**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**

Лабораторная работа 2

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

Студента 2 курса, 19210 группы

**Пирожков Андрей Константинович**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

П.О.Холявко

Новосибирск 2021

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc67597921)

[ЗАДАНИЕ 3](#_Toc67597922)

[ВАРИАНТ 3](#_Toc67597923)

[ОПИСАНИЕ РАБОТЫ 5](#_Toc67597924)

[ЗАКЛЮЧЕНИЕ 8](#_Toc67597925)

[Приложение 1 *Листинг файла lab2.c* 9](#_Toc67597926)

[Приложение 2 *Листинг файла lab2\_1.c* 12](#_Toc67597927)

[Приложение 3 *Листинг файла lab2\_2.c* 16](#_Toc67597928)

# ЦЕЛЬ

* Изучение библиотеки OpenMP как средство распараллеливания программы на работающие части
* Проанализировать полученную программу и выявить зависимость производительности программы от входных данных, варианта решения, количество ядер

# ЗАДАНИЕ

1. Последовательную программу из лабораторной работы 1, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида , 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).

1. 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
2. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков
3. На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы.

# ВАРИАНТ

***Метод минимальных невязок.***

В методе минимальных невязок преобразование решения на каждом шаге задается следующими формулами:

В отличие от метода простой итерации, метод минимальных невязок содержит два умножения матрицы на вектор на каждой итерации, однако сходится до нужной точности за меньшее число итераций. Критерий завершения счета можно взять такой же, как в методе простой итерации:

# **ОПИСАНИЕ РАБОТЫ**

Эту лабораторную работу я сделал на основе предыдущей, как и было сказано в задании.

В первом варианте я просто взял код без OpenMP, и подставил почти перед всеми циклами директиву «#pragma omp parallel for». В двух циклах я ещё использовал редукцию, так как нужно было получить сумму. А в циклах, которые генерируют рандомную матрицу и вектор не использовал, так как матрица может получиться слишком однотипной.

Во втором варианте мне пришлось полностью переделать код. Все функции, которые я так красиво расставлял, пришлось объединить в одно целое. У меня возникало много ошибок, и наилучшим вариантом я посчитал – поместить всё в «int main()». После этого, программа начала верно считать матрицу. Далее я очень много циклов объединил, для того чтобы код смотрелся более компактным и работал максимально быстро.

Также я переставил таймеры, которые фиксируют время работы программы непосредственно сразу перед началом итерационного алгоритма и сразу после него. Также сначала я вывожу несколько правильных элементов вектора , и в конце тоже, чтобы приблизительно сравнить, насколько верно считает мой алгоритм. Однако, я его закомментировал, так как он мне не сильно нужен, когда я был уверен, что программа считает верно.

Компилировал компилятором icc, потому что с gcc выдавал какие-то странные ошибки. Также отметил, что программа скомпилированная icc в 10 раз быстрее, чем скомпилированная gcc. Даже с ключом оптимизации «-O3» icc выполняет программу в 2,5 раза быстрее. Ключи для оптимизации icc компилятора никак не влияют на его производительность.

Мне пришлось поменять входные данные для этой лабораторной работы, потому что если матрица маленькая, то наблюдается странная тенденция: чем больше потоков – тем медленнее программа работает. Поэтому на одном потоке программа работала уже максимально быстро и смысл этой лабораторной работы вообще пропадал. Принял решение увеличить матрицу до . Далее поменял ключ для рандома на 278, т.е. . При таких входных данных получатся минимальное количество итераций в цикле, из-за чего я могу приблизить точность вычислений до: .

Таблица с компиляцией программы:

|  |  |  |
| --- | --- | --- |
| Файл | Описание | Команда компиляции |
| [lab2.c](#_Приложение_1) | Исходный файл без OpenMP | icc lab1.c -std=gnu99 -o lab1.exe -lm |
| [lab2\_1.c](#_Приложение_2_1) | Вариант 1-ый с OpenMP | icc -std=gnu99 -openmp -o lab2\_1.exe lab2\_1.c |
| [lab2\_2.c](#_Приложение_3) | Вариант 2-ой с OpenMP | icc -std=gnu99 -openmp -o lab2\_2.exe lab2\_2.c |

Пояснение к команде компиляции:

* -std=gnu99 – используемый мною стандарт языка си
* -lm – позволяет использовать sqrt() из библиотеки “math.h”
* -openmp – подключаем библиотеку с OpenMP

Пояснение что за варианты программ:

* Вариант 1-ый с OpenMP – здесь применяются только директивы ко всем циклам «#pragma omp parallel for»
* Вариант 2-ой с OpenMP – тут создаётся один блок «#pragma omp parallel» на весь итерационный алгоритм

Таблица с результатами:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество потоков | Результаты по времени | | Ускорение и эффективность | |
| lab1\_1 | lab2\_2 | lab1\_1 | lab2\_2 |
| без OpenMP | 36,79675 | 36,79675 | 1.0 | 1.0 |
| 1 | 36,82919 | 36,64022 | 0,999119 | 1,004272 |
| 2 | 19,56553 | 19,26376 | 1,880693 | 1,910154 |
| 3 | 13,99825 | 16,51081 | 2,628667 | 2,228646 |
| 4 | 13,648 | 15,17833 | 2,696127 | 2,424295 |
| 5 | 11,73781 | 16,9977 | 3,13489 | 2,164808 |
| 6 | 11,43343 | 15,19865 | 3,218346 | 2,421054 |
| 7 | 12,21536 | 15,91388 | 3,012335 | 2,312242 |
| 8 | 12,7555 | 14,98613 | 2,884776 | 2,455387 |
| 9 | 23,90564 | 21,7863 | 1,539249 | 1,688986 |
| 10 | 28,8488 | 15,5314 | 1,275503 | 2,369185 |
| 11 | 23,39999 | 22,69537 | 1,572511 | 1,621333 |
| 12 | 34,4472 | 26,94024 | 1,068207 | 1,365865 |
| 13 | 23,96255 | 14,16133 | 1,535594 | 2,598396 |
| 14 | 23,60882 | 13,25773 | 1,558602 | 2,775494 |
| 15 | 24,58262 | 13,58055 | 1,49686 | 2,709519 |
| 16 | 25,26828 | 13,38085 | 1,456243 | 2,749956 |

Вот соответствующие графики к этой таблице (0 – это обозначение запуска программы без подключения OpenMP):

Я привел 2 типа графиков, чтобы было понятнее на них смотреть. Обычно нас интересует количество потоков кратное двум, однако, эту лабораторную работу необходимо было сделать на всех доступных потоках.

Исходя из полученных результатов, можно сказать, о том, что 1-ый вариант программы работает более эффективно и быстрее на потоках не более 8, а 2-ой вариант, наоборот, на 9 и более количества потоков. Если смотреть на упрощённые графики, кажется, что программа работает очень даже хорошо, особенно, 2ой-варинат, потому что его время постепенно сокращается и сокращается. Но если взглянуть на полные графики, то можно понять, что после 8-ми потоков, график идёт совсем нестабильно то вверх, то вниз. Возможно, это связано с тем, что процессор не способен разделять программу на 9 и более потоков. К тому же, я запускал 2 процессора и 8 ядер с каждого. Значит потоки создавались на разных процессорах, что, возможно, усложнило подсчёт времени.

Самое интересное, что на 16-и потоках во 2-ом варианте рекордное малое время. Т.е. всё-таки имеет смысл разделять более чем на 2 потока программу. Но всё-таки, самое лучшее разделение – это на 2 потока, потому что такое количество гарантирует увеличение производительности в 2 раза.

Профилирование программы нормально сделать не получилось, во-первых, нехватка памяти на кластере, а во-вторых, MPE не способен отличать потоки от процессов, поэтому сколько бы потоков я не создавал, он всё равно покажет только процесс нулевой. Однако я научился делать подсветку кода, точнее подсветку нужных мне частей кода. Но в этом нет смысла, если отображается только один поток, ведь никак нельзя будет сравнить с другими потоками.

# **ЗАКЛЮЧЕНИЕ**

В ходе лабораторной работы я научился работать с библиотекой OpenMP и распараллеливать программы на потоки с помощью неё. Также я проанализировал полученные графики. И по ним можно сделать вывод о том, что 1-ый вариант лучше работает до 8-ми потоков, а 2ой с 9-и до 16-и потоков. Лучший выбор по количеству потоков – это 2 или 4. Такое количество гарантированно увеличит скорость программы в 2 и более раз. Не стоит забывать, что слишком большое число потоков может привести к увеличению времени работы программы.

# Приложение 1

*Листинг файла lab2.c*

*Исходный код без OpenMP*

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <time.h>

//----------Operations----------

// |x|=sqrt(sum(x\_i^2))

double modul(double\* x, int N)

{

double sum = 0.0;

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

{

sum += x[i] \* x[i];

}

return sqrt(sum);

}

// (x,y)=sum(x\_i\*y\_i)

double skobki(double\* x, double\* y, int N)

{

double res = 0.0;

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

{

res += x[i] \* y[i];

}

return res;

}

//multiplicate matrix to vector: result = x \* y (N - size; x - matrix; result, y - vector)

void mul\_M\_v(double\* result, double\* x, double\* y, int N)

{

double\* res = (double\*)malloc(N \* sizeof(double));

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

{

res[i] = 0.0;

}

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

{

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

{

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

}

}

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

{

result[i] = res[i];

}

free(res);

}

//multiplicate constant to vector: result = c \* y (N - size; c - constant; result, y - vector)

void mul\_c\_v(double\* result, double c, double\* y, int N)

{

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

{

result[i] = c \* y[i];

}

}

//substraction vector to vector: result = x - y (N - size; result, x, y - vector)

void sub(double\* result, double\* x, double\* y, int N)

{

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

{

result[i] = x[i] - y[i];

}

}

//----------Setters/Getters/Print----------

void random\_vector(double\* x, int N)

{

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

{

x[i] = (double)(rand() % 18 - 9);

}

}

void symmetrical\_matrix(double\* A, int N)

{

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

{

A[i \* N + i] = (double)(rand() % 18 - 9);

}

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

{

for (int j = 0; j < i; j++)

{

A[i \* N + j] = (double)(rand() % 18 - 9);

A[j \* N + i] = A[i \* N + j];

}

}

}

void print\_matrix(double\* A, int N)

{

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

{

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

{

printf("%2g ", A[i \* N + j]);

}

printf("\n");

}

}

void print\_vector(double\* V, int N)

{

printf("(");

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

{

printf("%g ", V[i]);

}

printf(")\n");

}

//----------Algorithm----------

void y\_pow(double\* y, double\* A, double\* x, double\* b, int N)

{

double\* Ax = (double\*)malloc(N \* sizeof(double));

mul\_M\_v(Ax, A, x, N);

sub(y, Ax, b, N);

free(Ax);

}

double t\_pow(double\* y, double\* A, int N)

{

double\* Ay = (double\*)malloc(N \* sizeof(double));

mul\_M\_v(Ay, A, y, N);

double t = skobki(y, Ay, N) / skobki(Ay, Ay, N);

free(Ay);

return t;

}

void x\_pow(double\* y, double\* A, double\* x, double\* b, int N)

{

y\_pow(y, A, x, b, N);

double t = t\_pow(y, A, N);

double\* ty = (double\*)malloc(N \* sizeof(double));

mul\_c\_v(ty, t, y, N);

sub(x, x, ty, N);

free(ty);

}

int test(double\* y, double\* b, double epsilon, int N)

{

double t = modul(y, N) / modul(b, N);

if (t < epsilon)

{

return 1; //остановка

}

else

{

return 0; //продолжай, пока не 1

}

}

int main()

{

//Входные данные

srand(278);

int N = 1024;

double epsilon = 0.004;

double\* A = (double\*)malloc(N \* N \* sizeof(double));

double\* x = (double\*)malloc(N \* sizeof(double));

double\* b = (double\*)malloc(N \* sizeof(double));

double\* y = (double\*)malloc(N \* sizeof(double)); // y=Ax-b

printf("-; ");

//Создаём матрицу A и x, считаем b, а затем переписываем x (как раз далее будем этот вектор находить)

symmetrical\_matrix(A, N);

random\_vector(x, N);

mul\_M\_v(b, A, x, N);

random\_vector(x, N);

y\_pow(y, A, x, b, N);

//Фиксируем время начала

struct timespec start, finish;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

//Начало алгоритма

int i = 0; //счётчик итераций

while (test(y, b, epsilon, N) == 0)

{

x\_pow(y, A, x, b, N);

i++;

}

//Фиксируем время конца

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &finish);

printf("%d ;", i);

printf("%lf \n", (finish.tv\_sec - start.tv\_sec + 0.000000001 \* (finish.tv\_nsec - start.tv\_nsec)));

free(A);

free(x);

free(b);

free(y);

return 0;

}

# Приложение 2

*Листинг файла lab2\_1.c*

*1-ый вариант программы*

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <omp.h>

//----------Operations----------

// |x|=sqrt(sum(x\_i^2))

double modul(double\* x, int N)

{

double sum = 0.0;

#pragma omp parallel for reduction(+: sum)

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

{

sum += x[i] \* x[i];

}

return sqrt(sum);

}

// (x,y)=sum(x\_i\*y\_i)

double skobki(double\* x, double\* y, int N)

{

double res = 0.0;

#pragma omp parallel for reduction(+: res)

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

{

res += x[i] \* y[i];

}

return res;

}

//multiplicate matrix to vector: result = x \* y (N - size; x - matrix; result, y - vector)

void mul\_M\_v(double\* result, double\* x, double\* y, int N)

{

double\* res = (double\*)malloc(N \* sizeof(double));

#pragma omp parallel for

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

{

res[i] = 0.0;

}

#pragma omp parallel for

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

{

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

{

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

}

}

#pragma omp parallel for

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

{

result[i] = res[i];

}

free(res);

}

//multiplicate constant to vector: result = c \* y (N - size; c - constant; result, y - vector)

void mul\_c\_v(double\* result, double c, double\* y, int N)

{

#pragma omp parallel for

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

{

result[i] = c \* y[i];

}

}

//substraction vector to vector: result = x - y (N - size; result, x, y - vector)

void sub(double\* result, double\* x, double\* y, int N)

{

#pragma omp parallel for

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

{

result[i] = x[i] - y[i];

}

}

//----------Setters/Getters/Print----------

void random\_vector(double\* x, int N)

{

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

{

x[i] = (double)(rand() % 18 - 9);

}

}

void symmetrical\_matrix(double\* A, int N)

{

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

{

A[i \* N + i] = (double)(rand() % 18 - 9);

}

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

{

for (int j = 0; j < i; j++)

{

A[i \* N + j] = (double)(rand() % 18 - 9);

A[j \* N + i] = A[i \* N + j];

}

}

}

void print\_matrix(double\* A, int N)

{

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

{

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

{

printf("%2g ", A[i \* N + j]);

}

printf("\n");

}

}

void print\_vector(double\* V, int N)

{

printf("(");

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

{

printf("%g ", V[i]);

}

printf(")\n");

}

//----------Algorithm----------

void y\_pow(double\* y, double\* A, double\* x, double\* b, int N)

{

double\* Ax = (double\*)malloc(N \* sizeof(double));

mul\_M\_v(Ax, A, x, N);

sub(y, Ax, b, N);

free(Ax);

}

double t\_pow(double\* y, double\* A, int N)

{

double\* Ay = (double\*)malloc(N \* sizeof(double));

mul\_M\_v(Ay, A, y, N);

double t = skobki(y, Ay, N) / skobki(Ay, Ay, N);

free(Ay);

return t;

}

void x\_pow(double\* y, double\* A, double\* x, double\* b, int N, int i)

{

y\_pow(y, A, x, b, N);

double t = t\_pow(y, A, N);

double\* ty = (double\*)malloc(N \* sizeof(double));

mul\_c\_v(ty, t, y, N);

sub(x, x, ty, N);

free(ty);

}

int test(double\* y, double\* b, double epsilon, int N, int i)

{

double t = modul(y, N) / modul(b, N);

if (t < epsilon)

{

return 1; //остановка

}

else

{

return 0; //продолжай, пока не 1

}

}

int main()

{

//Входные данные

srand(278); //чтобы входные данные при каждом запуске были разные

int N = 1024;

double epsilon = 0.004;

double\* A = (double\*)malloc(N \* N \* sizeof(double));

double\* x = (double\*)malloc(N \* sizeof(double));

double\* b = (double\*)malloc(N \* sizeof(double));

double\* y = (double\*)malloc(N \* sizeof(double)); // y=Ax-b

printf("%d; ", omp\_get\_max\_threads());

//Создаём матрицу A и x, считаем b, а затем переписываем x (как раз далее будем этот вектор находить)

symmetrical\_matrix(A, N);

random\_vector(x, N);

mul\_M\_v(b, A, x, N);

random\_vector(x, N);

y\_pow(y, A, x, b, N);

//Фиксируем время начала

struct timespec start, finish;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

//Начало алгоритма

int i = 0; //счётчик итераций

while (test(y, b, epsilon, N, i) == 0)

{

x\_pow(y, A, x, b, N, i);

i++;

}

//Фиксируем время конца

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &finish);

printf("%d; ", i);

printf("%lf; \n", (finish.tv\_sec - start.tv\_sec + 0.000000001 \* (finish.tv\_nsec - start.tv\_nsec)));

free(A);

free(x);

free(b);

free(y);

return 0;

}

# Приложение 3

*Листинг файла lab2\_2.c*

*2-ой вариант программы*

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <omp.h>

//----------Setters/Getters/Print----------

void random\_vector(double\* x, int N)

{

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

{

x[i] = (double)(rand() % 18 - 9);

}

}

void symmetrical\_matrix(double\* A, int N)

{

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

{

A[i \* N + i] = (double)(rand() % 18 - 9);

}

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

{

for (int j = 0; j < i; j++)

{

A[i \* N + j] = (double)(rand() % 18 - 9);

A[j \* N + i] = A[i \* N + j];

}

}

}

void print\_matrix(double\* A, int N)

{

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

{

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

{

printf("%2g ", A[i \* N + j]);

}

printf("\n");

}

}

void print\_vector(double\* V, int N)

{

printf("(");

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

{

printf("%g ", V[i]);

}

printf(")\n");

}

int main()

{

//Входные данные

srand(278); //чтобы входные данные при каждом запуске были разные

int N = 1024;

double epsilon = 0.004;

double\* A = (double\*)malloc(N \* N \* sizeof(double));

double\* x = (double\*)malloc(N \* sizeof(double));

double\* b = (double\*)malloc(N \* sizeof(double));

printf("%d; ", omp\_get\_max\_threads());

//Создаём матрицу A и x, считаем b, а затем переписываем x (как раз далее будем этот вектор находить)

symmetrical\_matrix(A, N);

random\_vector(x, N);

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

{

b[i] = 0;

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

{

b[i] += A[i \* N + j] \* x[j];

}

}

random\_vector(x, N);

//Начало алгоритма

double sum1 = 0.0;

double sum2 = 0.0;

double sum3 = 0.0;

double sum4 = 0.0;

double t;

int check = 1;

double\* Ax = (double\*)malloc(N \* sizeof(double));

double\* Ay = (double\*)malloc(N \* sizeof(double));

double\* ty = (double\*)malloc(N \* sizeof(double));

double\* y = (double\*)malloc(N \* sizeof(double));

//Фиксируем время начала

struct timespec start, finish;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

int k = 1; //счётчик итераций

#pragma omp parallel

{

while (check == 1)

{

#pragma omp single

{

sum1 = 0.0;

sum2 = 0.0;

sum3 = 0.0;

sum4 = 0.0;

}

#pragma omp for reduction(+: sum1, sum2)

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

{

Ax[i] = 0;

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

{

Ax[i] += A[i \* N + j] \* x[j];

}

y[i] = Ax[i] - b[i];

sum1 += y[i] \* y[i];

sum2 += b[i] \* b[i];

}

#pragma omp single

{

if ((sqrt(sum1) / sqrt(sum2)) < epsilon)

{

check = 0;

}

}

if (check == 1)

{

#pragma omp for reduction(+: sum3, sum4)

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

{

Ay[i] = 0;

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

{

Ay[i] += A[i \* N + j] \* y[j];

}

sum3 += y[i] \* Ay[i];

sum4 += Ay[i] \* Ay[i];

}

#pragma omp single

{

t = sum3 / sum4;

}

#pragma omp for

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

{

ty[i] = t \* y[i];

x[i] = x[i] - ty[i];

}

#pragma omp single

{

k++;

}

}

}

}

//Фиксируем время конца

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &finish);

printf("%d; ", k);

printf("%lf; \n", (finish.tv\_sec - start.tv\_sec + 0.000000001 \* (finish.tv\_nsec - start.tv\_nsec)));

free(A);

free(x);

free(b);

free(Ax);

free(Ay);

free(ty);

free(y);

return 0;