vec[i]);
    }
    return sqrt(result);
}

bool isGoodSolve(const std::vector<double>& AX_B, const
std::vector<double>& B, int size) {
    return (normOfVector(AX_B, size) / normOfVector(B, size)) < EPS ?
true : false;
}

// (main loop of program)
void findGoodSolve(const std::vector<std::vector<double>>& A,
std::vector<double>& X, const std::vector<double>& B, int size) {
    std::vector<double> AX_B(size);
    double norm_AB , norm_B;

```

```

#pragma omp parallel
{
    while (true) {
        #pragma omp for
        for (int i = 0; i < size; ++i) {
            double sum = 0.0;
            for (int j = 0; j < size; ++j) {
                sum += A[i][j] * X[j];
            }
            AX_B[i] = sum - B[i];
        }

        #pragma omp single
        {
            norm_AB = 0.0;
            norm_B = 0.0;
        }

        #pragma omp for reduction(+:norm_AB)
        for (int i = 0; i < size; ++i) {
            norm_AB += AX_B[i] * AX_B[i];
        }

        #pragma omp for reduction(+:norm_B)
        for (int i = 0; i < size; ++i) {
            norm_B += B[i] * B[i];
        }

        #pragma omp single
        {
            norm_AB = sqrt(norm_AB);
            norm_B = sqrt(norm_B);
        }

        if (norm_AB / norm_B < EPS) break;

        #pragma omp for
        for (int i = 0; i < size; ++i) {
            X[i] -= TAU * AX_B[i];
        }
    }
}

```

```

}

/* main func */
int main() {
    int size;
    std::cout << "input N (size of matrix): " << std::endl;
    std::cin >> size;

    int T;
    std::cout << "input T (num of threads): ";
    std::cin >> T;

    omp_set_num_threads(T);

    std::vector<std::vector<double>> A(size,
std::vector<double>(size));
    std::vector<double> X(size);
    std::vector<double> B(size);

    initMatrix(&A, size);
    initVectorB(&B, size);
    initVectorX(&X, size);

    double start = omp_get_wtime();
    findGoodSolve(A, X, B, size);
    double end = omp_get_wtime();

    std::cout << "\nTime: " << end - start << " seconds" << std::endl
<< std::endl;
    std::cout << "Answer ( some nums from vector X):" << std::endl;
    std::cout << X[0] << " " << X[size/3] << " " << X[size/2] << " " <<
X[size-1] << std::endl;

    return 0;
}

```



#### Приложение 4. Листинг компилирующего bash-скрипта

```
#!/bin/bash
```

```
SRC_DIR="src"
```

```
BIN_DIR="bin"
```

```
mkdir -p $BIN_DIR
```

```
g++ -Wall -Wextra -o $BIN_DIR/SLAU $SRC_DIR/SLAU.cpp
```

```
g++ -Wall -Wextra -fopenmp -o $BIN_DIR/omp1 $SRC_DIR/omp1.cpp
```

```
g++ -Wall -Wextra -fopenmp -o $BIN_DIR/omp2 $SRC_DIR/omp2.cpp
```

```
if [ $? -eq 0 ]; then
```

```
    echo "file compiled in [$BIN_DIR] directory"
```

```
else
```

```
    echo "error"
```

## Приложение 5. Программа для построения графиков

```
import matplotlib.pyplot as plt
import numpy as np

with open("output.txt", "r") as file:
    data = file.readline().strip()

measurements = list(map(float, data.split()))

x_positions = np.arange(len(measurements))
x_labels = [1, 2, 4, 8, 16, 24]

plt.figure(figsize=(10, 6))
plt.plot(x_positions, measurements, marker='o', linestyle='-', color='blue')

plt.title("График измерений")
plt.xlabel("Количество фрагментов")
plt.ylabel("Измеренные значения")

plt.xticks(x_positions, x_labels, rotation=45)

plt.grid(True)
plt.tight_layout()
plt.show()
```

## Приложение 6. Программа для поиска наилучших параметров schedule

```
#include <iostream>
#include <vector>
#include <cmath>
#include <ctime>
#include <omp.h>
#include <numeric>

#define EPS 0.00001
#define TAU (1.9 / (size + 1))

#ifndef SCHEDULE_TYPE
    #define SCHEDULE_TYPE static
#endif

#ifndef CHUNK_SIZE
    #define CHUNK_SIZE 10
#endif

std::vector<double> mulMatrixVector(const
std::vector<std::vector<double>>& matrix, const std::vector<double>&
vec, int size) {
    std::vector<double> result(size, 0.0);

    #pragma omp parallel for schedule(SCHEDULE_TYPE, CHUNK_SIZE)
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            result[i] += matrix[i][j] * vec[j];
        }
    }
    return result;
}

std::vector<double> subVector(const std::vector<double>& vec1, const
std::vector<double>& vec2, int size) {
    std::vector<double> result(size);

    #pragma omp parallel for schedule(SCHEDULE_TYPE, CHUNK_SIZE)
    for (int i = 0; i < size; ++i) {
        result[i] = vec1[i] - vec2[i];
    }
    return result;
}
```

```

void mulConstVector(std::vector<double>& vec, double num, int size) {
    #pragma omp parallel for schedule(SCHEDULE_TYPE, CHUNK_SIZE)
    for (int i = 0; i < size; ++i) {
        vec[i] *= num;
    }
}

void initMatrix(std::vector<std::vector<double>>& matrix, int size) {
    #pragma omp parallel for schedule(SCHEDULE_TYPE, CHUNK_SIZE)
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            matrix[i][j] = (i == j) ? 2 : 1;
        }
    }
}

void initVector(std::vector<double>& vec, int size, double value) {
    #pragma omp parallel for schedule(SCHEDULE_TYPE, CHUNK_SIZE)
    for (int i = 0; i < size; ++i) {
        vec[i] = value;
    }
}

void findGoodSolve(const std::vector<std::vector<double>>& A,
std::vector<double>& X, const std::vector<double>& B, int size) {
    std::vector<double> AX_B = subVector(mulMatrixVector(A, X, size),
B, size);

    while (sqrt(std::inner_product(AX_B.begin(), AX_B.end(),
AX_B.begin(), 0.0)) / sqrt(std::inner_product(B.begin(), B.end(),
B.begin(), 0.0)) >= EPS) {
        mulConstVector(AX_B, TAU, size);
        X = subVector(X, AX_B, size);
        AX_B = subVector(mulMatrixVector(A, X, size), B, size);
    }
}

int main() {
    int size = 1000;
    int T = 4;

    omp_set_num_threads(T);

```

```
    std::vector<std::vector<double>> A(size,
std::vector<double>(size));
    std::vector<double> X(size);
    std::vector<double> B(size);

    initMatrix(A, size);
    initVector(B, size, size + 1);
    initVector(X, size, 0.0);

    double start = omp_get_wtime();
    findGoodSolve(A, X, B, size);
    double end = omp_get_wtime();

    std::cout << "\nTime: " << end - start << " seconds" << std::endl;
    std::cout << "Answer (some nums from vector X):" << std::endl;
    std::cout << X[0] << " " << X[size/3] << " " << X[size/2] << " " <<
X[size-1] << std::endl;

    return 0;
}
```