МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №4

по курсу «Эффективное программирование современных микропроцессоров и мультипроцессоров»

(Вариант №3)

Выполнил: студент 3-го курса гр. 17208

Гафиятуллин А.Р

Новосибирск, 2020

1. **ЦЕЛИ РАБОТЫ:**

   Научиться распараллеливать в потоках простые программы численного моделирования.

   Вариант №3: *решение уравнения Пуассона методом Якоби на float-ах.* Алгоритм моделирует установление стационарного распределения тепла в пластинке с заданным распределением источников и стоков тепла. В начальный момент времени значения искомой функции на сетке инициализируются нулями. На каждом шаге моделирования значения искомой функции пересчитываются по заданной формуле.

2. **ХОД РАБОТЫ:**

   2.1.Параметры тестирования:

   2.1.1. Тестирование происходило на процессоре **Intel(R) Core(TM) i7-9700F CPU @ 3.00GHz (CPU max MHz: 4700.0000 (Turbo Boost))**. **8 ядер, 8 потоков, гипертредининга нет.**

   2.1.2. Компилятор**: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 19.1.1.216 Build 20200306;**

   2.1.3. Ключи компиляции (наиболее важные): **/Qopenmp /O2 /tune:coffeelake /arch:CORE-AVX2;**

   2.1.4. Параметры программы: $\mathbf{N_x = N_y = 9000, N_t = 110}$.

   2.2.Листинг программы из пункта 1(см. Приложение A);

   2.3.Листинг программы из пункта 2(см. Приложение B);

   2.4.Листинг программы из пункта 3(см. Приложение C);

   2.5.Все программы при любом количестве потоков выдают **Delta = 1.18546e-08.**

   2.6. Итоговая таблица с временами выполнения:

Потоки, кол-во	**Вариант 1, сек.**	**Вариант 2, сек.**	**Вариант 3, сек.**
1	6.988	5.425	5.41
2	4.844	2.87	2.855
3	4.516	2.082	2.058
4	4.577	1.721	1.727
5	4.669	1.568	1.573
6	4.614	1.579	1.58

7	4.61	1.569	1.656
8	4.779	1.682	1.736

Победителем по наилучшему показателю является вариант 2, однако вариант 3 практически никак не отличается по времени работы от варианта 2.

1 вариант ожидаемо хуже при любом количестве потоков, так как в нем не применялась оптимизация по памяти из 3 задания.

Все программы к 4-5 потоку достигают своей пиковой скорости работы, далее прироста либо нет, либо начинается только ухудшение производительности.

2.7. Производительность 1 и 3 программ на 8 потоках и сравнение с программой из задания 3:

2.7.1.1. Число инструкций на такт:

1 программа: 0.25438819638
3 программа: 0.76863950807
Из задания 3: 1.51
У программ 1 и 3 довольно низкие показатели по сравнению с программой из задания 3. Похоже, что ядра процессора большую часть времени простаивают без дела. Причем, однопоточная версия лучше 8-поточной 1-й программы примерно в 6 раз, что примерно равно количеству потоков.

2.7.1.2. Процент кэш-промахов для кэша 3 уровня:

1 программа: 73%.
3 программа: 22%.
Из задания 3: 19.76%
Из-за отсутствия оптимизации по памяти 1 программа практически всегда промахивается при доступе к LLC. Этим можно объяснять и низкий IPC, процессор долго ждет данные. Обратная ситуация у 3 программы и программы из здания 3, у которых схожие показатели.

2.7.1.3. Процент кэш-промахов для кэша 1 уровня:

1 программа: 3%.
3 программа: 2%.
Из задания 3: 1.96%

Примерно одинаковые результаты.

2.7.1.4. Процент неправильно предсказанных переходов:

1 программа: 1.08%.

3 программа: 0.99%.

Из задания 3: 1.68%

Примерно одинаковые результаты.

2.7.1.5. Время работы:

1 программа: 4.779 сек.

3 программа: 1.736 сек.

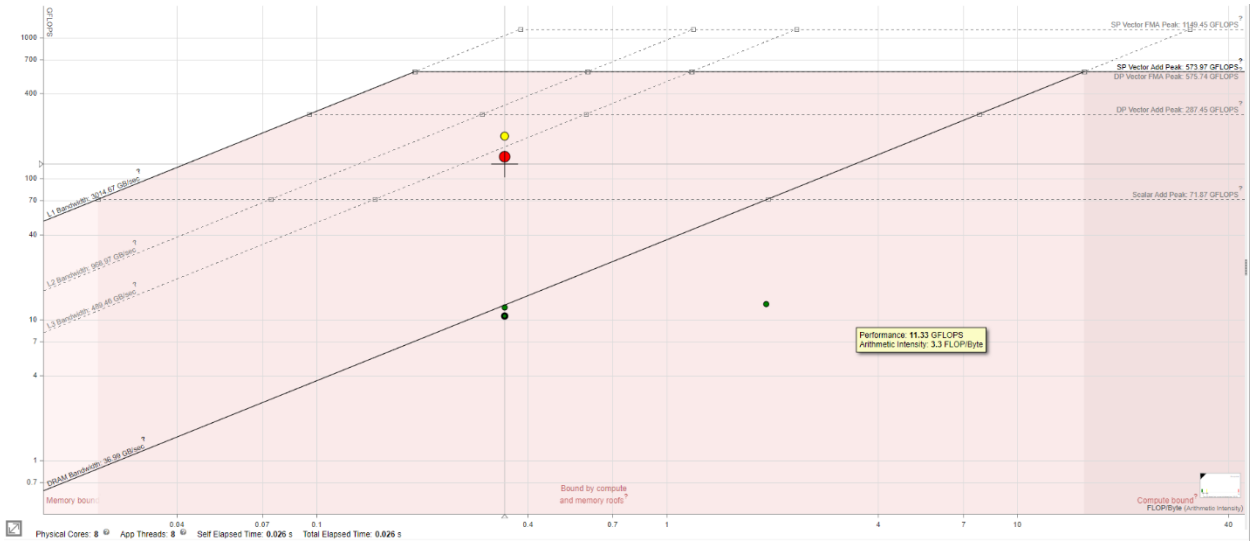
Из задания 3: 5.082 сек.

1 программа медленнее 3 в 2.75 раза. В примерно такое же количество раз (около 3) у этой программы больше промахов при доступе к LLC и хуже IPC.

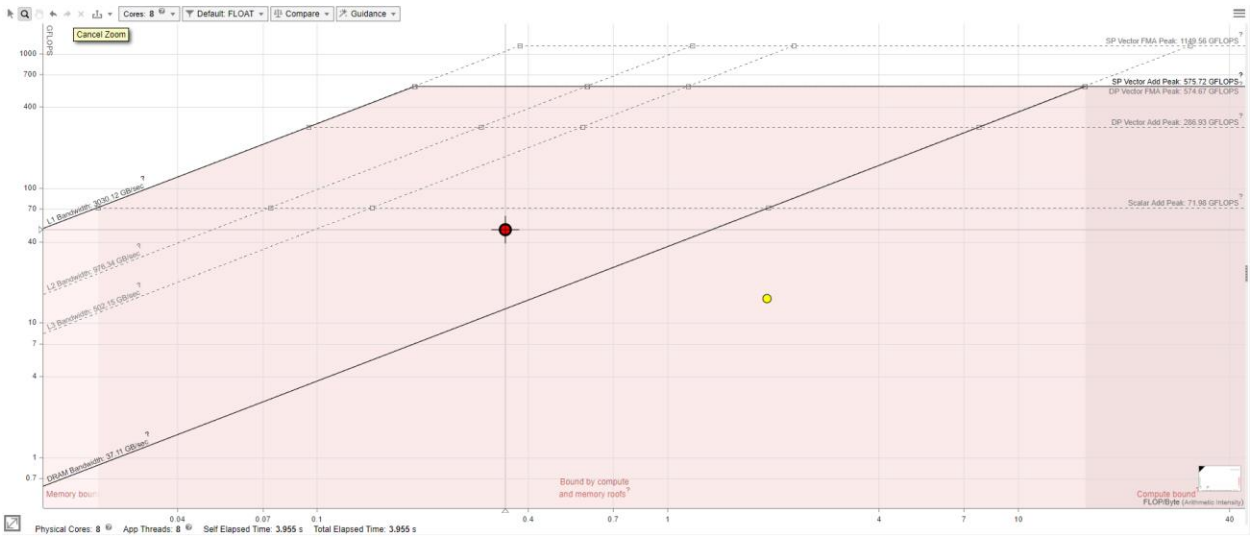
Программа из здания 3 очень близка по времени работы к программе 1. Из соотношения их IPC можно сказать, что инструкции в достаточной мере вместились бы и на одном ядре, но в многопоточной 1 программе они были просто размазаны по 8 ядрам, которые все равно простаивали в ожидании данных для инструкций.

2.8. Roofline-модель с точками, соответствующими основному циклу программы для 8 поточных программ 1 и 3:

Для программы 3:



Для программы 1:



Точки основного цикла находятся на одном и том же уровне по оси пропускной способности памяти, но по оси производительности процессора в разных. В купе с тем, что в прошлом пункте было выяснено влияние кэш-промахов на IPC и его соотношения у разных программ, можно сделать вывод, что дальнейшему увеличению ускорения при распараллеливании препятствует неспособность памяти успевать обеспечивать процессор данными для исполнения инструкций.

3. ВЫВОДЫ:

3.1. Научились распараллеливать в потоках простые программы численного моделирования;

3.2. Память все еще является бутылочным горлышком производительности программ.

4. ПРИЛОЖЕНИЕ А. Программа 1

```
constexpr auto ALIGN = 32;

constexpr auto VECTOR_SIZE = 8;
constexpr auto SHIFT1 = 1;

constexpr auto Nx = 9000;
constexpr auto Ny = 9000;
constexpr auto Nt = 111;

constexpr float Xa = 0.0f;
constexpr float Xb = 4.0f;
constexpr float Ya = 0.0f;
constexpr float Yb = 4.0f;
```

```
constexpr float hx = ((Xb - Xa) / (Nx - 1));
constexpr float hy = ((Yb - Ya) / (Ny - 1));
constexpr float coeff1 = (0.2f / ((1.0f / (hx * hx) + 1.0f / (hy * hy))));
constexpr float coeff2 = (0.5f * (5.0f / (hx * hx) - 1.0f / (hy * hy)));
constexpr float coeff2b = (0.5f * (5.0f / (hy * hy) - 1.0f / (hx * hx)));
constexpr float coeff3 = (0.25f * (1.0f / (hx * hx) + 1.0f / (hy * hy)));

inline float X(size_t j) { return (Xa + (j)*hx); }
inline float Y(size_t i) { return (Ya + (i)*hy); }

constexpr float Xs1 = (Xa + (Xb - Xa) / 3.0f);
constexpr float Xs2 = (Xa + (Xb - Xa) * 2.0f / 3.0f);
constexpr float Ys1 = (Ya + (Yb - Ya) * 2.0f / 3.0f);
constexpr float Ys2 = (Ya + (Yb - Ya) / 3.0f);
constexpr float R = (0.1f * ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)));

constexpr float GRID_SIZE = (Nx * Ny);
constexpr int THREADS = 8;
constexpr int CHUNK = Ny / THREADS;

inline float& get(float* p, size_t i, size_t j) { return p[(i)*Nx + (j)]; }

#include <iostream>
#include <fstream>
#include <cerrno>
#include <ctime>
#include <algorithm>
#include <immintrin.h>
using namespace std;

/* prepare for vectorization */
__m256 v_coeff1;
__m256 v_coeff2;
__m256 v_coeff2b;
__m256 v_coeff3;
__m256 v_delta;
__m256 global_v_delta;
__m256 v_coeff4;
__m256 v_coeff5;

inline void compute_process(float* F0, float* F1, float* p, int i) {
    for(int j = 1; j < Nx / VECTOR_SIZE - 1; j++) {
        __m256 rez = _mm256_mul_ps(v_coeff1, (
```

```
                        _mm256_fmadd_ps(v_coeff3, _mm256_add_ps(((__m256*)(F0 + (
i - 1) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i - 1)*Nx + SHIFT1))[j]),
                        _mm256_fmadd_ps(v_coeff2b, _mm256_add_ps(((__m256*)(F0 +
(i - 1) * Nx))[j], ((__m256*)(F0 + (i + 1) * Nx))[j]),
                        _mm256_fmadd_ps(v_coeff2, _mm256_add_ps(((__m256*)(F0 + (
i) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i) * Nx + SHIFT1))[j]),
                        _mm256_fmadd_ps(v_coeff3, _mm256_add_ps(((__m256*)(F0 + (
i + 1) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i + 1) * Nx + SHIFT1))[j]),

                        _mm256_fmadd_ps(v_coeff5, ((__m256*)(p + (i) * Nx))[j],
                            _mm256_mul_ps(v_coeff4, (
                                _mm256_add_ps(
                                    _mm256_add_ps(
                                        _mm256_add_ps(((__m256*)(p + (i -
 1) * Nx))[j], ((__m256*)(p + (i) * Nx - SHIFT1))[j]),
                                                ((__m256*)(p + (i) * Nx + SHIFT1))[j]
),
                                                    ((__m256*)(p + (i + 1) * Nx))[j])
                                    )
                                )
                            )
                        )
                        )
                        )
                        )
                    )
                );

        __m256 prev_rez = ((__m256*)(F1 + (i) * Nx))[j];
        v_delta = _mm256_max_ps(v_delta, _mm256_max_ps(_mm256_sub_ps(prev_rez, re
z), _mm256_sub_ps(rez, prev_rez)));
        ((__m256*)(F1 + (i) * Nx))[j] = rez;
    }
}

int main() {
    /* allocate memory */
    float *F0 = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *F1 = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *p = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    if(!F0 || !F1 || !p) {
        perror("_mm_malloc");
        exit(errno);
    }
```

```
    /* prepare for vectorization */
    v_coeff1 = _mm256_set1_ps(coeff1);
    v_coeff2 = _mm256_set1_ps(coeff2);
    v_coeff2b = _mm256_set1_ps(coeff2b);
    v_coeff3 = _mm256_set1_ps(coeff3);
    v_delta = _mm256_setzero_ps();
    v_coeff4 = _mm256_set1_ps(0.25f);
    v_coeff5 = _mm256_set1_ps(2.0f);

    clock_t start, end;

#pragma omp parallel proc_bind(close) num_threads(THREADS) private(v_delta) // OM
P_PLACES is "cores" by default
    {
        /* init arrays */
#pragma omp for schedule(static, CHUNK)
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                float xj = X(j);
                float yi = Y(i);
                if ((xj - Xs1) * (xj - Xs1) + (yi - Ys1) * (yi - Ys1) < R * R) {
                    get(p, i, j) = 0.1f;
                }
                else if ((xj - Xs2) * (xj - Xs2) + (yi - Ys2) * (yi -
 Ys2) < R * R) {
                    get(p, i, j) = -0.1f;
                }
                else {
                    get(p, i, j) = 0.0f;
                }
                get(F0, i, j) = 0.0f;
            }
        }
#pragma omp barrier

        /* compute process */
#pragma omp single
        start = clock();

        for (int n = 0; n < Nt - 1; n++) {
#pragma omp single
            global_v_delta = _mm256_setzero_ps();

            v_delta = _mm256_setzero_ps();
```

```
#pragma omp for schedule(static, CHUNK)
            for (int i = 2; i < Ny - 3; i++) {
                compute_process(F0, F1, p, i);
            }
#pragma omp critical
            {
                global_v_delta = _mm256_max_ps(v_delta, global_v_delta);
            }
#pragma omp single
            swap(F0, F1);
#pragma omp barrier
        }

#pragma omp single
        end = clock();
    }
    /* print max delta */
    float max_delta = 0.0f;
    float* vec_delta = (float*)(&global_v_delta);
    for (int i = 0; i < VECTOR_SIZE; i++) {
        max_delta = max(max_delta, vec_delta[i]);
    }
    cout << "n = " << Nt - 1 << " sigma = " << max_delta << endl;
    cout << "Total time: " << (double)(end -
 start) / CLOCKS_PER_SEC << " sec." << endl;

    cout << "Generating plot..." << endl;
    auto plot_file = fstream("computation.plot", ios::binary | ios::trunc | ios::
out);
    if (!plot_file.good()) {
        cerr << "Error while openning file!" << endl;
        return -1;
    }
    plot_file.write((char*)F0, GRID_SIZE * sizeof(float));
    plot_file.close();

    system("wsl gnuplot plot_script");  // call Linux to create plot

    _mm_free(F0);
    _mm_free(F1);
    _mm_free(p);

    return 0;
}
```

ПРИЛОЖЕНИЕ В. Программа 2

```
constexpr auto ALIGN = 32;

constexpr auto VECTOR_SIZE = 8;
constexpr auto SHIFT1 = 1;

constexpr auto Nx = 9000;
constexpr auto Ny = 9000;
constexpr auto Nt = 111;

constexpr float Xa = 0.0f;
constexpr float Xb = 4.0f;
constexpr float Ya = 0.0f;
constexpr float Yb = 4.0f;

constexpr float hx = ((Xb - Xa) / (Nx - 1));
constexpr float hy = ((Yb - Ya) / (Ny - 1));
constexpr float coeff1 = (0.2f / ((1.0f / (hx * hx) + 1.0f / (hy * hy))));
constexpr float coeff2 = (0.5f * (5.0f / (hx * hx) - 1.0f / (hy * hy)));
constexpr float coeff2b = (0.5f * (5.0f / (hy * hy) - 1.0f / (hx * hx)));
constexpr float coeff3 = (0.25f * (1.0f / (hx * hx) + 1.0f / (hy * hy)));

inline float X(size_t j) { return (Xa + (j)*hx); }
inline float Y(size_t i) { return (Ya + (i)*hy); }

constexpr float Xs1 = (Xa + (Xb - Xa) / 3.0f);
constexpr float Xs2 = (Xa + (Xb - Xa) * 2.0f / 3.0f);
constexpr float Ys1 = (Ya + (Yb - Ya) * 2.0f / 3.0f);
constexpr float Ys2 = (Ya + (Yb - Ya) / 3.0f);
constexpr float R = (0.1f * ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)));

constexpr float GRID_SIZE = (Nx * Ny);
constexpr int THREADS = 8;
constexpr int CHUNK = Ny / THREADS;

inline float& get(float* p, size_t i, size_t j) { return p[(i)*Nx + (j)]; }

#include <iostream>
#include <fstream>
#include <cerrno>
#include <ctime>
#include <algorithm>
#include <immintrin.h>
#include <omp.h>
using namespace std;
```

```
/* prepare for vectorization */
__m256 v_coeff1;
__m256 v_coeff2;
__m256 v_coeff2b;
__m256 v_coeff3;
__m256 v_delta;
__m256 global_v_delta;
__m256 v_coeff4;
__m256 v_coeff5;

inline void compute_process(float* F0, float* F1, float* p, int i) {
    for(int j = 1; j < Nx / VECTOR_SIZE - 1; j++) {
        __m256 rez = _mm256_mul_ps(v_coeff1, (
                        _mm256_fmadd_ps(v_coeff3, _mm256_add_ps(((__m256*)(F0 + (
i - 1) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i - 1) * Nx + SHIFT1))[j]),
                        _mm256_fmadd_ps(v_coeff2b, _mm256_add_ps(((__m256*)(F0 +
(i - 1) * Nx))[j], ((__m256*)(F0 + (i + 1) * Nx))[j]),
                        _mm256_fmadd_ps(v_coeff2, _mm256_add_ps(((__m256*)(F0 + (
i) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i) * Nx + SHIFT1))[j]),
                        _mm256_fmadd_ps(v_coeff3, _mm256_add_ps(((__m256*)(F0 + (
i + 1) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i + 1) * Nx + SHIFT1))[j]),

                        _mm256_fmadd_ps(v_coeff5, ((__m256*)(p + (i) * Nx))[j],
                            _mm256_mul_ps(v_coeff4, (
                                _mm256_add_ps(
                                    _mm256_add_ps(
                                        _mm256_add_ps(((__m256*)(p + (i -
 1)*Nx))[j], ((__m256*)(p + (i) * Nx - SHIFT1))[j]),
                                                ((__m256*)(p + (i) * Nx + SHIFT1))[j]
),
                                                    ((__m256*)(p + (i + 1) * Nx))[j])
                                    )
                                )
                            )
                        )
                        )
                        )
                        )
                    )
                );

        __m256 prev_rez = ((__m256*)(F1 + (i) * Nx))[j];
        v_delta = _mm256_max_ps(v_delta, _mm256_max_ps(_mm256_sub_ps(prev_rez, re
z), _mm256_sub_ps(rez, prev_rez)));
```

```
        ((__m256*)(F1 + (i) * Nx))[j] = rez;
    }
}

int main() {
    /* allocate memory */
    float *F0 = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *F1 = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *p = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    if(!F0 || !F1 || !p) {
        perror("_mm_malloc");
        exit(errno);
    }

    /* prepare for vectorization */
    v_coeff1 = _mm256_set1_ps(coeff1);
    v_coeff2 = _mm256_set1_ps(coeff2);
    v_coeff2b = _mm256_set1_ps(coeff2b);
    v_coeff3 = _mm256_set1_ps(coeff3);
    v_delta = _mm256_setzero_ps();
    v_coeff4 = _mm256_set1_ps(0.25f);
    v_coeff5 = _mm256_set1_ps(2.0f);

    clock_t start, end;

#pragma omp parallel proc_bind(close) num_threads(THREADS) private(v_delta) // OM
P_PLACES is "cores" by default
    {
        /* init arrays */
#pragma omp for schedule(static, CHUNK)
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                float xj = X(j);
                float yi = Y(i);
                if ((xj - Xs1) * (xj - Xs1) + (yi - Ys1) * (yi - Ys1) < R * R) {
                    get(p, i, j) = 0.1f;
                }
                else if ((xj - Xs2) * (xj - Xs2) + (yi - Ys2) * (yi -
 Ys2) < R * R) {
                    get(p, i, j) = -0.1f;
                }
                else {
                    get(p, i, j) = 0.0f;
                }
                get(F0, i, j) = 0.0f;
```

```
            }
        }
#pragma omp barrier

        /* compute process */
#pragma omp single
        start = clock();

        for (int n = 0; n < Nt - 1; n += 6) {
#pragma omp single
            global_v_delta = _mm256_setzero_ps();

            v_delta = _mm256_setzero_ps();

            int num_thread = omp_get_thread_num();
            int shift = num_thread * CHUNK;

            int starting_correction = (num_thread == 0 ? 0 : 1);    // correction
 for zero thread
            int finalizing_correction = (num_thread == THREADS -
 1 ? 0 : 1);    // correction for last thread

            for (int k = 2 - 7 * starting_correction; k < 5 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F0, F1, p, k + shift);
            }

            for (int k = 2 - 6 * starting_correction; k < 4 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F1, F0, p, k + shift);
            }

            for(int k = 2 - 5 * starting_correction; k < 3 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F0, F1, p, k + shift);
            }

            for (int k = 2 - 4 * starting_correction; k < 2 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F1, F0, p, k + shift);
            }

            for (int k = 2 - 3 * starting_correction; k < 1 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F0, F1, p, k + shift);
```

```
            }

            int last_element = shift + CHUNK;

            for (int i = shift + 7 - 2 * starting_correction; i < last_element -
 3 - 2 * finalizing_correction; i++) {
                compute_process(F0, F1, p, i);
                compute_process(F1, F0, p, i - 1);
                compute_process(F0, F1, p, i - 2);
                compute_process(F1, F0, p, i - 3);
                compute_process(F0, F1, p, i - 4);
                compute_process(F1, F0, p, i - 5);
            }

#pragma omp barrier

            for (int k = 0; k < 1 + finalizing_correction; k++) {
                compute_process(F1, F0, p, last_element - 4 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 2 + 2 * finalizing_correction; k++) {
                compute_process(F0, F1, p, last_element - 5 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 3 + 3 * finalizing_correction; k++) {
                compute_process(F1, F0, p, last_element - 6 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 4 + 4 * finalizing_correction; k++) {
                compute_process(F0, F1, p, last_element - 7 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 5 + 5 * finalizing_correction; k++) {
                compute_process(F1, F0, p, last_element - 8 -
 2 * finalizing_correction + k);
            }
#pragma omp barrier

#pragma omp critical
            {
                global_v_delta = _mm256_max_ps(v_delta, global_v_delta);
```

```
            }
        }

#pragma omp single
        end = clock();
    }
    /* print max delta */
    float max_delta = 0.0f;
    float* vec_delta = (float*)(&global_v_delta);
    for (int i = 0; i < VECTOR_SIZE; i++) {
        max_delta = max(max_delta, vec_delta[i]);
    }
    cout << "n = " << Nt - 1 << " sigma = " << max_delta << endl;
    cout << "Total time: " << (double)(end -
 start) / CLOCKS_PER_SEC << " sec." << endl;

    cout << "Generating plot..." << endl;
    auto plot_file = fstream("computation.plot", ios::binary | ios::trunc | ios::
out);
    if (!plot_file.good()) {
        cerr << "Error while openning file!" << endl;
        return -1;
    }
    plot_file.write((char*)F0, GRID_SIZE * sizeof(float));
    plot_file.close();

    system("wsl gnuplot plot_script");  // call Linux to create plot

    _mm_free(F0);
    _mm_free(F1);
    _mm_free(p);

    return 0;
}
```

ПРИЛОЖЕНИЕ С. Программа 3

```
constexpr auto ALIGN = 32;

constexpr auto VECTOR_SIZE = 8;
constexpr auto SHIFT1 = 1;

constexpr auto Nx = 9000;
constexpr auto Ny = 9000;
constexpr auto Nt = 111;
```

```
constexpr float Xa = 0.0f;
constexpr float Xb = 4.0f;
constexpr float Ya = 0.0f;
constexpr float Yb = 4.0f;

constexpr float hx = ((Xb - Xa) / (Nx - 1));
constexpr float hy = ((Yb - Ya) / (Ny - 1));
constexpr float coeff1 = (0.2f / ((1.0f / (hx * hx) + 1.0f / (hy * hy))));
constexpr float coeff2 = (0.5f * (5.0f / (hx * hx) - 1.0f / (hy * hy)));
constexpr float coeff2b = (0.5f * (5.0f / (hy * hy) - 1.0f / (hx * hx)));
constexpr float coeff3 = (0.25f * (1.0f / (hx * hx) + 1.0f / (hy * hy)));

inline float X(size_t j) { return (Xa + (j)*hx); }
inline float Y(size_t i) { return (Ya + (i)*hy); }

constexpr float Xs1 = (Xa + (Xb - Xa) / 3.0f);
constexpr float Xs2 = (Xa + (Xb - Xa) * 2.0f / 3.0f);
constexpr float Ys1 = (Ya + (Yb - Ya) * 2.0f / 3.0f);
constexpr float Ys2 = (Ya + (Yb - Ya) / 3.0f);
constexpr float R = (0.1f * ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)));

constexpr float GRID_SIZE = (Nx * Ny);
constexpr int THREADS = 8;
constexpr int CHUNK = Ny / THREADS;

constexpr int CACHE_LINE_SIZE = 64; // bytes, from CPU-Z
inline int RED_ZONE_FLAG(int thread_num) { return CACHE_LINE_SIZE * thread_num; }
inline int BLUE_ZONE_FLAG(int thread_num) { return CACHE_LINE_SIZE * thread_num +
 1; }

inline float& get(float* p, size_t i, size_t j) { return p[(i)*Nx + (j)]; }

#include <iostream>
#include <fstream>
#include <cerrno>
#include <ctime>
#include <algorithm>
#include <immintrin.h>
#include <omp.h>
using namespace std;

/* prepare for vectorization */
__m256 v_coeff1;
__m256 v_coeff2;
__m256 v_coeff2b;
```

```
__m256 v_coeff3;
__m256 v_delta;
__m256 global_v_delta;
__m256 v_coeff4;
__m256 v_coeff5;

inline void compute_process(float* F0, float* F1, float* p, int i) {
    for (int j = 1; j < Nx / VECTOR_SIZE - 1; j++) {
        __m256 rez = _mm256_mul_ps(v_coeff1, (
            _mm256_fmadd_ps(v_coeff3, _mm256_add_ps(((__m256*)(F0 + (i -
 1) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i - 1) * Nx + SHIFT1))[j]),
                _mm256_fmadd_ps(v_coeff2b, _mm256_add_ps(((__m256*)(F0 + (i -
 1) * Nx))[j], ((__m256*)(F0 + (i + 1) * Nx))[j]),
                    _mm256_fmadd_ps(v_coeff2, _mm256_add_ps(((__m256*)(F0 + (i)*N
x - SHIFT1))[j], ((__m256*)(F0 + (i)*Nx + SHIFT1))[j]),
                        _mm256_fmadd_ps(v_coeff3, _mm256_add_ps(((__m256*)(F0 + (
i + 1) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i + 1) * Nx + SHIFT1))[j]),

                            _mm256_fmadd_ps(v_coeff5, ((__m256*)(p + (i)*Nx))[j],
                                _mm256_mul_ps(v_coeff4, (
                                    _mm256_add_ps(
                                        _mm256_add_ps(
                                            _mm256_add_ps(((__m256*)(p + (i -
 1) * Nx))[j], ((__m256*)(p + (i)*Nx - SHIFT1))[j]),
                                                ((__m256*)(p + (i)*Nx + SHIFT1))[j]),
                                            ((__m256*)(p + (i + 1) * Nx))[j])
                                    )
                                )
                            )
                        )
                    )
                )
            )
            )
        );

        __m256 prev_rez = ((__m256*)(F1 + (i)*Nx))[j];
        v_delta = _mm256_max_ps(v_delta, _mm256_max_ps(_mm256_sub_ps(prev_rez, re
z), _mm256_sub_ps(rez, prev_rez)));
        ((__m256*)(F1 + (i)*Nx))[j] = rez;
    }
}

int main() {
    /* allocate memory */
```

```
    float *F0 = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *F1 = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *p = (float*)_mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    int *synchro_flags = (int*)calloc(THREADS * CACHE_LINE_SIZE, sizeof(int));

    if(!F0 || !F1 || !p || !synchro_flags) {
        perror("_mm_malloc");
        exit(errno);
    }

    /* prepare for vectorization */
    v_coeff1 = _mm256_set1_ps(coeff1);
    v_coeff2 = _mm256_set1_ps(coeff2);
    v_coeff2b = _mm256_set1_ps(coeff2b);
    v_coeff3 = _mm256_set1_ps(coeff3);
    v_delta = _mm256_setzero_ps();
    v_coeff4 = _mm256_set1_ps(0.25f);
    v_coeff5 = _mm256_set1_ps(2.0f);

    clock_t start, end;

#pragma omp parallel proc_bind(close) num_threads(THREADS) private(v_delta) // OM
P_PLACES is "cores" by default
    {
        /* init arrays */
#pragma omp for schedule(static, CHUNK)
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                float xj = X(j);
                float yi = Y(i);
                if ((xj - Xs1) * (xj - Xs1) + (yi - Ys1) * (yi - Ys1) < R * R) {
                    get(p, i, j) = 0.1f;
                }
                else if ((xj - Xs2) * (xj - Xs2) + (yi - Ys2) * (yi -
 Ys2) < R * R) {
                    get(p, i, j) = -0.1f;
                }
                else {
                    get(p, i, j) = 0.0f;
                }
                get(F0, i, j) = 0.0f;
            }
        }
#pragma omp barrier
```

```
        /* compute process */
#pragma omp single
        start = clock();

        for (int n = 0; n < Nt - 1; n += 6) {
#pragma omp single
            global_v_delta = _mm256_setzero_ps();

            v_delta = _mm256_setzero_ps();

            int num_thread = omp_get_thread_num();
            int shift = num_thread * CHUNK;

            int starting_correction = (num_thread == 0 ? 0 : 1);    // correction
 for zero thread
            int finalizing_correction = (num_thread == THREADS -
 1 ? 0 : 1);    // correction for last thread

            int value = -1;
            if (num_thread != 0) {
#pragma omp atomic read
                value = synchro_flags[BLUE_ZONE_FLAG(num_thread - 1)];
                while (value < n) {
#pragma omp atomic read
                    value = synchro_flags[BLUE_ZONE_FLAG(num_thread - 1)];
                }
            }

            for (int k = 2 - 7 * starting_correction; k < 5 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F0, F1, p, k + shift);
            }

            for (int k = 2 - 6 * starting_correction; k < 4 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F1, F0, p, k + shift);
            }

            for (int k = 2 - 5 * starting_correction; k < 3 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F0, F1, p, k + shift);
            }

            for (int k = 2 - 4 * starting_correction; k < 2 + 2 * (1 -
 starting_correction); k++) {
```

```
                compute_process(F1, F0, p, k + shift);
            }

            for (int k = 2 - 3 * starting_correction; k < 1 + 2 * (1 -
 starting_correction); k++) {
                compute_process(F0, F1, p, k + shift);
            }

            int last_element = shift + CHUNK;

            for (int i = shift + 7 - 2 * starting_correction; i < last_element -
 3 - 2 * finalizing_correction; i++) {
                compute_process(F0, F1, p, i);
                compute_process(F1, F0, p, i - 1);
                compute_process(F0, F1, p, i - 2);
                compute_process(F1, F0, p, i - 3);
                compute_process(F0, F1, p, i - 4);
                compute_process(F1, F0, p, i - 5);
            }


#pragma omp atomic write
            synchro_flags[RED_ZONE_FLAG(num_thread)] = n + 6;

            if (num_thread != THREADS - 1) {
#pragma omp atomic read
                value = synchro_flags[RED_ZONE_FLAG(num_thread + 1)];
                while (value < n) {
#pragma omp atomic read
                    value = synchro_flags[RED_ZONE_FLAG(num_thread + 1)];
                }
            }

            for (int k = 0; k < 1 + finalizing_correction; k++) {
                compute_process(F1, F0, p, last_element - 4 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 2 + 2 * finalizing_correction; k++) {
                compute_process(F0, F1, p, last_element - 5 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 3 + 3 * finalizing_correction; k++) {
```

```
                compute_process(F1, F0, p, last_element - 6 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 4 + 4 * finalizing_correction; k++) {
                compute_process(F0, F1, p, last_element - 7 -
 2 * finalizing_correction + k);
            }

            for (int k = 0; k < 5 + 5 * finalizing_correction; k++) {
                compute_process(F1, F0, p, last_element - 8 -
 2 * finalizing_correction + k);
            }

#pragma omp atomic write
            synchro_flags[BLUE_ZONE_FLAG(num_thread)] = n + 6;

#pragma omp critical
            {
                global_v_delta = _mm256_max_ps(v_delta, global_v_delta);
            }
        }

#pragma omp single
        end = clock();
    }
    /* print max delta */
    float max_delta = 0.0f;
    float* vec_delta = (float*)(&global_v_delta);
    for (int i = 0; i < VECTOR_SIZE; i++) {
        max_delta = max(max_delta, vec_delta[i]);
    }
    cout << "n = " << Nt - 1 << " sigma = " << max_delta << endl;
    cout << "Total time: " << (double)(end -
 start) / CLOCKS_PER_SEC << " sec." << endl;

    cout << "Generating plot..." << endl;
    auto plot_file = fstream("computation.plot", ios::binary | ios::trunc | ios::
out);
    if (!plot_file.good()) {
        cerr << "Error while openning file!" << endl;
        return -1;
    }
    plot_file.write((char*)F0, GRID_SIZE * sizeof(float));
    plot_file.close();
```

```
    system("wsl gnuplot plot_script");  // call Linux to create plot

    _mm_free(F0);
    _mm_free(F1);
    _mm_free(p);

    return 0;
}
```