###### 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, 21203 группы

**Неретина Степана Ивановича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЗАДАНИЕ 3](#_heading=h.gjdgxs)

[ОПИСАНИЕ РАБОТЫ 4](#_heading=h.30j0zll)

Вывод9

[Приложение 1. Последовательная программа](#_heading=h.1fob9te)  10

Приложение 2. Параллельная программа 15

# ЗАДАНИЕ

1. Написать программу на языке C или C++, 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. Замерить время работы 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.
3. Провести исследование на определение оптимальных параметров #pragma  
   omp for schedule(...) при некотором фиксированном размере задачи и  
   количестве потоков.

# ОПИСАНИЕ РАБОТЫ

# Написан код (Приложение 1, 2), который реализует поставленную задачу(решение слау методом минимальных невязок).

Компиляция производилась командой gcc с флагом fopenmp для параллельной, а для последовательной при помощи gcc

Тестирование производилось на кластере. Подключение осуществлялось

посредством протокола ssh с логином и паролем.

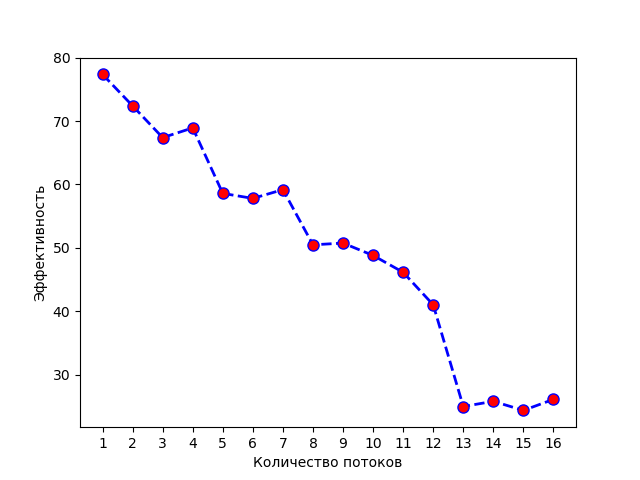
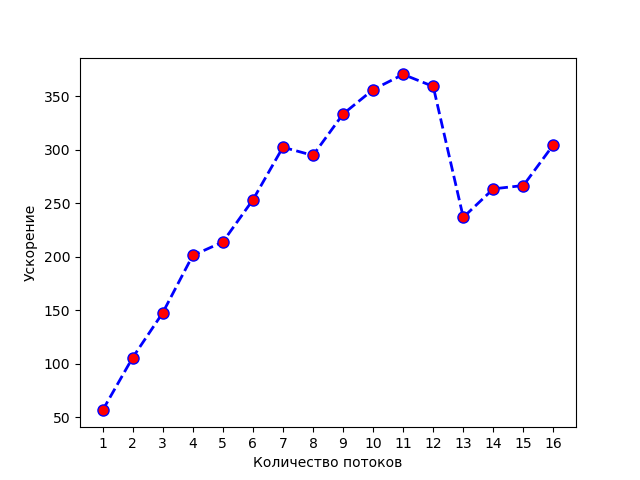
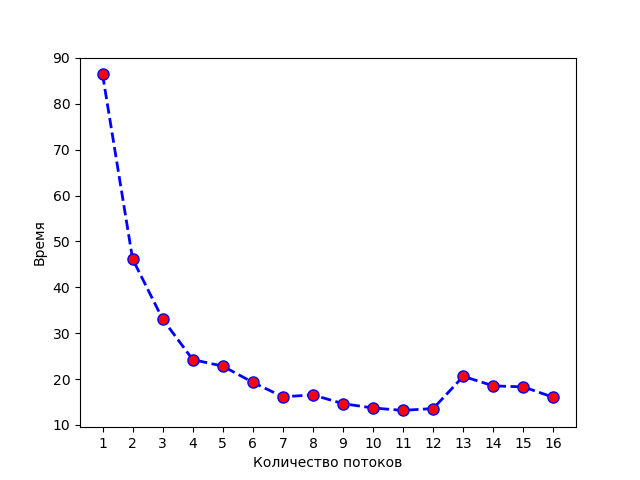
Для выполнения условия «... решение задачи на одном ядре занимало не

менее 30 секунд.» подобраны значения N = 1000, ε = 10^(-5), ITERATIONS\_COUNT 10000

В результате замеров получены следующие результаты:

|  |  |  |  |
| --- | --- | --- | --- |
| N | итерации | время | относительное время |
| 1 | 8236 | 86.397421 | 1.183526 |
| 2 | 8235 | 46.159977 | 0.632328 |
| 3 | 8204 | 33.061125 | 0.452892 |
| 4 | 8060 | 24.244829 | 0.332121 |
| 5 | 8571 | 22.817171 | 0.312564 |
| 6 | 7949 | 19.274066 | 0.264028 |
| 7 | 8230 | 16.134119 | 0.221015 |
| 8 | 8256 | 16.552661 | 0.226749 |
| 9 | 7888 | 14.634177 | 0.200468 |
| 10 | 8047 | 13.69609 | 0.187618 |
| 11 | 8024 | 13.170559 | 0.180419 |
| 12 | 8044 | 13.583031 | 0.186069 |
| 13 | 8173 | 20.595419 | 0.282129 |
| 14 | 8217 | 18.515873 | 0.253642 |
| 15 | 8937 | 18.299211 | 0.250674 |
| 16 | 8093 | 16.009136 | 0.219303 |

**Графики**



# Исследование Schedule:

# 

# 

# 

# **Вывод**

OpenMP - простой и мощный инструмент для написания параллельных программ. Множество различных директив предоставляют широкий набор инструментов для программиста, а инкапсуляция позволяет избежать множества  
ошибок (и частично - дебага параллельных программ)

# **Приложение 1. *Последовательная программа***

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <math.h>

#include <stdbool.h>

#define N 1000

#define START (-100)

#define STOP (100)

#define ITERATIONS\_COUNT 10000

double \* generate\_double\_arr(size\_t n){

return (double \*) malloc(n \* sizeof(double));

}

double generate\_random\_in\_range(double min, double max) {

double range = (max - min);

double div = RAND\_MAX / range;

return min + (rand() / div);

}

double \*generate\_vector\_with\_filled\_value(double value){

double \* res = generate\_double\_arr(N);

for(size\_t i = 0; i < N; i++){

res[i] = value;

}

return res;

}

double \*fill\_vector(){

double \* res = generate\_double\_arr(N);

for(size\_t i = 0; i < N; i++){

res[i] = i\*i;

}

return res;

}

double \*generate\_vector\_with\_random\_value(){

double \* res = generate\_double\_arr(N);

for(size\_t i = 0; i < N; i++){

res[i] = generate\_random\_in\_range(START, STOP);

}

return res;

}

double\* generate\_matrix\_a() {

double \*res = generate\_double\_arr(N\*N);

for (size\_t i = 0; i < N; i++) {

for(size\_t j = 0; j < N; j++){

//res[i]= generate\_random\_in\_range(START, STOP);

res[j\*N +i] = (i == j ? i : N);

// if(i == j){

// //res[i\*N + j] += (N / 8.0);

// res[i\*N + j] = (N\*8);

// }

}

}

// for (size\_t i = 0; i < N; i++) {

// for(size\_t j = 0; j < N; j++){

// if(i == j){

// res[i\*N + j] =2;

// }else{

// res[i\*N + j] =1;

// }

// }

// }

return res;

}

void print\_matrix(double \*arr, size\_t size){

for(size\_t i = 0; i < size; i++){

for(size\_t j = 0; j < size; j++){

printf("%lf ", arr[i\*size + j]);

}

printf("\n");

}

printf("\n\n");

}

void print\_vec(double \*vec, size\_t size){

for(size\_t i = 0; i < size; i++){

printf("%lf\n", vec[i]);

}

printf("\n\n");

}

double scalar\_dot(const double \*vec1, const double \*vec2, size\_t size){

double res = 0;

for(size\_t i = 0; i < size; i++){

res += vec1[i]\*vec2[i];

}

return res;

}

void multiple\_vec\_with\_num(const double \*vec, double num, double \*res, size\_t size){

for(size\_t i = 0; i < size; i++){

res[i] = vec[i] \* num;

}

}

double calculate\_squared\_norm(double \*vec, size\_t size){

double res = 0;

for(size\_t i = 0; i < size; i++){

res += (double)powl(vec[i], 2);

}

return res;

}

bool check\_convergence(double \*left, double right\_part){

return calculate\_squared\_norm(left, N) < right\_part;

}

double calculate\_right\_part(double \*b){

double epsilon = powl(10, -5);

double a = calculate\_squared\_norm(b, N);

return a \* epsilon \* epsilon;

}

void subtract\_vectors(const double \*vec1, const double \*vec2, double \*res, int size){

for(size\_t i = 0; i < (size\_t)size; i++){

res[i] = vec1[i] - vec2[i];

}

}

void multiple\_matrix\_and\_vector(const double \*matrix, const double \*vec, double \*res){

memset(res, 0, sizeof(double) \* N);

for(size\_t i = 0; i < N; i++) {

for (size\_t j = 0; j < N; j++) {

res[j] += matrix[j\*N + i] \* vec[i];

}

}

}

void calculate\_vector\_y(const double \*matrix\_a, double\* x, double \*b, double \*res) {

multiple\_matrix\_and\_vector(matrix\_a, x, res);

subtract\_vectors(res, b, res, N);

}

double \*calculate\_ay\_n(double \*a, double \*y) {

double \*res = generate\_double\_arr(N);

multiple\_matrix\_and\_vector(a,y, res);

return res;

}

double tau\_update(double \*a, double \*y) {

double \*ay\_n = calculate\_ay\_n(a, y);

double numerator = scalar\_dot(y, ay\_n, N);

double denumerator = scalar\_dot(ay\_n, ay\_n, N);

return numerator / denumerator;

}

void save\_matrix\_to\_file(double\* matrix, char\* filename) {

// Open the file for writing

FILE\* file = fopen(filename, "w");

// Write the matrix dimensions to the file

fprintf(file, "%d ", N);

// Write the matrix elements to the file

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

fprintf(file, "%f ", matrix[i\*N + j]);

}

//fprintf(file, "\n");

}

// Close the file

fclose(file);

}

void save\_vector\_to\_file(const char\* filename, double \* vector) {

FILE\* fp = fopen(filename, "w");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return;

}

fprintf(fp, "%d ", N);

for (int i = 0; i < N; i++) {

fprintf(fp, "%f\n", vector[i]);

}

fclose(fp);

}

double\* read\_matrix\_from\_file(const char\* filename) {

FILE\* fp = fopen(filename, "r");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return NULL;

}

size\_t n;

fscanf(fp, "%zd", &n);

char c = fgetc(fp);

// Allocate memory for the matrix

double\* arr = generate\_double\_arr(n\*n);

// Read the matrix elements from the file

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

double res;

fscanf(fp, "%lf", &res);

arr[i\* n + j] = res;

fgetc(fp);

}

}

fclose(fp);

return arr;

}

double\* read\_vector\_from\_file(const char\* filename) {

FILE\* fp = fopen(filename, "r");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return NULL;

}

size\_t n;

fscanf(fp, "%zd", &n);

fgetc(fp);

// Allocate memory for the matrix

double\* arr = generate\_double\_arr(n);

// Read the matrix elements from the file

for (int i = 0; i < n; i++) {

double res;

fscanf(fp, "%lf", &res);

arr[i] = res;

fgetc(fp);

}

fclose(fp);

return arr;

}

int main() {

srand (time ( NULL));

double\* a = generate\_matrix\_a();

double\* b = fill\_vector();

double right\_part = calculate\_right\_part(b);

//printf("Right part %f\n", right\_part);

double\* x = generate\_vector\_with\_filled\_value(0);

double \*y = generate\_double\_arr(N);

double \*tau\_y = generate\_double\_arr(N);

// print\_matrix(a, N);

// print\_vec(b, N);

calculate\_vector\_y(a, x, b, y);

for(size\_t i = 0; i < ITERATIONS\_COUNT; i++){

//printf("%zu\n", i+1);

double tau = tau\_update(a, y);

//printf("Tau %f\n", tau);

multiple\_vec\_with\_num(y, tau, tau\_y, N);

subtract\_vectors(x, tau\_y, x, N);

//print\_vec(x, N);

calculate\_vector\_y(a, x, b, y);

if(check\_convergence(y, right\_part)){

printf("Finished on: %zu\n", i+1);

// if(i+1 > 1500){

// save\_matrix\_to\_file(a, "matrix.txt");

// save\_vector\_to\_file("vector.txt", b);

// }

break;

}

}

//print\_vec(x, N);

free(a);

free(b);

free(x);

free(y);

free(tau\_y);

return 0;

}

# **Приложение 2. *Параллельная программа.***

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#define N 1000

#define START (-100)

#define STOP (100)

#define ITERATIONS\_COUNT 10000

double \* generate\_double\_arr(size\_t n){

return (double \*) malloc(n \* sizeof(double));

}

double \*fill\_vector(){

double \* res = generate\_double\_arr(N);

for(size\_t i = 0; i < N; i++){

res[i] = i\*i;

}

return res;

}

void fill\_vector\_with\_value(double \*res, double value) {

for (size\_t i = 0; i < N; i++) {

res[i] = value;

}

}

double\* generate\_matrix\_a() {

double \*res = generate\_double\_arr(N\*N);

for (size\_t i = 0; i < N; i++) {

for(size\_t j = 0; j < N; j++){

res[j\*N +i] = (i == j ? (double)i : N);

}

}

return res;

}

void calculate\_squared\_norm(const double \*b, double \*norm\_b\_square) {

}

int main(int argc, char\* argv[]) {

double start;

double end;

start = omp\_get\_wtime();

double \*a = generate\_matrix\_a();

double \*x = generate\_double\_arr(N);

fill\_vector\_with\_value(x, 0);

double \*b = fill\_vector();

double \*y = generate\_double\_arr(N);

double norm\_b\_square = 0;

double norm\_y\_square = 0;

double right\_part = 0;

double \*vec\_rec = generate\_double\_arr(N);

double tau = 0;

double left\_part = 0;

double denominator;

double numerator;

size\_t k;

#pragma omp parallel shared(denominator,a, x, b, norm\_b\_square, right\_part, y, vec\_rec, numerator, tau, norm\_y\_square), private(k) default(none)

{

#pragma omp for reduction(+:norm\_b\_square)

for (size\_t i = 0; i < N; i++) {

norm\_b\_square += b[i]\*b[i];

}

#pragma omp single

{

double epsilon = pow(10, -5);

right\_part = epsilon \* epsilon \* norm\_b\_square;

printf("Right part %f\n", right\_part);

}

#pragma omp barrier

#pragma omp for

for (size\_t i = 0; i < N; i++) {

y[i] = 0.0;

for (size\_t j = 0; j < N; j++) {

y[i] += a[i\*N+j] \* x[j];

}

y[i] -= b[i];

}

for(k = 0; k < ITERATIONS\_COUNT; k++){

#pragma omp single

{

denominator = 0;

numerator = 0;

norm\_y\_square = 0;

}

#pragma omp for

for (size\_t i = 0; i < N; i++) {

double part = 0;

for (size\_t j = 0; j < N; j++) {

part += a[i \* N + j] \* y[j];

}

vec\_rec[i] = part;

part = pow(part, 2);

#pragma omp atomic

denominator += part;

}

#pragma omp for reduction(+:numerator)

for (size\_t i = 0; i < N; i++) {

numerator += vec\_rec[i]\*y[i];

}

#pragma omp single

tau = numerator / denominator;

#pragma omp for

for(size\_t j = 0; j < N; j++){

x[j] = x[j] - tau\*y[j];

}

#pragma omp for

for (size\_t i = 0; i < N; i++) {

y[i] = 0.0;

for (size\_t j = 0; j < N; j++) {

y[i] += a[i\*N+j] \* x[j];

}

y[i] -= b[i];

}

#pragma omp for reduction(+:norm\_y\_square)

for (size\_t i = 0; i < N; i++) {

norm\_y\_square += y[i]\*y[i];

}

if(norm\_y\_square < right\_part){

if(omp\_get\_thread\_num() == 0){

printf("Finished on: %zu\n", k+1);

}

break;

}

#pragma omp barrier

}

}

free(a);

free(b);

free(x);

free(y);

free(vec\_rec);

end = omp\_get\_wtime();

printf("Work took %f seconds. Relative time: %f\n", end - start, (end-start)/k);

}