###### 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, 23210 группы

**Лаухина Егора Денисовича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Артюхов А.А.

Новосибирск 2025

[ЦЕЛЬ 3](#_Toc99151371)

[ЗАДАНИЕ 3](#_Toc99151372)

[ХОД РАБОТЫ 4](#_Toc99151373)

[ЗАКЛЮЧЕНИЕ 5](#_Toc99151374)

[Приложение 1. *Таблицы и графики* 6](#_Toc99151375)

[Приложение 2. *Код программы 1 (без schedule)* 7](#_Toc99151376)

[Приложение 3. *Код программы 2 (без schedule)* 11](#_Toc99151377)

[Приложение 4. *Код программы (c schedule)* 11](#_Toc99151377)

# ***ЦЕЛЬ***

Ознакомиться с принципами работы OpenMP. Реализовать программу решения системы линейных алгебраических уравнений, используя функционал OpenMP. Оценить эффективность и ускорение программ при различном числе потоков. Определить наиболее подходящий способ разделения данных между процессами

# ***ЗАДАНИЕ***

1. Распараллелить программу (последовательную из лабораторной работы №1) с помощью OpenMP двумя вариантами:
   * 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 двух вариантов программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 12, 16. Исходные данные выбрать так, чтобы программа работала на одном ядре не менее 30 секунд.
3. Провести исследование на определение оптимальных параметров #pragma omp for schedule при фиксированном размере задачи и количестве потоков.

# *ХОД РАБОТЫ*

Из лабораторной работы была позаимствована программа последовательного алгоритма решения СЛАУ методом минимальных невязок, которая впоследствии была распараллелена при помощи директив OpenMP. В последующем все варианты алгоритмов были подвергнуты тестированию на корректность решения при разном числе процессов и разных размерах исходной матрицы (параметр N). Также были проведены замеры времени исполнения каждой программы при варьировании числа процессов и при одном размере матрицы для всех вариантов. Предварительно был подобран параметр N (размер матрицы) так, чтобы время исполнения последовательной программы не было меньше 30 секунд. На основании полученных данных были построены таблицы и графики времени исполнения, эффективности и ускорения в зависимости от числа процессов при одинаковом параметре N (см. Приложение).

Параметр N = 1575, 4 потока для schedule

|  |  |
| --- | --- |
| **Параметр** | **Время** |
| static | 33.792 |
| dynamic | 56.727 |
| guided | 17.1644 |

|  |  |  |  |
| --- | --- | --- | --- |
| ***OpenMP1*** | | | |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 63,6917 | 100% | 1 |
| 2 | 38,1346 | 83% | 1,67 |
| 4 | 37,6837 | 42% | 1,69 |
| 8 | 21,2612 | 37% | 3 |
| 12 | 18,5667 | 28% | 3,43 |
| ***OpenMP2*** | | | |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 65,0227 | 100% | 1 |
| 2 | 44,8722 | 72% | 1,44 |
| 4 | 41,0523 | 40% | 1,58 |
| 8 | 37,3969 | 21% | 1,7 |
| 12 | 30,1316 | 17% | 3,15 |

# ***ЗАКЛЮЧЕНИЕ***

После анализа данных, полученных в результате замеров времени исполнения программ, в которых используются OpenMP-команды, можно сделать вывод о том, что использование функций библиотеки OMP и исполнения программы на большом количестве процессов может оказаться довольно эффективным методом ускорения исходной программы

***График***

***Приложение 2. Несколько параллельных секций***

#include <iostream>  
#include <vector>  
#include <cmath>  
#include <omp.h>  
  
#define Nx 35  
#define Ny 45  
#define N (Nx \* Ny)  
  
using namespace std;  
  
double scalar(vector<double>& y, vector<vector<double>>& A, vector<double>& multi) {  
 #pragma omp parallel for  
 for (int i = 0; i < N; i++) {  
 multi[i] = 0;  
 for (int j = 0; j < N; j++) {  
 multi[i] += A[i][j] \* y[j];  
 }  
 }  
  
 double local\_numerator = 0, local\_denominator = 0;  
 #pragma omp parallel for reduction(+:local\_numerator, local\_denominator)  
 for (int i = 0; i < N; i++) {  
 local\_numerator += y[i] \* multi[i];  
 local\_denominator += multi[i] \* multi[i];  
 }  
 return local\_numerator / local\_denominator;  
}  
  
void method(vector<vector<double>>& A, vector<double>& x, vector<double>& b) {  
 vector<double> y(N, 0);  
 vector<double> multi(N, 0);  
  
 #pragma omp parallel for  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 y[i] += A[i][j] \* x[j];  
 }  
 y[i] -= b[i];  
 }  
  
 double tau = scalar(y, A, multi);  
  
 #pragma omp parallel for  
 for (int i = 0; i < N; i++) {  
 x[i] -= y[i] \* tau;  
 }  
}  
  
double get\_test(vector<vector<double>>& A, vector<double>& x, vector<double>& b) {  
 vector<double> help(N, 0);  
  
 #pragma omp parallel for  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 help[i] += A[i][j] \* x[j];  
 }  
 help[i] -= b[i];  
 }  
  
 double local\_numerator = 0, local\_denominator = 0;  
 #pragma omp parallel for reduction(+:local\_numerator, local\_denominator)  
 for (int i = 0; i < N; i++) {  
 local\_numerator += pow(help[i], 2);  
 local\_denominator += pow(b[i], 2);  
 }  
 return sqrt(local\_numerator) / sqrt(local\_denominator);  
}  
  
void initialize(vector<vector<double>>& A, vector<double>& b) {  
 #pragma omp parallel for  
 for (int i = 0; i < N; i++) {  
 A[i][i] = -4.0;  
 if (i % Nx != 0) {  
 A[i][i - 1] = 1.0;  
 }  
 if ((i + 1) % Nx != 0) {  
 A[i][i + 1] = 1.0;  
 }  
 if (i >= Nx) {  
 A[i][i - Nx] = 1.0;  
 }  
 if (i < N - Nx) {  
 A[i][i + Nx] = 1.0;  
 }  
 if (i%2 == 0) {  
 b[i] = (i+63) % 50;  
 }  
 else {  
 b[i] = -(i+63) % 50;  
 }  
 }  
}  
  
void start() {  
 double epsilon = 1e-5;  
 vector<vector<double>> A(N, vector<double>(N, 0.0));  
 vector<double> b(N, 0.0);  
 vector<double> x(N, 0.0);  
  
 initialize(A, b);  
 double test = 1e10;  
 while (test > epsilon) {  
 method(A, x, b);  
 test = get\_test(A, x, b);  
 }  
// std::cout << std::endl << "x" << std::endl;  
// for (int i = 0; i < N; i++) {  
// std::cout << x[i] << " ";  
// }  
// std::cout << std::endl;  
 std::cout << "All is good" << std::endl;  
}  
  
int main() {  
 double startTime = omp\_get\_wtime();  
 start();  
 double endTime = omp\_get\_wtime();  
 cout << "Время выполнения: " << (endTime - startTime) << " с" << endl;  
 return 0;  
}

***Приложение 3. Одна параллельная секция***

#include <iostream>  
#include <cmath>  
#include <vector>  
#include <omp.h>  
#define Nx 35  
#define Ny 45  
#define N (Nx\*Ny)  
using namespace std;  
  
  
double scalyar(vector<double>& y, vector<vector<double>>& A, vector<double>& multi) {  
 double numerator = 0;  
 double denominator = 0;  
  
#pragma omp parallel for reduction(+:numerator, denominator)  
 for (int i = 0; i < N; i++) {  
 multi[i] = 0;  
 for (int j = 0; j < N; j++) {  
 multi[i] += A[i][j] \* y[j];  
 }  
 numerator += y[i] \* multi[i];  
 denominator += multi[i] \* multi[i];  
 }  
 return numerator/denominator;  
}  
  
  
  
  
  
void intialize(vector<vector<double>>& A, vector<double>& b) {  
  
  
 for (int i = 0; i < N; i++) {  
 A[i][i] = -4.0;  
 if (i % Nx != 0) {  
 A[i][i - 1] = 1.0;  
 }  
 if ((i + 1) % Nx != 0) {  
 A[i][i + 1] = 1.0;  
 }  
 if (i >= Nx) {  
 A[i][i - Nx] = 1.0;  
 }  
 if (i < N - Nx) {  
 A[i][i + Nx] = 1.0;  
 }  
 if (i % 2 == 0) {  
 b[i] = (i + 63) % 50;  
 } else {  
 b[i] = -(i + 63) % 50;  
 }  
  
 }  
}  
  
  
  
  
  
void start() {  
 double epsilon = 1e-5;  
 vector<vector<double>> A(N, vector<double>(N, 0.0));  
 vector<double> b(N, 0.0);  
 vector<double> x(N, 0.0);  
 double tau = 0;  
 intialize(A, b);  
 double test = pow(10, 10);  
 vector<double> multi(N, 0);  
 vector<double> y(N, 0);  
 vector<double> help(N, 0);  
#pragma omp parallel  
 {  
 while (epsilon < test) {  
 double local\_num = 0, local\_den = 0, numerator = 0, denominator = 0;  
#pragma omp single  
 {  
#pragma omp taskloop  
 for (int i = 0; i < N; i++) {  
 y[i] = 0;  
 for (int j = 0; j < N; j++) {  
 y[i] += A[i][j] \* x[j];  
 }  
 y[i] -= b[i];  
 }  
 }  
#pragma omp single  
 {  
 tau = scalyar(y, A, multi);  
 }  
#pragma omp for  
 for (int i = 0; i < N; i++) {  
 x[i] -= y[i] \* tau;  
 }  
#pragma omp single  
 {  
#pragma omp taskloop  
 for (int i = 0; i < N; i++) {  
 help[i] = 0;  
 for (int j = 0; j < N; j++) {  
 help[i] += A[i][j] \* x[j];  
 }  
 help[i] -= b[i];  
 }  
 }  
#pragma omp for nowait  
 for (int i = 0; i < N; i++) {  
 local\_num += help[i] \* help[i];  
 local\_den += b[i] \* b[i];  
 }  
  
#pragma omp critical  
 {  
 numerator += local\_num;  
 denominator += local\_den;  
 }  
  
#pragma omp barrier  
#pragma omp single  
 test = sqrt(numerator) / sqrt(denominator);  
 }  
 }  
// cout << endl << "x" << endl;  
// for (int i = 0; i < N; i++) {  
// cout << x[i] << " ";  
// }  
 cout << endl << "all is good" << endl;  
}  
  
  
  
int main() {  
 double startTime = omp\_get\_wtime();  
 start();  
 double endTime = omp\_get\_wtime();  
 cout << "Время выполнения: " << (endTime - startTime) << " с" << endl;  
 return 0;  
}

***Приложение 4. С параметрами schedule***

#include <iostream>  
#include <vector>  
#include <cmath>  
#include <omp.h>  
  
#define Nx 35  
#define Ny 45  
#define N (Nx \* Ny)  
  
using namespace std;  
  
double scalar(vector<double>& y, vector<vector<double>>& A, vector<double>& multi) {  
#pragma omp parallel for schedule(dynamic, 4)  
 for (int i = 0; i < N; i++) {  
 multi[i] = 0;  
 for (int j = 0; j < N; j++) {  
 multi[i] += A[i][j] \* y[j];  
 }  
 }  
  
 double local\_numerator = 0, local\_denominator = 0;  
#pragma omp parallel for schedule(dynamic, 4) reduction(+:local\_numerator, local\_denominator)  
 for (int i = 0; i < N; i++) {  
 local\_numerator += y[i] \* multi[i];  
 local\_denominator += multi[i] \* multi[i];  
 }  
 return local\_numerator / local\_denominator;  
}  
  
void method(vector<vector<double>>& A, vector<double>& x, vector<double>& b) {  
 vector<double> y(N, 0);  
 vector<double> multi(N, 0);  
  
#pragma omp parallel for schedule(dynamic, 4)  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 y[i] += A[i][j] \* x[j];  
 }  
 y[i] -= b[i];  
 }  
  
 double tau = scalar(y, A, multi);  
  
#pragma omp parallel for schedule(dynamic, 4)  
 for (int i = 0; i < N; i++) {  
 x[i] -= y[i] \* tau;  
 }  
}  
  
double get\_test(vector<vector<double>>& A, vector<double>& x, vector<double>& b) {  
 vector<double> help(N, 0);  
  
#pragma omp parallel for schedule(dynamic, 4)  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 help[i] += A[i][j] \* x[j];  
 }  
 help[i] -= b[i];  
 }  
  
 double local\_numerator = 0, local\_denominator = 0;  
#pragma omp parallel for schedule(dynamic, 4) reduction(+:local\_numerator, local\_denominator)  
 for (int i = 0; i < N; i++) {  
 local\_numerator += pow(help[i], 2);  
 local\_denominator += pow(b[i], 2);  
 }  
 return sqrt(local\_numerator) / sqrt(local\_denominator);  
}  
  
void initialize(vector<vector<double>>& A, vector<double>& b) {  
 for (int i = 0; i < N; i++) {  
 A[i][i] = -4.0;  
 if (i % Nx != 0) {  
 A[i][i - 1] = 1.0;  
 }  
 if ((i + 1) % Nx != 0) {  
 A[i][i + 1] = 1.0;  
 }  
 if (i >= Nx) {  
 A[i][i - Nx] = 1.0;  
 }  
 if (i < N - Nx) {  
 A[i][i + Nx] = 1.0;  
 }  
 if (i%2 == 0) {  
 b[i] = (i+63) % 50;  
 }  
 else {  
 b[i] = -(i+63) % 50;  
 }  
 }  
}  
  
void start() {  
 double epsilon = 1e-5;  
 vector<vector<double>> A(N, vector<double>(N, 0.0));  
 vector<double> b(N, 0.0);  
 vector<double> x(N, 0.0);  
  
 initialize(A, b);  
 double test = 1e10;  
 while (test > epsilon) {  
 method(A, x, b);  
 test = get\_test(A, x, b);  
 }  
// std::cout << std::endl << "x" << std::endl;  
// for (int i = 0; i < N; i++) {  
// std::cout << x[i] << " ";  
// }  
// std::cout << std::endl;  
 std::cout << "All is good" << std::endl;  
}  
  
int main() {  
 double startTime = omp\_get\_wtime();  
 start();  
 double endTime = omp\_get\_wtime();  
 cout << "Время выполнения: " << (endTime - startTime) << " с" << endl;  
 return 0;  
}