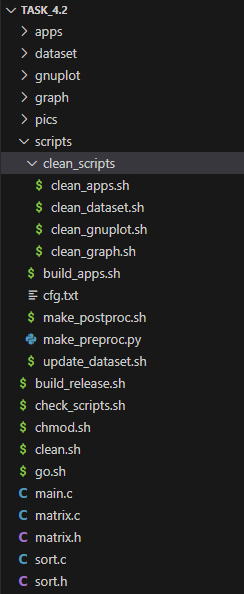
**Смирнов Иван ИУ7-22Б - 2023г.**

**Отчет**

**Задание №4.2**

**Задание**

В данном задании необходимо проанализировать время выполнения двух реализаций сортировки строк матрицы (по возрастанию суммы элементов строки), а именно: сортировка с кешированием суммы элементов строки, сортировка без кеширования. Программа собирается с несколькими типами оптимизаций (Os, O0, O3). Вид рабочей папки выглядит следующим образом:



(рис.1 – файловая структура задания №2)

Основной алгоритм сортировки – сортировка выбором. Ниже приведены 2 реализации.

1. С кешированием суммы

// Реализация 1

// Функция находит минимум в диапазоне от strat\_ind до end\_ind (с кешированием суммы)

size\_t find\_min\_elem1(int a[][NMAX], size\_t start\_ind, size\_t end\_ind)

{

    size\_t min\_elem = start\_ind;

    int min\_sum = find\_sum(a[start\_ind], end\_ind);

    for (size\_t j = start\_ind; j < end\_ind; j++)

    {

        int cur\_sum = find\_sum(a[j], end\_ind);

        if (cur\_sum < min\_sum)

        {

            min\_sum = cur\_sum;

            min\_elem = j;

        }

    }

    return min\_elem;

}

// Функция сортировки выбором

void selection\_sort1(int a[][NMAX], size\_t a\_size)

{

    size\_t min\_elem = 0;

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

    {

        min\_elem = find\_min\_elem1(a, i, a\_size);

        swap\_arrays(a[i], a[min\_elem], a\_size);

    }

}

// конец

1. Без кеширования суммы

// Реализация 2

// Функция находит минимум в диапазоне от strat\_ind до end\_ind (без кеширования суммы)

size\_t find\_min\_elem2(int a[][NMAX], size\_t start\_ind, size\_t end\_ind)

{

    size\_t min\_elem = start\_ind;

    for (size\_t j = start\_ind; j < end\_ind; j++)

    {

        if (find\_sum(a[j], end\_ind) < find\_sum(a[min\_elem], end\_ind))

        {

            min\_elem = j;

        }

    }

    return min\_elem;

}

// Функция сортировки выбором

void selection\_sort2(int a[][NMAX], size\_t a\_size)

{

    size\_t min\_elem = 0;

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

    {

        min\_elem = find\_min\_elem2(a, i, a\_size);

        swap\_arrays(a[i], a[min\_elem], a\_size);

    }

}

// конец

В обоих реализациях используются следующие вспомогательные функции:

// Вспомогательные функции для обоих реализаций сортировок

// Функция находит сумму элементов массива

int find\_sum(const int \*a, size\_t a\_size)

{

    int sum = 0;

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

        sum += a[i];

    return sum;

}

// Функция меняет местами массивы (по-элементно)

void swap\_arrays(int a[], int b[], size\_t n)

{

    for (size\_t i = 0; i < n; i++)

    {

        int tmp = a[i];

        a[i] = b[i];

        b[i] = tmp;

    }

}

Все реализации находятся в файле sort.c. Описание каждой из реализаций можно прочитать в заголовочном файле sort.h.

Также были реализованы функции генерации матрицы случайных чисел (init). Описание функций находится в заголовочном файле matrix.h; сами функции в matrix.c.

В данной задаче мы будем рассматривать только квадратную матрицу, так как проводить тестовые измерения будет проще именно с квадратной матрицей. Так же с помощью квадратной матрицы можно получить более наглядный график, то есть сравнить время выполнения программы для матрицы 50х50 с матрицей 60х60 будет проще, чем время для матрицы 50х80 с матрицей 60х70. Проще ориентироваться по изменению одной составляющей, чем по двум.

Листинг init из matrix.h:

void init(int a[][NMAX])

{

srand(time(NULL));

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

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

a[i][j] = (rand() % (NMAX \* 3));

}

Листинг основной программы (***main.c***):

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <sys/time.h>

#include "matrix.h"

#include "sort.h"

#define OK 0

#define ERR\_IO 1

#ifndef NMAX

#error NMAX IS NOT DEFINED

#endif

#ifndef SORT\_TYPE

#error TYPE IS NOT DEFINED

#endif

struct timespec

{

time\_t tv\_sec;

long tv\_nsec;

};

int nanosleep(const struct timespec \*req, struct timespec \*rem);

// Замерительный метод - gettimeofday

int main(int argc, char \*\*argv)

{

if (argc != 2)

return ERR\_IO;

int a[NMAX][NMAX];

int sort\_type = SORT\_TYPE;

int size = atoi(argv[1]);

init(a);

struct timeval current\_time;

unsigned long long beg, end;

// Замер начала

gettimeofday(&current\_time, NULL);

beg = current\_time.tv\_sec \* 1000000ULL + current\_time.tv\_usec;

// Действия

switch(sort\_type)

{

case 1:

selection\_sort1(a, size);

break;

case 2:

selection\_sort2(a, size);

break;

default:

return ERR\_IO;

}

// Замер конца

gettimeofday(&current\_time, NULL);

end = current\_time.tv\_sec \* 1000000ULL + current\_time.tv\_usec;

printf("%llu\n", (end - beg));

return OK;

}

Программу необходимо запустить с одним аргументом, который отвечает за размерность матрицы (в программе это переменная size). Такие переменные как максимальный размер статического массива (NMAX), тип сортировки (SORT\_TYPE), тип массива (ARR\_TYPE) определяются на этапе сборки. Программа выводит на экран время работы выбранной сортировки в микросекундах (1e-6c).

Чтобы проанализировать время для разного размера массива, разных типов оптимизации и разных реализаций сортировок, была написана целая анализирующая система. Необходимо запустить скрипт ***go.sh***.

#!/bin/bash

./scripts/build\_apps.sh

./scripts/update\_dataset.sh

./scripts/clean\_scripts/clean\_graph.sh

python3 ./scripts/make\_preproc.py

./scripts/make\_postproc.sh

Проанализируем работу скрипта. **(1)** Сначала из папки scripts запускается скрипт ***build\_apps.sh***, который собирает все возможные версии программы (с типами оптимизации и двумя типами сортировок, то есть собирается в сумме 3\*2=6 различных исполняемых файлов).

build\_apps.sh:

#!/bin/bash

cd ./scripts || exit

./clean\_scripts/clean\_apps.sh

i=0

while read -r line; do

    if [ $i -eq 0 ]; then

        opt=$line

    fi

    if [ $i -eq 1 ]; then

        sort\_types=$line

    fi

    if [ $i -eq 2 ]; then

        nmax=$line

    fi

    if [ $i -eq 3 ]; then

        tests=$line

    fi

    i=$((i + 1))

done < ./cfg.txt

# opt - Os; O0; O2

# sort\_type: 1 - с кешированием суммы; 2 - без кеширования

cd ..

for option in $opt; do

    for sort\_type in $sort\_types; do

        ./build\_release.sh "$option" "$sort\_type" "$nmax"

    done

done

Вся информация о сборке программы хранятся в файле ***./scripts/cfg.txt***

***cfg.txt:***

Os O0 O3

1 2

100

20

1 2 5 10 15 20 25 35 40 50 60 65 75 85 90 100

В первой строке написаны типы оптимизации (Os, О0, О2). Во второй строке расположены разные реализации сортировки (описаны ранее). Во четвертой строке написано количество проводимых измерений (тестов), то есть сколько раз запустится каждый из 6 исполняемых файлов. В третьей строке написано максимальное количество строк/столбцов в матрице, по-другому, - размерность матрицы (по условию задачи оно равно 100). В пятой строке написаны все тестируемые размерности матрицы.  
Далее все 6 исполняемых файлов собираются с помощью скрипта ***build\_release.sh:***

#!/bin/bash

gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla -"${1}" -DSORT\_TYPE="${2}" -DNMAX="${3}" -c main.c

gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla -DNMAX="${3}" -c matrix.c

gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla -DNMAX="${3}" -c sort.c

gcc -o ./apps/app\_"$1"\_s"$2"\_n"$3".exe main.o matrix.o sort.o -lm

Все исполняемые файлы помещаются в папку **apps**

**(2)** Далее запускается скрипт ***./scripts/update\_data.sh***, который запускает tests раз (в данном случае 20) на разных размерностях матрицы каждый из исполняемых файлов. Результаты (время в мкс) помещаются в текстовые файлы в папку dataset.

#!/bin/bash

cd ./scripts || exit

./clean\_scripts/clean\_dataset.sh

i=0

while read -r line; do

    if [ $i -eq 0 ]; then

        opt=$line

    fi

    if [ $i -eq 1 ]; then

        sort\_types=$line

    fi

    if [ $i -eq 2 ]; then

        nmax=$line

    fi

    if [ $i -eq 3 ]; then

        tests=$line

    fi

    if [ $i -eq 4 ]; then

        sizes=$line

    fi

    i=$((i + 1))

done < ./cfg.txt

cd ..

for option in $opt; do

    for sort\_type in $sort\_types; do

        echo "app\_'${option}'\_s'${sort\_type}'\_n'${nmax}'.exe - done"

        for size in $sizes; do

            for (( i = 0; i < "$tests"; i++ )); do

                cmd="$(./apps/app\_"${option}"\_s"${sort\_type}"\_n"${nmax}".exe "${size}")"

                # echo ../applications/main"${option}".exe "${type}" "${size}" "${type\_of\_sort}"

                echo "$cmd" >> ./dataset/"${option}"\_s"${sort\_type}"\_n"${size}".txt

            done

        done

    done

done

**(3)** Далее запускается скрипт ***./scripts/make\_preproc.py,*** который на основе данных из **dataset** формирует новые данные, необходимые для дальнейшего анализа (среднее, минимум и максимум, медианное значение, верхний и нижний квартили). Все полученные данные скрипт сохраняет в текстовых файлах в папку ***graph.*** Перед запуском скрипта удаляется старый “graph” (если существовал) с помощью вспомогательного скрипта **clean\_graph.sh.**

# Считываем данные из конфига

with open("./scripts/cfg.txt", "r") as f:

    line = f.readline()

    i = 0

    while (len(line) != 0):

        line = line.strip()

        if (i == 0):

            options = list(line.split())

        elif (i == 1):

            sort\_types = list(line.split())

        elif (i == 3):

            tests = int(line)

        elif (i == 4):

            sizes = list(line.split())

        line = f.readline()

        i += 1

# Обрабатываем данные

def analyze(time):

    # time.sort()

    avg = sum(time) // len(time)

    minimum = min(time)

    low\_quartile = time[len(time)//4]

    median = time[len(time)//2]

    high\_quartile = time[len(time)//4\*3]

    maximum = max(time)

    return str(avg), str(minimum), str(low\_quartile), str(median), str(high\_quartile), str(maximum)

# Считываем и обрабатываем данные из датасета

for opt in options:

    for sort\_type in sort\_types:

        for size in sizes:

            time = [0]\*tests

            with open(f"./dataset/{opt}\_s{sort\_type}\_n{size}.txt", "r") as fin:

                for i in range(tests):

                    time[i] = int(fin.readline().strip())

            time.sort()

            data = list(analyze(time))

            with open(f"./graph/{opt}\_s{sort\_type}.txt", 'a') as fout:

                for t in time:

                    fout.write(str(str(t) + " " + " ".join(data) + " " + str(size) + "\n"))

Данные из graph выглядят примерно так:

. . .  
108 137 105 105 106 107 712 40

139 137 105 105 106 107 712 40

712 137 105 105 106 107 712 40

134 267 134 219 233 316 503 50

191 267 134 219 233 316 503 50

. . .

1-ый столбец – время из dataset

2-ой столбец – полученное среднее

3-ий столбец – минимум

4-ый столбец – нижний квартиль

5-ый столбец – медианное значение

6-ой столбец – верхний квартиль

7-ой столбец – максимум

8-ой столбец – размер массива

На основе этих данных можно построить графики зависимости времени сортировки от размерности матрицы.

**(4)** Запускается скрипт ***./scripts/make\_outproc.sh,*** который строит графики и сохраняет их в образцах (файлах .gpi) в папке gnuplot.

#!/bin/bash

TXT="./graph/\*.txt"

PREFIX="./gnuplot/gnuplot\_"

./scripts/clean\_scripts/clean\_gnuplot.sh

# Общий кусочно-линейный график

for file in $TXT; do

    basename=$(basename "${file}")

    name=$PREFIX$basename".gpi"

    {

        echo "set output \"./gnuplot/${basename}.svg\""

        cat "./gnuplot/tmpl.txt"

        echo "set terminal svg size 1080, 720"

        echo "plot \"$file\" using 8:2 with lines title \"$basename\""

    } >> "$name"

done

# Графики из задания

cd ./scripts || exit

./clean\_scripts/clean\_dataset.sh

i=0

while read -r line; do

    if [ $i -eq 0 ]; then

        opt=$line

    fi

    if [ $i -eq 2 ]; then

        sort\_types=$line

    fi

    if [ $i -eq 4 ]; then

        arr\_types="${line}"

    fi

    i=$((i + 1))

done < ./cfg.txt

cd ..

# ЗАПУСТИТЬ ГРАФИК

#gnuplot $PATH$NAME".gpi" -persist

В конце скрипта написана команда, с помощью которой можно увидеть отображения графика в программе gnuplot (*gnuplot $PATH$NAME".gpi" -persist*).   
Графики строятся следующим образом: по оси х – количество элементов (столбец 8) в массиве, по оси у – среднее замеренное время для данного количества элементов (столбец 2).

Графики расположены в папке pics.

Как видно из графиков время сортировки с кешированием суммы занимает во всех случаях оптимизации меньше времени, чем без кеширования, так как в первой реализации на каждой итерации внутреннего цикла функция суммы вызывается только один раз, а не два, как во второй реализации. И чем больше уровень оптимизации, тем более ощутима разница во времени. Так при уровне оптимизации O0 разница во времени находится в пределах 1000мкс, для Os, который использует параметры O2, разница во времени находится в пределах 3300мкс! А для O3, несмотря на выхлопы на графиках O3\_s1; O3\_s2, все равно разница находится в пределах 2000мкс.

Если же сравнивать графики по уровню оптимизации, то самым стабильным из всех будет уровень O0, то есть без оптимизации. Так как эксперимент проводится на маленькой программе с маленьким кэшем, то заметить увеличение производительности от оптимизации O3 практически невозможно.

Листинг вспомогательных скриптов:

**check\_scripts.sh** (проверяет все скрипты при помощи shellcheck):

#!/bin/bash

scripts="./\*.sh"

for file in $scripts ; do

shellcheck "$file"

done

scripts="./scripts/\*.sh"

for file in $scripts ; do

shellcheck "$file"

done

**chmod.sh** (дает всем скриптам право на выполнение):

#!/bin/bash

files="./scripts/\*.sh ./scripts/clean\_scripts/\*.sh ./\*.sh"

for file\_in in $files; do

chmod +x "$file\_in"

done

**clean.sh** (очищает мусор в рабочей папке (не включая внутренние папки)):

#!/bin/bash

files="\*.exe \*.o \*.out \*.gcno \*.gcda \*.gcov"

for file in $files; do

rm -f "$file"

done

**clean\_apps.sh** (очищает мусор в папке apps)):

#!/bin/bash

files="./apps/\*.exe"

for file in $files; do

rm -f "$file"

done

**clean\_dataset.sh** (очищает dataset)):

#!/bin/bash

files="./dataset/\*.txt"

for file in $files; do

rm -f "$file"

done

**clean\_gnuplot.sh** (очищает папку gnuplot)):

#!/bin/bash

files="./gnuplot/\*.gpi ./gnuplot/\*.svg"

for file in $files; do

rm -f "$file"

done

**clean\_graph.sh** (очищает папку graph)):

#!/bin/bash

files="./graph/\*.txt"

for file in $files; do

rm -f "$file"

done

Таблица для O3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер N |  |  |  |  |
| 1 | - |  | - | - |
| 2 | - |  | - | - |
| 5 | - | 54,77225575 | - | -1,64047815 |
| 10 | 20,41241452 | 17,56820922 | -2,62543967 | 4,26104858 |
| 15 | 7,04007686 | 98,86917388 | -1,34011617 | -9,51938714 |
| 20 | 4,78789897 | 6,39323550 | -2,49797553 | -6,51714784 |
| 25 | 2,74199170 | 1,49328884 | 7,33759202 | 3,00747977 |
| 35 | 32,38128255 | 4,10791012 | -8,15325032 | -2,57163829 |
| 40 | 10,90112163 | 2,91398036 | -5,61388986 | 5,95481373 |
| 50 | 3,11479196 | 11,00443369 | -0,94249404 | -15,13554093 |
| 60 | 2,62301753 | 0,69681374 | -3,68845190 | 29,10028268 |
| 65 | 1,95246310 | 7,15655191 | 14,61838896 | -8,39459788 |
| 75 | 15,81557951 | 2,15276755 | -29,47947757 | -2,44478738 |
| 85 | 0,39504777 | 1,58527354 | 47,93400361 | 5,98462043 |
| 90 | 6,11706892 | 2,23184453 | -9,83941627 | -8,47632564 |
| 100 | 2,16928381 | 0,91371035 | - | - |