lab_4

Распараллелена на потоки, Синхронизация барьерами

4 потока | 8,8 секунд

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# gcc main_pthread_second.c -lm -O3 -march=native -pthread
 -o pthread.out
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4
main: Finising Jacobi method with pthreads. Time=9.05785s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4
main: Finising Jacobi method with pthreads. Time=8.73794s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4
main: Finising Jacobi method with pthreads. Time=8.72204s
```

8 потоков | 9,3 секунды

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread.out 8
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=8
main: Finising Jacobi method with pthreads. Time=9.3757s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread.out 8
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=8
main: Finising Jacobi method with pthreads. Time=9.32579s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread.out 8
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=8
main: Finising Jacobi method with pthreads. Time=9.29185s
```

Распараллелена на потоки, Оптимизирована по памяти, Синхронизация барьерами

4 потока, 4 итерации за шаг | 4,5 - 5 секунд

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# gcc main_pthread_second_lines.c -lm -O3 -march=native -pthread -o pthread_lines.out
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=4
main: Finising Jacobi method with pthreads. Time=4.50983s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=4
main: Finising Jacobi method with pthreads. Time=4.58021s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=4
main: Finising Jacobi method with pthreads. Time=5.00048s
```

4 потоков, 8 итераций за шаг | 3,8 - 4 секунды

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# gcc main_pthread_second_lines.c -lm -O3 -march=native -pthread -o pthread_lines.out
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=3.87644s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=3.99493s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=4.02561s
```

8 потоков, 8 итераций за шаг | 5,2 - 5,4 секунды

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# gcc main_pthread_second_lines.c -lm -O3 -march=native -pthread -o pthread_lines.out
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 8
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=8, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=5.25557s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 8
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=8, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=5.44403s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines.out 8
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=8, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=5.43018s
```

Распараллелена на потоки, Оптимизирована по памяти, Синхронизация флаговыми переменными и барьером

4 потока, 4 итерации за шаг | 4,6 - 5 секунд

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# gcc main_pthread_second_lines_sync.c -lm -O3 -march=native -pthread -o pthread_lines_sync.out
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines_sync.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=4
main: Finising Jacobi method with pthreads. Time=4.61394s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines_sync.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=4
main: Finising Jacobi method with pthreads. Time=4.86976s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines_sync.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=4
main: Finising Jacobi method with pthreads. Time=4.98975s
```

4 потока, 8 итераций за шаг | 3,9 - 4 секунды

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# gcc main_pthread_second_lines_sync.c -lm -O3 -march=native -pthread -o pthread_lines_sync.out
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines_sync.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=3.89884s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines_sync.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=4.02066s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# ./pthread_lines_sync.out 4
main: Starting Jacobi method with pthreads. N_X=8000, N_Y=8000, N_T=128, NUM_THREADS=4, LINE_NUMBER=8
main: Finising Jacobi method with pthreads. Time=3.99865s
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4#
```

Вывод

В итоге, больше всего на оптимизацию программы повлияла оптимизация по памяти (выполнения нескольких итераций за шаг)

В моей программе, то, какой способ синхронизации выбран (барьер или атомарные флаговые переменные) повлиял мало на оптимизацию (правда я не донца реализовал программу из пункта 3, частично используются атомарные флаговые переменные, барьеры также используются)

lscpu

```
root@DESKTOP-BUBJJQK:~/epsmim/Lab_4# lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          48 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   8
  On-line CPU(s) list:    0-7
Vendor ID:                AuthenticAMD
  Model name:             AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx
    CPU family:           23
    Model:                24
    Thread(s) per core:   2
    Core(s) per socket:   4
    Socket(s):            1
    Stepping:             1
    BogoMIPS:             4192.13
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mm
                          x fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_g
                          ood nopl tsc_reliable nonstop_tsc cpuid extd_apicid pni pclmulqdq ssse3 fma cx16 s
                          se4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm cmp_legacy
                          cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw topoext ssbd ibpb vmmcall fsgs
                          base bmi1 avx2 smep bmi2 rdseed adx smap clflushopt sha_ni xsaveopt xsavec xgetbv1
                           clzero xsaveerptr virt_ssbd arat
Virtualization features:
  Hypervisor vendor:      Microsoft
  Virtualization type:    full
Caches (sum of all):
  L1d:                    128 KiB (4 instances)
  L1i:                    256 KiB (4 instances)
  L2:                     2 MiB (4 instances)
  L3:                     4 MiB (1 instance)
Vulnerabilities:
  Gather data sampling:   Not affected
  Itlb multihit:          Not affected
```

```
  L1tf:                  Not affected
  Mds:                   Not affected
  Meltdown:              Not affected
  Mmio stale data:       Not affected
  Reg file data sampling: Not affected
  Retbleed:              Mitigation; untrained return thunk; SMT vulnerable
  Spec rstack overflow:  Mitigation; safe RET
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl and seccomp
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:            Mitigation; Retpolines; IBPB conditional; STIBP disabled; RSB filling; PBRSB-eIBRS
                          Not affected; BHI Not affected
  Srbds:                 Not affected
  Tsx async abort:       Not affected
```

Листинг программы для пункта 1

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <xmmintrin.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>


#define X_A 0.0f
#define X_B 4.0f
#define Y_A 0.0f
#define Y_B 4.0f

#define N_X 8000
```

```
#define N_Y 8000
#define N_T 128

#define VECTOR_SIZE_IN_FLOATS 4
#define VECTORS_NUMBER_IN_LINE ((N_X - 2) / VECTOR_SIZE_IN_FLOATS)
#define REMAINS_IN_LINE ((N_X - 2) % VECTOR_SIZE_IN_FLOATS)

#define H_X ((X_B - X_A) / (N_X - 1))
#define H_Y ((Y_B - Y_A) / (N_Y - 1))

#define X_s1 (X_A + (X_B - X_A) / 3.0f)
#define Y_s1 (Y_A + (Y_B - Y_A) * 2.0f / 3.0f)
#define X_s2 (X_A + (X_B - X_A) * 2.0f / 3.0f)
#define Y_s2 (Y_A + (Y_B - Y_A) / 3.0f)

#define MAIN_COEF (0.2f / (1.0f / (H_X * H_X) + 1.0f / (H_Y * H_Y)))
#define FIRST_COEF (2.5f / (H_X * H_X) - 0.5f / (H_Y * H_Y))
#define SECOND_COEF (2.5f / (H_Y * H_Y) - 0.5f / (H_X * H_X))
#define THIRD_COEF (0.25f / (H_X * H_X) + 0.25f / (H_Y * H_Y))

typedef struct {
    int thread_id;
    int num_threads;

    float *local_phi;
    float *local_phi_new;
    float *local_rho;

    int local_rows;
    int global_start_row_idx;
} thread_data_t;


typedef struct {
```

```
52      float* phi_storage;
53      float* rho_storage;
54      float* step_deltas;
55      thread_data_t* threads_args;
56      FILE* phi_output_file;
57      FILE* rho_output_file;
58      pthread_barrier_t barrier;
59  } task_t;
60
61  __m128 main_coef_m128_g;
62  __m128 first_coef_m128_g;
63  __m128 second_coef_m128_g;
64  __m128 third_coef_m128_g;
65
66  task_t task_g;
67
68  static pthread_mutex_t print_mutex;
69  volatile float global_delta = 1.0f;
70  volatile int current_iteration_g = 0;
71  volatile int stop_all_threads_g = 0;
72
73  void set_cpu(int n) {
74      int err;
75      cpu_set_t cpuset;
76      pthread_t tid = pthread_self();
77
78      CPU_ZERO(&cpuset);
79      CPU_SET(n, &cpuset);
80
81      err = pthread_setaffinity_np(tid, sizeof(cpu_set_t), &cpuset);
82      if (err) {
83          printf("set_cpu: pthread_setaffinity() failed for cpu %d\n", n);
84          return;
85      }
```

```
}

float compute_delta(float* local_phi, float* local_phi_new, int local_rows) {
    float d;
    float step_delta = -1.0f;
    for (int i = 0; i < local_rows; i++) {
        for (int j = 0; j < N_X; j++) {
            d = fabsf(local_phi[i * N_X + j] - local_phi_new[i * N_X + j]);
            if (d > step_delta) {
                step_delta = d;
            }
        }
    }
    return step_delta;
}

void compute_line(float* phi_new, float* phi, float* rho, int line_index) {
    int index;
    for (int j = 1; j < VECTORS_NUMBER_IN_LINE * VECTOR_SIZE_IN_FLOATS + 1; j += VECTOR_SIZE_IN_FLOATS) {
        index = line_index + j;

        __m128 v_phi_left     = _mm_loadu_ps(&phi[index - 1]);
        __m128 v_phi_right    = _mm_loadu_ps(&phi[index + 1]);
        __m128 v_phi_bottom   = _mm_loadu_ps(&phi[index - N_X]);
        __m128 v_phi_top      = _mm_loadu_ps(&phi[index + N_X]);
        __m128 v_phi_bot_left = _mm_loadu_ps(&phi[index - N_X - 1]);
        __m128 v_phi_bot_right= _mm_loadu_ps(&phi[index - N_X + 1]);
        __m128 v_phi_top_left = _mm_loadu_ps(&phi[index + N_X - 1]);
        __m128 v_phi_top_right= _mm_loadu_ps(&phi[index + N_X + 1]);

        __m128 v_rho_center   = _mm_loadu_ps(&rho[index]);
        __m128 v_rho_bottom   = _mm_loadu_ps(&rho[index - N_X]);
        __m128 v_rho_top      = _mm_loadu_ps(&rho[index + N_X]);
        __m128 v_rho_left     = _mm_loadu_ps(&rho[index - 1]);
```

```
        __m128 v_rho_right    = _mm_loadu_ps(&rho[index + 1]);

        __m128 first_line    = _mm_add_ps(_mm_mul_ps(first_coef_m128_g, _mm_add_ps(v_phi_left, v_phi_right)),
                                          _mm_mul_ps(second_coef_m128_g, _mm_add_ps(v_phi_bottom, v_phi_top)));
        __m128 second_line   = _mm_mul_ps(third_coef_m128_g,
                                          _mm_add_ps(_mm_add_ps(v_phi_bot_left, v_phi_top_left),
                                                     _mm_add_ps(v_phi_bot_right, v_phi_top_right)));
        __m128 third_line    = _mm_add_ps(_mm_mul_ps(_mm_set1_ps(0.25f),
                                                     _mm_add_ps(_mm_add_ps(v_rho_bottom, v_rho_top),
                                                                _mm_add_ps(v_rho_left, v_rho_right))),
                                          _mm_mul_ps(_mm_set1_ps(2.0f), v_rho_center));
        __m128 result        = _mm_mul_ps(main_coef_m128_g,
                                          _mm_add_ps(first_line, _mm_add_ps(second_line, third_line)));

        _mm_storeu_ps(&phi_new[index], result);
    }

    for (int j = VECTORS_NUMBER_IN_LINE * VECTOR_SIZE_IN_FLOATS + 1; j < N_X - 1; j++) {
        index = line_index + j;

        phi_new[index] = MAIN_COEF * (FIRST_COEF * (phi[index - 1] + phi[index + 1]) +
                                      SECOND_COEF * (phi[index - N_X] + phi[index + N_X]) +
                                      THIRD_COEF * (phi[index - N_X - 1] + phi[index - N_X + 1] + phi[index + N_X - 1] +
phi[index + N_X + 1]) +
                                      2.0f * rho[index] +
                                      0.25f * (rho[index - N_X] + rho[index + N_X] + rho[index - 1] + rho[index + 1]));
    }
}

void swap_float_ptr(float** a, float** b) {
    float* tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
153
154 void fill_file_strip(int tid, int num_threads, FILE* fd, float* local_matrix, int local_rows) {
155     if (fd == NULL) {
156         printf("tid=%d: fd is closed", tid);
157         return;
158     }
159
160     for (int i = 0; i < local_rows; i++) {
161         for (int j = 0; j < N_X; j++) {
162             fprintf(fd, "%f\t", local_matrix[i * N_X + j]);
163         }
164         fprintf(fd, "\n");
165     }
166 }
167
168 void fill_file_pthread(const thread_data_t* data, FILE* fd, const int flag) {
169     if (fd == NULL) {
170         printf("tid=%d: fd is closed\n", data->thread_id);
171         return;
172     }
173
174     for (int current_writer_id = 0; current_writer_id < data->num_threads; current_writer_id++) {
175         if (data->thread_id == current_writer_id) {
176             pthread_mutex_lock(&print_mutex);
177
178             fill_file_strip(data->thread_id, data->num_threads, fd, (flag == 0) ? data->local_rho : data->local_phi,
    data->local_rows);
179
180             pthread_mutex_unlock(&print_mutex);
181         }
182         pthread_barrier_wait(&task_g.barrier);
183     }
184 }
185
```

```
186 void init_rho_pthread(int tid, int num_threads, float* local_rho, int local_rows, int global_start_row_idx) {
187     const float R = 0.1f * fminf(X_B - X_A, Y_B - Y_A);
188     int y_coord;
189     for (int i = 0; i < local_rows; i++) {
190         y_coord = global_start_row_idx + i;
191         for (int j = 0; j < N_X; j++) {
192             if ((X_A + j * H_X - X_s1) * (X_A + j * H_X - X_s1) + (Y_A + y_coord * H_Y - Y_s1) * (Y_A + y_coord * H_Y -
    Y_s1) < R * R) {
193                 local_rho[i * N_X + j] = 1.0f;
194             }
195             else if ((X_A + j * H_X - X_s2) * (X_A + j * H_X - X_s2)  + (Y_A + y_coord * H_Y - Y_s2) * (Y_A + y_coord *
    H_Y - Y_s2) < R * R) {
196                 local_rho[i * N_X + j] = -1.0f;
197             }
198             else {
199                 local_rho[i * N_X + j] = 0.0f;
200             }
201         }
202     }
203 }
204
205 void* thread_worker_function(void* arg) {
206     thread_data_t* data = (thread_data_t*)arg;
207     int tid = data->thread_id;
208
209     set_cpu(tid);
210
211     // printf("tid[%d] = [%d %d]: local_rows=%d\tglobal_start_row_idx=%d\n", tid, getpid(), gettid(), data->local_rows,
    data->global_start_row_idx);
212
213     for (int i = 0; i < data->local_rows; i++) {
214         for (int j = 0; j < N_X; j++) {
215             data->local_phi[i * N_X + j] = 0.0f;
216             data->local_phi_new[i * N_X + j] = 0.0f;
```

```
217             }
218         }
219 
220         init_rho_pthread(data->thread_id, data->num_threads, data->local_rho, data->local_rows, data->global_start_row_idx);
221         pthread_barrier_wait(&task_g.barrier);
222 
223         // fill_file_pthread(data, task_g.rho_output_file, 0);
224         // pthread_barrier_wait(&task_g.barrier);
225 
226         float local_delta, max_local_delta;
227         int start_line_idx = (tid == 0) ? 1 : 0;
228         int finish_line_idx = data->local_rows - ((tid == data->num_threads - 1) ? 1 : 0);
229 
230         while (current_iteration_g < N_T && !stop_all_threads_g) {
231             // compute phi_new
232             for (int i = start_line_idx; i < finish_line_idx; i++) {
233                 compute_line(data->local_phi_new, data->local_phi, data->local_rho, i * N_X);
234             }
235 
236             // delta
237             local_delta = compute_delta(data->local_phi, data->local_phi_new, data->local_rows);
238             task_g.step_deltas[tid] = local_delta;
239             pthread_barrier_wait(&task_g.barrier);
240             if (tid == 0) {
241                 max_local_delta = task_g.step_deltas[0];
242                 for(int t = 1; t < data->num_threads; t++) {
243                     if ((task_g.step_deltas[t] - max_local_delta) > 0.000001) {
244                         max_local_delta = task_g.step_deltas[t];
245                     }
246                 }
247 
248                 if ((max_local_delta - global_delta) < 0.000001) {
249                     global_delta = max_local_delta;
250                     // printf("iter_number=%d\tglobal_delta = %f\n", current_iteration_g, global_delta);
```

```
            }
            else {
                printf("ERROR: DELTA IS GROWWING: global_delta = %f, iter_delta = %f\n", global_delta, max_local_delta);
                stop_all_threads_g = 1;
            }
        }

        // swap
        swap_float_ptr(&(data->local_phi), &(data->local_phi_new));
        if(tid == 0) current_iteration_g++;
        pthread_barrier_wait(&task_g.barrier);
    }

    // pthread_barrier_wait(&task_g.barrier);
    // fill_file_pthread(data, task_g.phi_output_file, 1);
    return NULL;
}

void free_task(int num_threads) {
    printf("[%d, %d]: free_task()\n", getpid(), gettid());

    pthread_barrier_destroy(&task_g.barrier);
    if (task_g.rho_output_file != NULL) {
        fclose(task_g.rho_output_file);
        task_g.rho_output_file = NULL;
    }
    if (task_g.phi_output_file != NULL) {
        fclose(task_g.phi_output_file);
        task_g.phi_output_file = NULL;
    }
    if (task_g.threads_args != NULL) {
        free(task_g.threads_args);
        task_g.threads_args = NULL;
    }
```

```
285         if (task_g.step_deltas != NULL) {
286             free(task_g.step_deltas);
287             task_g.step_deltas = NULL;
288         }
289         if (task_g.rho_storage != NULL) {
290             free(task_g.rho_storage);
291             task_g.rho_storage = NULL;
292         }
293         if (task_g.phi_storage != NULL) {
294             free(task_g.phi_storage);
295             task_g.phi_storage = NULL;
296         }
297     }
298 
299     int allocate_task(int num_threads) {
300         printf("[%d %d]: allocate_task()\n", getpid(), gettid());
301 
302         task_g.phi_storage = (float*)malloc(2 * N_X * N_Y * sizeof(float));
303         task_g.rho_storage = (float*)malloc(N_X * N_Y * sizeof(float));
304         task_g.step_deltas = (float*)malloc(num_threads * sizeof(float));
305         task_g.threads_args = (thread_data_t*)malloc(num_threads * sizeof(thread_data_t));
306         task_g.rho_output_file = fopen("rho_pthread.txt", "wb");
307         task_g.phi_output_file = fopen("phi_pthread.txt", "wb");
308         int barrier_err = pthread_barrier_init(&task_g.barrier, NULL, num_threads);
309 
310         if (task_g.phi_storage == NULL
311             || task_g.rho_storage == NULL
312             || task_g.step_deltas == NULL
313             || task_g.threads_args == NULL
314             || task_g.phi_output_file == NULL
315             || task_g.rho_output_file == NULL
316             || barrier_err != 0
317         ) {
318             printf("[%d %d]: Failed to allocate program memory\n", getpid(), gettid());
```

```
319             free_task(num_threads);
320             return 1;
321         }
322
323         return 0;
324     }
325
326     int run_jacoby_method_pthread(int num_threads) {
327         current_iteration_g = 0;
328         global_delta = 1.0f;
329         stop_all_threads_g = 0;
330
331         if (N_Y - 2 <= 0) {
332             printf("main: Too small N_Y size\n");
333             return 1;
334         }
335         if (N_Y - 2 < num_threads) {
336             printf("main: Fewer computable rows (%d) than threads (%d). Some threads might not compute actual data rows.\n",
    N_Y - 2, num_threads);
337             num_threads = N_Y - 2;
338             printf("Now num_threads = %d", num_threads);
339         }
340
341         allocate_task(num_threads);
342
343         int base_rows_per_thread = N_Y / num_threads;
344         int remainder_rows = N_Y % num_threads;
345
346         int current_global_compute_row_start_idx = 0;
347         int local_strip_capacity = 0;
348         int memory_offset = 0;
349
350         for (int t = 0; t < num_threads; t++) {
351             task_g.threads_args[t].thread_id = t;
```

```
352         task_g.threads_args[t].num_threads = num_threads;
353 
354         task_g.threads_args[t].local_rows = base_rows_per_thread + ((t < remainder_rows) ? 1 : 0);
355         task_g.threads_args[t].global_start_row_idx = current_global_compute_row_start_idx;
356         current_global_compute_row_start_idx += task_g.threads_args[t].local_rows;
357 
358         task_g.threads_args[t].local_phi = task_g.phi_storage + memory_offset;
359         task_g.threads_args[t].local_phi_new = task_g.phi_storage + (N_X * N_Y) + memory_offset; // тот же, что и phi, но
    со сдвигом на матрицу N_X * N_Y
360         task_g.threads_args[t].local_rho = task_g.rho_storage + memory_offset;
361         memory_offset += task_g.threads_args[t].local_rows * N_X;
362     }
363 
364     pthread_t threads[num_threads];
365 
366     printf("main: Starting Jacobi method with pthreads. N_X=%d, N_Y=%d, N_T=%d, NUM_THREADS=%d\n", N_X, N_Y, N_T,
    num_threads);
367         long long t1, t2;
368         double tDiff;
369         struct timespec curTime;
370         clock_gettime(CLOCK_BOOTTIME, &curTime);
371         t1 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
372 
373 
374         for (int t = 0; t < num_threads; t++) {
375             if (pthread_create(&threads[t], NULL, thread_worker_function, &task_g.threads_args[t])) {
376                 printf("main: Failed to create thread %d\n", t);
377                 free_task(num_threads);
378                 return 1;
379             }
380         }
381 
382         for (int t = 0; t < num_threads; t++) {
383             if (pthread_join(threads[t], NULL)) {
```

```
                printf("main: Failed to join thread %d\n", t);
                free_task(num_threads);
                return 1;
            }
        }


        clock_gettime(CLOCK_BOOTTIME, &curTime);
        t2 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
        tDiff = (double) (t2 - t1) / 1000000000.0;
    printf("main: Finising Jacobi method with pthreads. Time=%gs\n", tDiff);

    free_task(num_threads);
    return 0;
}

int main(int argc, char* argv[]) {
    int pid = getpid();
    int tid = gettid();
    printf("main[%d %d] Hello\n", pid, tid);
    if (argc != 2) {
        printf("usage: %s <num_threads>\n", argv[0]);
        return 1;
    }

    int num_threads = atoi(argv[1]);
    if (num_threads <= 0) {
        printf("main[%d %d]: num_threads = %d, It must be > 0\n", pid, tid, num_threads);
        return 1;
    }
    if (num_threads > 8) {
        printf("main[%d %d]: num_threads = %d, It must be <= 8 (my num_kernels = 8)", pid, tid, num_threads);
        return 1;
    }
```

```
418
419       main_coef_m128_g  = _mm_set1_ps(MAIN_COEF);
420       first_coef_m128_g = _mm_set1_ps(FIRST_COEF);
421       second_coef_m128_g= _mm_set1_ps(SECOND_COEF);
422       third_coef_m128_g = _mm_set1_ps(THIRD_COEF);
423
424       pthread_mutex_init(&print_mutex, NULL);
425
426       if (run_jacoby_method_pthread(num_threads)) {
427           printf("main[%d %d]: run_jacoby_method_pthread() failed\n", pid, tid);
428           pthread_mutex_destroy(&print_mutex);
429           return 1;
430       }
431
432       pthread_mutex_destroy(&print_mutex);
433
434       return 0;
435   }
```

Листинг программы из пункта 2

```
1   #define _GNU_SOURCE
2   #include <stdio.h>
3   #include <math.h>
4   #include <stdlib.h>
5   #include <time.h>
6   #include <xmmintrin.h>
7   #include <pthread.h>
8   #include <string.h>
9   #include <unistd.h>
10
11
12  #define LINE_NUMBER 4
```

```
13
14  #define X_A 0.0f
15  #define X_B 4.0f
16  #define Y_A 0.0f
17  #define Y_B 4.0f
18
19  #define N_X 8000
20  #define N_Y 8000
21  #define N_T 128
22
23  #define VECTOR_SIZE_IN_FLOATS 4
24  #define VECTORS_NUMBER_IN_LINE ((N_X - 2) / VECTOR_SIZE_IN_FLOATS)
25  #define REMAINS_IN_LINE ((N_X - 2) % VECTOR_SIZE_IN_FLOATS)
26
27  #define H_X ((X_B - X_A) / (N_X - 1))
28  #define H_Y ((Y_B - Y_A) / (N_Y - 1))
29
30  #define X_s1 (X_A + (X_B - X_A) / 3.0f)
31  #define Y_s1 (Y_A + (Y_B - Y_A) * 2.0f / 3.0f)
32  #define X_s2 (X_A + (X_B - X_A) * 2.0f / 3.0f)
33  #define Y_s2 (Y_A + (Y_B - Y_A) / 3.0f)
34
35  #define MAIN_COEF (0.2f / (1.0f / (H_X * H_X) + 1.0f / (H_Y * H_Y)))
36  #define FIRST_COEF (2.5f / (H_X * H_X) - 0.5f / (H_Y * H_Y))
37  #define SECOND_COEF (2.5f / (H_Y * H_Y) - 0.5f / (H_X * H_X))
38  #define THIRD_COEF (0.25f / (H_X * H_X) + 0.25f / (H_Y * H_Y))
39
40  typedef struct {
41      int thread_id;
42      int num_threads;
43
44      float *local_phi;
45      float *local_phi_new;
46      float *local_rho;
```

```
47
48      int local_rows;
49      int global_start_row_idx;
50  } thread_data_t;
51
52
53  typedef struct {
54      float* phi_storage;
55      float* rho_storage;
56      float* step_deltas;
57      thread_data_t* threads_args;
58      FILE* phi_output_file;
59      FILE* rho_output_file;
60      pthread_barrier_t barrier;
61  } task_t;
62
63  __m128 main_coef_m128_g;
64  __m128 first_coef_m128_g;
65  __m128 second_coef_m128_g;
66  __m128 third_coef_m128_g;
67
68  task_t task_g;
69
70  static pthread_mutex_t print_mutex;
71  volatile float global_delta = 1.0f;
72  volatile int current_iteration_g = 0;
73  volatile int stop_all_threads_g = 0;
74
75  void set_cpu(int n) {
76      int err;
77      cpu_set_t cpuset;
78      pthread_t tid = pthread_self();
79
80      CPU_ZERO(&cpuset);
```

```
81      CPU_SET(n, &cpuset);
82
83      err = pthread_setaffinity_np(tid, sizeof(cpu_set_t), &cpuset);
84      if (err) {
85          printf("set_cpu: pthread_setaffinity() failed for cpu %d\n", n);
86          return;
87      }
88  }
89
90  float compute_delta(float* local_phi, float* local_phi_new, int local_rows) {
91      float d;
92      float step_delta = -1.0f;
93      for (int i = 0; i < local_rows; i++) {
94          for (int j = 0; j < N_X; j++) {
95              d = fabsf(local_phi[i * N_X + j] - local_phi_new[i * N_X + j]);
96              if (d > step_delta) {
97                  step_delta = d;
98              }
99          }
100     }
101     return step_delta;
102 }
103
104 void compute_line(float* phi_new, float* phi, float* rho, int line_index) {
105     int index;
106     for (int j = 1; j < VECTORS_NUMBER_IN_LINE * VECTOR_SIZE_IN_FLOATS + 1; j += VECTOR_SIZE_IN_FLOATS) {
107         index = line_index + j;
108
109         __m128 v_phi_left     = _mm_loadu_ps(&phi[index - 1]);
110         __m128 v_phi_right    = _mm_loadu_ps(&phi[index + 1]);
111         __m128 v_phi_bottom   = _mm_loadu_ps(&phi[index - N_X]);
112         __m128 v_phi_top      = _mm_loadu_ps(&phi[index + N_X]);
113         __m128 v_phi_bot_left = _mm_loadu_ps(&phi[index - N_X - 1]);
114         __m128 v_phi_bot_right= _mm_loadu_ps(&phi[index - N_X + 1]);
```

```
115             __m128 v_phi_top_left = _mm_loadu_ps(&phi[index + N_X - 1]);
116             __m128 v_phi_top_right= _mm_loadu_ps(&phi[index + N_X + 1]);
117 
118             __m128 v_rho_center   = _mm_loadu_ps(&rho[index]);
119             __m128 v_rho_bottom   = _mm_loadu_ps(&rho[index - N_X]);
120             __m128 v_rho_top      = _mm_loadu_ps(&rho[index + N_X]);
121             __m128 v_rho_left     = _mm_loadu_ps(&rho[index - 1]);
122             __m128 v_rho_right    = _mm_loadu_ps(&rho[index + 1]);
123 
124             __m128 first_line    = _mm_add_ps(_mm_mul_ps(first_coef_m128_g, _mm_add_ps(v_phi_left, v_phi_right)),
125                                               _mm_mul_ps(second_coef_m128_g, _mm_add_ps(v_phi_bottom, v_phi_top)));
126             __m128 second_line   = _mm_mul_ps(third_coef_m128_g,
127                                               _mm_add_ps(_mm_add_ps(v_phi_bot_left, v_phi_top_left),
128                                                          _mm_add_ps(v_phi_bot_right, v_phi_top_right)));
129             __m128 third_line    = _mm_add_ps(_mm_mul_ps(_mm_set1_ps(0.25f),
130                                                          _mm_add_ps(_mm_add_ps(v_rho_bottom, v_rho_top),
131                                                                     _mm_add_ps(v_rho_left, v_rho_right))),
132                                               _mm_mul_ps(_mm_set1_ps(2.0f), v_rho_center));
133             __m128 result        = _mm_mul_ps(main_coef_m128_g,
134                                               _mm_add_ps(first_line, _mm_add_ps(second_line, third_line)));
135 
136             _mm_storeu_ps(&phi_new[index], result);
137         }
138 
139         for (int j = VECTORS_NUMBER_IN_LINE * VECTOR_SIZE_IN_FLOATS + 1; j < N_X - 1; j++) {
140             index = line_index + j;
141 
142             phi_new[index] = MAIN_COEF * (FIRST_COEF * (phi[index - 1] + phi[index + 1]) +
143                                           SECOND_COEF * (phi[index - N_X] + phi[index + N_X]) +
144                                           THIRD_COEF * (phi[index - N_X - 1] + phi[index - N_X + 1] + phi[index + N_X - 1] +
    phi[index + N_X + 1]) +
145                                           2.0f * rho[index] +
146                                           0.25f * (rho[index - N_X] + rho[index + N_X] + rho[index - 1] + rho[index + 1]));
147         }
```

```
148 }
149
150 void swap_float_ptr(float** a, float** b) {
151     float* tmp = *a;
152     *a = *b;
153     *b = tmp;
154 }
155
156 void fill_file_strip(int tid, int num_threads, FILE* fd, float* local_matrix, int local_rows) {
157     if (fd == NULL) {
158         printf("tid=%d: fd is closed", tid);
159         return;
160     }
161
162     for (int i = 0; i < local_rows; i++) {
163         for (int j = 0; j < N_X; j++) {
164             fprintf(fd, "%f\t", local_matrix[i * N_X + j]);
165         }
166         fprintf(fd, "\n");
167     }
168 }
169
170 void fill_file_pthread(const thread_data_t* data, FILE* fd, const int flag) {
171     if (fd == NULL) {
172         printf("tid=%d: fd is closed\n", data->thread_id);
173         return;
174     }
175
176     for (int current_writer_id = 0; current_writer_id < data->num_threads; current_writer_id++) {
177         if (data->thread_id == current_writer_id) {
178             pthread_mutex_lock(&print_mutex);
179
180             fill_file_strip(data->thread_id, data->num_threads, fd, (flag == 0) ? data->local_rho : data->local_phi,
    data->local_rows);
```

```
181
182             pthread_mutex_unlock(&print_mutex);
183         }
184         pthread_barrier_wait(&task_g.barrier);
185     }
186 }
187
188 void init_rho_pthread(int tid, int num_threads, float* local_rho, int local_rows, int global_start_row_idx) {
189     const float R = 0.1f * fminf(X_B - X_A, Y_B - Y_A);
190     int y_coord;
191     for (int i = 0; i < local_rows; i++) {
192         y_coord = global_start_row_idx + i;
193         for (int j = 0; j < N_X; j++) {
194             if ((X_A + j * H_X - X_s1) * (X_A + j * H_X - X_s1) + (Y_A + y_coord * H_Y - Y_s1) * (Y_A + y_coord * H_Y -
    Y_s1) < R * R) {
195                 local_rho[i * N_X + j] = 1.0f;
196             }
197             else if ((X_A + j * H_X - X_s2) * (X_A + j * H_X - X_s2)  + (Y_A + y_coord * H_Y - Y_s2) * (Y_A + y_coord *
    H_Y - Y_s2) < R * R) {
198                 local_rho[i * N_X + j] = -1.0f;
199             }
200             else {
201                 local_rho[i * N_X + j] = 0.0f;
202             }
203         }
204     }
205 }
206
207 void* thread_worker_function(void* arg) {
208     thread_data_t* data = (thread_data_t*)arg;
209     // printf("tid[%d] = [%d %d]: local_rows=%d\tglobal_start_row_idx=%d\n", data->thread_id, getpid(), gettid(), data-
    >local_rows, data->global_start_row_idx);
210
211     set_cpu(data->thread_id);
```

```
212
213     for (int i = 0; i < data->local_rows; i++) {
214         for (int j = 0; j < N_X; j++) {
215             data->local_phi[i * N_X + j] = 0.0f;
216             data->local_phi_new[i * N_X + j] = 0.0f;
217         }
218     }
219
220     init_rho_pthread(data->thread_id, data->num_threads, data->local_rho, data->local_rows, data->global_start_row_idx);
221     pthread_barrier_wait(&task_g.barrier);
222
223     // fill_file_pthread(data, task_g.rho_output_file, 0);
224     // pthread_barrier_wait(&task_g.barrier);
225
226
227     float local_delta, max_local_delta;
228
229     int tid = data->thread_id;
230     int num_threads = data->num_threads;
231
232     int start_red_idx = ((tid == 0) ? 1 : 0) + (LINE_NUMBER - 1);
233     int finish_red_idx = data->local_rows - ((tid != num_threads) ? (LINE_NUMBER - 1) : 0);
234
235     while (current_iteration_g < (N_T / LINE_NUMBER) && !stop_all_threads_g) {
236         // compute phi_new
237
238         // green step
239         {
240             /*EXAMPLE
241                 LINE_NUMBER = 4
242                 i=1: k=1: u[1]
243                 i=2: k=1: u[2]
244                      k=2: v[1]
245                 i=3: k=1: u[3]
```

```
246                      k=2: v[2]
247                      k=3: u[1]
248             */
249             if (tid == 0) {
250                 for(int i = 1; i < LINE_NUMBER; i++) {
251                     for(int k = 1; k <= i; k++) {
252                         if (k % 2 == 1) { // u
253                             compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k + 1) * N_X);
254                         }
255                         else {            // v
256                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k + 1) * N_X);
257                         }
258                     }
259                 }
260             }
261
262             /*EXAMPLE
263                 LINE_NUMBER = 4
264                 i=0: k=0: u[-1] , u[0]
265                 i=1: k=0: u[-2] , u[1]
266                      k=1: v[-1] , v[0]
267                 i=2: k=0: u[-3] , u[2]
268                      k=1: v[-2] , v[1]
269                      k=2: u`[-1], u`[0]
270             */
271             else {
272                 for(int i = 0; i < LINE_NUMBER - 1; i++) {
273                     for (int k = 0; k <= i; k++) {
274                         if (k % 2 == 0) { // u
275                             compute_line(data->local_phi_new, data->local_phi, data->local_rho, (-1 -(i - k)) * N_X);
276                             compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
277                         } else {          // v
278                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (-1 -(i - k)) * N_X);
279                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
```

```
                        }
                    }
                }
            }
        }

        // red step
        {
            /*EXAMPLE
                i=4: k=0: u[4]
                    k=1: v[3]
                    k=2: u[2]
                    k=3: v[1]
                ...
            */
            if (tid == 0) {
                for (int i = LINE_NUMBER; i < data->local_rows - (LINE_NUMBER - 1); i++) {
                    for (int k = 0; k < LINE_NUMBER; k++) {
                        if (k % 2 == 0) { // u
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
                        }
                        else {            // v
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
                        }
                    }
                }
            }
            /*EXAMPLE
                i=3: k=0: u[3]
                    k=1: v[2]
                    k=2: u[1]
                    k=3: v[0]
            */
            else if (tid == num_threads - 1) {
```

```
                for (int i = LINE_NUMBER - 1; i < data->local_rows; i++) {
                    for (int k = 0; k < LINE_NUMBER; k++) {
                        if (k % 2 == 0) { // u
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
                        }
                        else {            // v
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
                        }
                    }
                }
            }
            else {
                for (int i = LINE_NUMBER - 1; i < data->local_rows - (LINE_NUMBER - 1); i++) {
                    for (int k = 0; k < LINE_NUMBER; k++) {
                        if (k % 2 == 0) { // u
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
                        }
                        else {            // v
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
                        }
                    }
                }
            }
        }
        pthread_barrier_wait(&task_g.barrier);

        // blue step
        {
            /*EXAMPLE
                k=0: i=63: v[62]
                k=1: i=63: u[61]
                     i=64: u[62]
                k=2: i=63: v[60]
                     i=64: v[61]
```

```
                        i=65: v[62]
            */
            if (tid == num_threads - 1) {
                for (int k = 0; k < LINE_NUMBER - 1; k++) {
                    for(int i = data->local_rows - 1; i < data->local_rows + k; i++) {
                        if(k % 2 == 1) { // u
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k - 1) * N_X);
                        } else {         // v
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k - 1) * N_X);
                        }
                    }
                }
            }
            /*EXAMPLE
                k=0: i=13: v[12], v[13]
                k=1: i=13: u[11], u[13]
                    i=14: u[12], u[14]
                k=2: i=13: v[10], v[13]
                    i=14: v[11], v[14]
                    i=15: v[12], v[15]
            */
            else {
                for (int k = 0; k < LINE_NUMBER - 1; k++) {
                    for(int i = data->local_rows - (LINE_NUMBER - 1); i < data->local_rows - (LINE_NUMBER - 1) + k + 1;
    i++) {
                        if(k % 2 == 1) { // u
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k - 1) * N_X);
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i) * N_X);
                        } else {         // v
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k - 1) * N_X);
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i) * N_X);
                        }
                    }
                }
```

```
381             }
382         }
383 
384         // delta
385         local_delta = compute_delta(data->local_phi, data->local_phi_new, data->local_rows);
386         task_g.step_deltas[tid] = local_delta;
387 
388         pthread_barrier_wait(&task_g.barrier);
389         if (tid == 0) {
390             max_local_delta = task_g.step_deltas[0];
391             for(int t = 1; t < data->num_threads; t++) {
392                 if ((task_g.step_deltas[t] - max_local_delta) > 0.000001) {
393                     max_local_delta = task_g.step_deltas[t];
394                 }
395             }
396 
397             if ((max_local_delta - global_delta) < 0.000001) {
398                 global_delta = max_local_delta;
399                 // printf("iter_number=%d\tglobal_delta = %f\n", current_iteration_g, global_delta);
400             }
401             else {
402                 printf("ERROR: DELTA IS GROWWING: global_delta = %f, iter_delta = %f", global_delta, max_local_delta);
403                 stop_all_threads_g = 1;
404             }
405         }
406 
407         // swap
408         // swap_float_ptr(&(data->local_phi), &(data->local_phi_new));
409         if(tid == 0) current_iteration_g++;
410         pthread_barrier_wait(&task_g.barrier);
411     }
412 
413     // pthread_barrier_wait(&task_g.barrier);
414     // fill_file_pthread(data, task_g.phi_output_file, 1);
```

```
415
416         return NULL;
417     }
418
419     void free_task(int num_threads) {
420         printf("[%d, %d]: free_task()\n", getpid(), gettid());
421
422         pthread_barrier_destroy(&task_g.barrier);
423         if (task_g.rho_output_file != NULL) {
424             fclose(task_g.rho_output_file);
425             task_g.rho_output_file = NULL;
426         }
427         if (task_g.phi_output_file != NULL) {
428             fclose(task_g.phi_output_file);
429             task_g.phi_output_file = NULL;
430         }
431         if (task_g.threads_args != NULL) {
432             free(task_g.threads_args);
433             task_g.threads_args = NULL;
434         }
435         if (task_g.step_deltas != NULL) {
436             free(task_g.step_deltas);
437             task_g.step_deltas = NULL;
438         }
439         if (task_g.rho_storage != NULL) {
440             free(task_g.rho_storage);
441             task_g.rho_storage = NULL;
442         }
443         if (task_g.phi_storage != NULL) {
444             free(task_g.phi_storage);
445             task_g.phi_storage = NULL;
446         }
447     }
448
```

```
449  int allocate_task(int num_threads) {
450      printf("[%d %d]: allocate_task()\n", getpid(), gettid());
451
452      task_g.phi_storage = (float*)malloc(2 * N_X * N_Y * sizeof(float));
453      task_g.rho_storage = (float*)malloc(N_X * N_Y * sizeof(float));
454      task_g.step_deltas = (float*)malloc(num_threads * sizeof(float));
455      task_g.threads_args = (thread_data_t*)malloc(num_threads * sizeof(thread_data_t));
456      task_g.rho_output_file = fopen("rho_pthread.txt", "wb");
457      task_g.phi_output_file = fopen("phi_pthread.txt", "wb");
458      int barrier_err = pthread_barrier_init(&task_g.barrier, NULL, num_threads);
459
460      if (task_g.phi_storage == NULL
461          || task_g.rho_storage == NULL
462          || task_g.step_deltas == NULL
463          || task_g.threads_args == NULL
464          || task_g.phi_output_file == NULL
465          || task_g.rho_output_file == NULL
466          || barrier_err != 0
467      ) {
468          printf("[%d %d]: Failed to allocate program memory\n", getpid(), gettid());
469          free_task(num_threads);
470          return 1;
471      }
472
473      return 0;
474  }
475
476  int run_jacoby_method_pthread(int num_threads) {
477      current_iteration_g = 0;
478      global_delta = 1.0f;
479      stop_all_threads_g = 0;
480
481      if (N_Y - 2 <= 0) {
482          printf("main: Too small N_Y size\n");
```

```
483             return 1;
484         }
485         if (N_Y - 2 < num_threads) {
486             printf("main: Fewer computable rows (%d) than threads (%d). Some threads might not compute actual data rows.\n",
    N_Y - 2, num_threads);
487             num_threads = N_Y - 2;
488             printf("Now num_threads = %d", num_threads);
489         }
490 
491         allocate_task(num_threads);
492 
493         int base_rows_per_thread = N_Y / num_threads;
494         int remainder_rows = N_Y % num_threads;
495 
496         int current_global_compute_row_start_idx = 0;
497         int local_strip_capacity = 0;
498         int memory_offset = 0;
499 
500         for (int t = 0; t < num_threads; t++) {
501             task_g.threads_args[t].thread_id = t;
502             task_g.threads_args[t].num_threads = num_threads;
503 
504             task_g.threads_args[t].local_rows = base_rows_per_thread + ((t < remainder_rows) ? 1 : 0);
505             task_g.threads_args[t].global_start_row_idx = current_global_compute_row_start_idx;
506             current_global_compute_row_start_idx += task_g.threads_args[t].local_rows;
507 
508             task_g.threads_args[t].local_phi = task_g.phi_storage + memory_offset;
509             task_g.threads_args[t].local_phi_new = task_g.phi_storage + (N_X * N_Y) + memory_offset;
510             task_g.threads_args[t].local_rho = task_g.rho_storage + memory_offset;
511             memory_offset += task_g.threads_args[t].local_rows * N_X;
512         }
513 
514         pthread_t threads[num_threads];
515 
```

```
516     printf("main: Starting Jacobi method with pthreads. N_X=%d, N_Y=%d, N_T=%d, NUM_THREADS=%d\n", N_X, N_Y, N_T,
    num_threads);
517         long long t1, t2;
518         double tDiff;
519         struct timespec curTime;
520         clock_gettime(CLOCK_BOOTTIME, &curTime);
521         t1 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
522
523
524         for (int t = 0; t < num_threads; t++) {
525             if (pthread_create(&threads[t], NULL, thread_worker_function, &task_g.threads_args[t])) {
526                 printf("main: Failed to create thread %d\n", t);
527                 free_task(num_threads);
528                 return 1;
529             }
530         }
531
532         for (int t = 0; t < num_threads; t++) {
533             if (pthread_join(threads[t], NULL)) {
534                 printf("main: Failed to join thread %d\n", t);
535                 free_task(num_threads);
536                 return 1;
537             }
538         }
539
540
541         clock_gettime(CLOCK_BOOTTIME, &curTime);
542         t2 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
543         tDiff = (double) (t2 - t1) / 1000000000.0;
544     printf("main: Finising Jacobi method with pthreads. Time=%gs\n", tDiff);
545
546     free_task(num_threads);
547     return 0;
548 }
```

```
549
550 int main(int argc, char* argv[]) {
551     int pid = getpid();
552     int tid = gettid();
553     printf("main[%d %d] Hello\n", pid, tid);
554     if (argc != 2) {
555         printf("usage: %s <num_threads>\n", argv[0]);
556         return 1;
557     }
558
559     int num_threads = atoi(argv[1]);
560     if (num_threads <= 0) {
561         printf("main[%d %d]: num_threads = %d, It must be > 0\n", pid, tid, num_threads);
562         return 1;
563     }
564     if (num_threads > 8) {
565         printf("main[%d %d]: num_threads = %d, It must be <= 8 (my num_kernels = 8)", pid, tid, num_threads);
566         return 1;
567     }
568
569     main_coef_m128_g  = _mm_set1_ps(MAIN_COEF);
570     first_coef_m128_g = _mm_set1_ps(FIRST_COEF);
571     second_coef_m128_g= _mm_set1_ps(SECOND_COEF);
572     third_coef_m128_g = _mm_set1_ps(THIRD_COEF);
573
574     pthread_mutex_init(&print_mutex, NULL);
575
576     if (run_jacoby_method_pthread(num_threads)) {
577         printf("main[%d %d]: run_jacoby_method_pthread() failed\n", pid, tid);
578         pthread_mutex_destroy(&print_mutex);
579         return 1;
580     }
581
582     pthread_mutex_destroy(&print_mutex);
```

```
583
584         return 0;
585     }
```

Листинг программы из пункта 3

```
1   #define _GNU_SOURCE
2   #include <stdio.h>
3   #include <math.h>
4   #include <stdlib.h>
5   #include <time.h>
6   #include <xmmintrin.h>
7   #include <pthread.h>
8   #include <string.h>
9   #include <unistd.h>
10  #include <stdatomic.h>
11  #include <sched.h>
12
13  #define LINE_NUMBER 4
14  #define CACHE_LINE_SIZE 64
15
16  #define X_A 0.0f
17  #define X_B 4.0f
18  #define Y_A 0.0f
19  #define Y_B 4.0f
20
21  #define N_X 8000
22  #define N_Y 8000
23  #define N_T 128
24
25  #define VECTOR_SIZE_IN_FLOATS 4
26  #define VECTORS_NUMBER_IN_LINE ((N_X - 2) / VECTOR_SIZE_IN_FLOATS)
27  #define REMAINS_IN_LINE ((N_X - 2) % VECTOR_SIZE_IN_FLOATS)
```

```
28
29  #define H_X ((X_B - X_A) / (N_X - 1))
30  #define H_Y ((Y_B - Y_A) / (N_Y - 1))
31
32  #define X_s1 (X_A + (X_B - X_A) / 3.0f)
33  #define Y_s1 (Y_A + (Y_B - Y_A) * 2.0f / 3.0f)
34  #define X_s2 (X_A + (X_B - X_A) * 2.0f / 3.0f)
35  #define Y_s2 (Y_A + (Y_B - Y_A) / 3.0f)
36
37  #define MAIN_COEF (0.2f / (1.0f / (H_X * H_X) + 1.0f / (H_Y * H_Y)))
38  #define FIRST_COEF (2.5f / (H_X * H_X) - 0.5f / (H_Y * H_Y))
39  #define SECOND_COEF (2.5f / (H_Y * H_Y) - 0.5f / (H_X * H_X))
40  #define THIRD_COEF (0.25f / (H_X * H_X) + 0.25f / (H_Y * H_Y))
41
42
43  typedef struct {
44      _Atomic int val;
45      char padding[CACHE_LINE_SIZE - sizeof(_Atomic int)];
46  } cache_padded_atomic_int_t;
47
48  typedef struct {
49      int thread_id;
50      int num_threads;
51
52      float *local_phi;
53      float *local_phi_new;
54      float *local_rho;
55
56      int local_rows;
57      int global_start_row_idx;
58  } thread_data_t;
59
60
61  typedef struct {
```

```
62      float* phi_storage;
63      float* rho_storage;
64      float* step_deltas;
65      thread_data_t* threads_args;
66      FILE* phi_output_file;
67      FILE* rho_output_file;
68      pthread_barrier_t barrier;
69
70      cache_padded_atomic_int_t* red_step_completed_flags; // Флаги для попарной синхронизации между red и blue step
71
72      _Atomic int current_sync_generation;
73
74  } task_t;
75
76  __m128 main_coef_m128_g;
77  __m128 first_coef_m128_g;
78  __m128 second_coef_m128_g;
79  __m128 third_coef_m128_g;
80
81  task_t task_g;
82
83  static pthread_mutex_t print_mutex;
84  volatile float global_delta = 1.0f;
85  volatile int current_iteration_g = 0;
86  volatile int stop_all_threads_g = 0;
87
88  void set_cpu(int n) {
89      int err;
90      cpu_set_t cpuset;
91      pthread_t tid = pthread_self();
92
93      CPU_ZERO(&cpuset);
94      CPU_SET(n, &cpuset);
95
```

```
96      err = pthread_setaffinity_np(tid, sizeof(cpu_set_t), &cpuset);
97      if (err) {
98          printf("set_cpu: pthread_setaffinity() failed for cpu %d\n", n);
99          return;
100     }
101 }
102
103 float compute_delta(float* local_phi, float* local_phi_new, int local_rows) {
104     float d;
105     float step_delta = -1.0f;
106     for (int i = 0; i < local_rows; i++) {
107         for (int j = 0; j < N_X; j++) {
108             d = fabs(local_phi[i * N_X + j] - local_phi_new[i * N_X + j]);
109             if (d > step_delta) {
110                 step_delta = d;
111             }
112         }
113     }
114     return step_delta;
115 }
116
117 void compute_line(float* phi_new, float* phi, float* rho, int line_index) {
118     int index;
119     for (int j = 1; j < VECTORS_NUMBER_IN_LINE * VECTOR_SIZE_IN_FLOATS + 1; j += VECTOR_SIZE_IN_FLOATS) {
120         index = line_index + j;
121
122         __m128 v_phi_left     = _mm_loadu_ps(&phi[index - 1]);
123         __m128 v_phi_right    = _mm_loadu_ps(&phi[index + 1]);
124         __m128 v_phi_bottom   = _mm_loadu_ps(&phi[index - N_X]);
125         __m128 v_phi_top      = _mm_loadu_ps(&phi[index + N_X]);
126         __m128 v_phi_bot_left = _mm_loadu_ps(&phi[index - N_X - 1]);
127         __m128 v_phi_bot_right= _mm_loadu_ps(&phi[index - N_X + 1]);
128         __m128 v_phi_top_left = _mm_loadu_ps(&phi[index + N_X - 1]);
129         __m128 v_phi_top_right= _mm_loadu_ps(&phi[index + N_X + 1]);
```

```
130
131         __m128 v_rho_center   = _mm_loadu_ps(&rho[index]);
132         __m128 v_rho_bottom   = _mm_loadu_ps(&rho[index - N_X]);
133         __m128 v_rho_top      = _mm_loadu_ps(&rho[index + N_X]);
134         __m128 v_rho_left     = _mm_loadu_ps(&rho[index - 1]);
135         __m128 v_rho_right    = _mm_loadu_ps(&rho[index + 1]);
136
137         __m128 first_line    = _mm_add_ps(_mm_mul_ps(first_coef_m128_g, _mm_add_ps(v_phi_left, v_phi_right)),
138                                           _mm_mul_ps(second_coef_m128_g, _mm_add_ps(v_phi_bottom, v_phi_top)));
139         __m128 second_line   = _mm_mul_ps(third_coef_m128_g,
140                                           _mm_add_ps(_mm_add_ps(v_phi_bot_left, v_phi_top_left),
141                                                      _mm_add_ps(v_phi_bot_right, v_phi_top_right)));
142         __m128 third_line    = _mm_add_ps(_mm_mul_ps(_mm_set1_ps(0.25f),
143                                                      _mm_add_ps(_mm_add_ps(v_rho_bottom, v_rho_top),
144                                                                 _mm_add_ps(v_rho_left, v_rho_right))),
145                                           _mm_mul_ps(_mm_set1_ps(2.0f), v_rho_center));
146         __m128 result        = _mm_mul_ps(main_coef_m128_g,
147                                           _mm_add_ps(first_line, _mm_add_ps(second_line, third_line)));
148
149         _mm_storeu_ps(&phi_new[index], result);
150     }
151
152     for (int j = VECTORS_NUMBER_IN_LINE * VECTOR_SIZE_IN_FLOATS + 1; j < N_X - 1; j++) {
153         index = line_index + j;
154
155         phi_new[index] = MAIN_COEF * (FIRST_COEF * (phi[index - 1] + phi[index + 1]) +
156                                       SECOND_COEF * (phi[index - N_X] + phi[index + N_X]) +
157                                       THIRD_COEF * (phi[index - N_X - 1] + phi[index - N_X + 1] + phi[index + N_X - 1] +
    phi[index + N_X + 1]) +
158                                       2.0f * rho[index] +
159                                       0.25f * (rho[index - N_X] + rho[index + N_X] + rho[index - 1] + rho[index + 1]));
160     }
161 }
162
```

```
163 void swap_float_ptr(float** a, float** b) {
164     float* tmp = *a;
165     *a = *b;
166     *b = tmp;
167 }
168 
169 void fill_file_strip(int tid, int num_threads, FILE* fd, float* local_matrix, int local_rows) {
170     if (fd == NULL) {
171         printf("tid=%d: fd is closed", tid);
172         return;
173     }
174 
175     for (int i = 0; i < local_rows; i++) {
176         for (int j = 0; j < N_X; j++) {
177             fprintf(fd, "%f\t", local_matrix[i * N_X + j]);
178         }
179         fprintf(fd, "\n");
180     }
181 }
182 
183 void fill_file_pthread(const thread_data_t* data, FILE* fd, const int flag) {
184     if (fd == NULL) {
185         printf("tid=%d: fd is closed\n", data->thread_id);
186         return;
187     }
188 
189     for (int current_writer_id = 0; current_writer_id < data->num_threads; current_writer_id++) {
190         if (data->thread_id == current_writer_id) {
191             pthread_mutex_lock(&print_mutex);
192 
193             fill_file_strip(data->thread_id, data->num_threads, fd, (flag == 0) ? data->local_rho : data->local_phi,
    data->local_rows);
194 
195             pthread_mutex_unlock(&print_mutex);
```

```
196             }
197             pthread_barrier_wait(&task_g.barrier);
198         }
199     }
200
201     void init_rho_pthread(int tid, int num_threads, float* local_rho, int local_rows, int global_start_row_idx) {
202         const float R = 0.1f * fminf(X_B - X_A, Y_B - Y_A);
203         int y_coord;
204         for (int i = 0; i < local_rows; i++) {
205             y_coord = global_start_row_idx + i;
206             for (int j = 0; j < N_X; j++) {
207                 if ((X_A + j * H_X - X_s1) * (X_A + j * H_X - X_s1) + (Y_A + y_coord * H_Y - Y_s1) * (Y_A + y_coord * H_Y -
    Y_s1) < R * R) {
208                     local_rho[i * N_X + j] = 1.0f;
209                 }
210                 else if ((X_A + j * H_X - X_s2) * (X_A + j * H_X - X_s2)  + (Y_A + y_coord * H_Y - Y_s2) * (Y_A + y_coord *
    H_Y - Y_s2) < R * R) {
211                     local_rho[i * N_X + j] = -1.0f;
212                 }
213                 else {
214                     local_rho[i * N_X + j] = 0.0f;
215                 }
216             }
217         }
218     }
219
220     void* thread_worker_function(void* arg) {
221         thread_data_t* data = (thread_data_t*)arg;
222         // printf("tid[%d] = [%d %d]: local_rows=%d\tglobal_start_row_idx=%d\n", data->thread_id, getpid(), gettid(), data-
    >local_rows, data->global_start_row_idx);
223
224         set_cpu(data->thread_id);
225
226         for (int i = 0; i < data->local_rows; i++) {
```

```
227             for (int j = 0; j < N_X; j++) {
228                 data->local_phi[i * N_X + j] = 0.0f;
229                 data->local_phi_new[i * N_X + j] = 0.0f;
230             }
231         }
232 
233         init_rho_pthread(data->thread_id, data->num_threads, data->local_rho, data->local_rows, data->global_start_row_idx);
234         pthread_barrier_wait(&task_g.barrier);
235 
236         // fill_file_pthread(data, task_g.rho_output_file, 0);
237         // pthread_barrier_wait(&task_g.barrier);
238 
239         float local_delta, max_local_delta;
240         int tid = data->thread_id;
241         int num_threads = data->num_threads;
242         int iter_sync_generation;
243 
244         while (current_iteration_g < (N_T / LINE_NUMBER) && !stop_all_threads_g) {
245             // compute phi_new
246             iter_sync_generation = atomic_load_explicit(&task_g.current_sync_generation, memory_order_acquire);
247             // green step
248             {
249                 /*EXAMPLE
250                     LINE_NUMBER = 4
251                     i=1: k=1: u[1]
252                     i=2: k=1: u[2]
253                          k=2: v[1]
254                     i=3: k=1: u[3]
255                          k=2: v[2]
256                          k=3: u[1]
257                 */
258                 if (tid == 0) {
259                     for(int i = 1; i < LINE_NUMBER; i++) {
260                         for(int k = 1; k <= i; k++) {
```

```
261                         if (k % 2 == 1) { // u
262                             compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k + 1) * N_X);
263                         }
264                         else {            // v
265                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k + 1) * N_X);
266                         }
267                     }
268                 }
269             }
270 
271             /*EXAMPLE
272                 LINE_NUMBER = 4
273                 i=0: k=0: u[-1] , u[0]
274                 i=1: k=0: u[-2] , u[1]
275                      k=1: v[-1] , v[0]
276                 i=2: k=0: u[-3] , u[2]
277                      k=1: v[-2] , v[1]
278                      k=2: u`[-1], u`[0]
279             */
280             else {
281                 for(int i = 0; i < LINE_NUMBER - 1; i++) {
282                     for (int k = 0; k <= i; k++) {
283                         if (k % 2 == 0) { // u
284                             compute_line(data->local_phi_new, data->local_phi, data->local_rho, (-1 -(i - k)) * N_X);
285                             compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
286                         } else {          // v
287                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (-1 -(i - k)) * N_X);
288                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
289                         }
290                     }
291                 }
292             }
293         }
294 
```

```
295             // red step
296             {
297                 /*EXAMPLE
298                     i=4: k=0: u[4]
299                         k=1: v[3]
300                         k=2: u[2]
301                         k=3: v[1]
302                     ...
303                 */
304                 if (tid == 0) {
305                     for (int i = LINE_NUMBER; i < data->local_rows - (LINE_NUMBER - 1); i++) {
306                         for (int k = 0; k < LINE_NUMBER; k++) {
307                             if (k % 2 == 0) { // u
308                                 compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
309                             }
310                             else {            // v
311                                 compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
312                             }
313                         }
314                     }
315                 }
316                 /*EXAMPLE
317                     i=3: k=0: u[3]
318                         k=1: v[2]
319                         k=2: u[1]
320                         k=3: v[0]
321                 */
322                 else if (tid == num_threads - 1) {
323                     for (int i = LINE_NUMBER - 1; i < data->local_rows; i++) {
324                         for (int k = 0; k < LINE_NUMBER; k++) {
325                             if (k % 2 == 0) { // u
326                                 compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
327                             }
328                             else {            // v
```

```
329                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
330                         }
331                     }
332                 }
333             }
334             else {
335                 for (int i = LINE_NUMBER - 1; i < data->local_rows - (LINE_NUMBER - 1); i++) {
336                     for (int k = 0; k < LINE_NUMBER; k++) {
337                         if (k % 2 == 0) { // u
338                             compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k) * N_X);
339                         }
340                         else {            // v
341                             compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k) * N_X);
342                         }
343                     }
344                 }
345             }
346         }
347 
348         // --- Начало попарной синхронизации ---
349         atomic_store_explicit(&task_g.red_step_completed_flags[tid].val, iter_sync_generation, memory_order_release);
350 
351         // blue step
352         {
353             if (tid != num_threads - 1) {
354                 while(atomic_load_explicit(&task_g.red_step_completed_flags[tid + 1].val, memory_order_acquire) !=
    iter_sync_generation) {
355                     sched_yield();
356                 }
357             }
358             /*EXAMPLE
359                 k=0: i=63: v[62]
360                 k=1: i=63: u[61]
361                      i=64: u[62]
```

```
                k=2: i=63: v[60]
                     i=64: v[61]
                     i=65: v[62]
            */
            if (tid == num_threads - 1) {
                for (int k = 0; k < LINE_NUMBER - 1; k++) {
                    for(int i = data->local_rows - 1; i < data->local_rows + k; i++) {
                        if(k % 2 == 1) { // u
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k - 1) * N_X);
                        } else {         // v
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k - 1) * N_X);
                        }
                    }
                }
            }
            /*EXAMPLE
                k=0: i=13: v[12], v[13]
                k=1: i=13: u[11], u[13]
                    i=14: u[12], u[14]
                k=2: i=13: v[10], v[13]
                    i=14: v[11], v[14]
                    i=15: v[12], v[15]
            */
            else {
                for (int k = 0; k < LINE_NUMBER - 1; k++) {
                    for(int i = data->local_rows - (LINE_NUMBER - 1); i < data->local_rows - (LINE_NUMBER - 1) + k + 1;
    i++) {
                        if(k % 2 == 1) { // u
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i - k - 1) * N_X);
                            compute_line(data->local_phi_new, data->local_phi, data->local_rho, (i) * N_X);
                        } else {         // v
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i - k - 1) * N_X);
                            compute_line(data->local_phi, data->local_phi_new, data->local_rho, (i) * N_X);
                        }
```

```
                    }
                }
            }
        }
        // --- Конец попарной синхронизации ---

        // delta
        local_delta = compute_delta(data->local_phi, data->local_phi_new, data->local_rows);
        task_g.step_deltas[tid] = local_delta;
        // printf("tid=%d: local_delta=%f\n", tid, local_delta);
        pthread_barrier_wait(&task_g.barrier);
        if (tid == 0) {
            max_local_delta = task_g.step_deltas[0];
            for(int t = 1; t < data->num_threads; t++) {
                if ((task_g.step_deltas[t] - max_local_delta) > 0.000001) {
                    max_local_delta = task_g.step_deltas[t];
                }
            }

            if ((max_local_delta - global_delta) < 0.000001) {
                global_delta = max_local_delta;
                // printf("iter_number=%d\tglobal_delta = %f\n", current_iteration_g, global_delta);
            }
            else {
                printf("ERROR: DELTA IS GROWWING: global_delta = %f, iter_delta = %f", global_delta, max_local_delta);
                stop_all_threads_g = 1;
            }
        }

        // swap
        // swap_float_ptr(&(data->local_phi), &(data->local_phi_new));
        if(tid == 0) {
            current_iteration_g++;
            atomic_store_explicit(&task_g.current_sync_generation, iter_sync_generation + 1, memory_order_release);
```

```
429             }
430             pthread_barrier_wait(&task_g.barrier);
431         }
432
433         // pthread_barrier_wait(&task_g.barrier);
434         // fill_file_pthread(data, task_g.phi_output_file, 1);
435         return NULL;
436     }
437
438     void free_task(int num_threads) {
439         printf("[%d, %d]: free_task()\n", getpid(), gettid());
440
441         pthread_barrier_destroy(&task_g.barrier);
442
443         if (task_g.red_step_completed_flags != NULL) {
444             free(task_g.red_step_completed_flags);
445             task_g.red_step_completed_flags = NULL;
446         }
447
448         if (task_g.rho_output_file != NULL) {
449             fclose(task_g.rho_output_file);
450             task_g.rho_output_file = NULL;
451         }
452         if (task_g.phi_output_file != NULL) {
453             fclose(task_g.phi_output_file);
454             task_g.phi_output_file = NULL;
455         }
456         if (task_g.threads_args != NULL) {
457             free(task_g.threads_args);
458             task_g.threads_args = NULL;
459         }
460         if (task_g.step_deltas != NULL) {
461             free(task_g.step_deltas);
462             task_g.step_deltas = NULL;
```

```
463         }
464         if (task_g.rho_storage != NULL) {
465             free(task_g.rho_storage);
466             task_g.rho_storage = NULL;
467         }
468         if (task_g.phi_storage != NULL) {
469             free(task_g.phi_storage);
470             task_g.phi_storage = NULL;
471         }
472     }
473
474     int allocate_task(int num_threads) {
475         printf("[%d %d]: allocate_task()\n", getpid(), gettid());
476
477         task_g.phi_storage = (float*)malloc(2 * N_X * N_Y * sizeof(float));
478         task_g.rho_storage = (float*)malloc(N_X * N_Y * sizeof(float));
479         task_g.step_deltas = (float*)malloc(num_threads * sizeof(float));
480         task_g.threads_args = (thread_data_t*)malloc(num_threads * sizeof(thread_data_t));
481         task_g.rho_output_file = fopen("rho_pthread.txt", "wb");
482         task_g.phi_output_file = fopen("phi_pthread.txt", "wb");
483         int barrier_err = pthread_barrier_init(&task_g.barrier, NULL, num_threads);
484
485         task_g.red_step_completed_flags = (cache_padded_atomic_int_t*)malloc(num_threads *
    sizeof(cache_padded_atomic_int_t));
486         for (int i = 0; i < num_threads; ++i) {
487             atomic_init(&task_g.red_step_completed_flags[i].val, -1);
488         }
489         atomic_init(&task_g.current_sync_generation, 0);
490
491         if (task_g.phi_storage == NULL
492             || task_g.rho_storage == NULL
493             || task_g.step_deltas == NULL
494             || task_g.threads_args == NULL
495             || task_g.phi_output_file == NULL
```

```
496             || task_g.rho_output_file == NULL
497             || barrier_err != 0
498             || task_g.red_step_completed_flags == NULL
499         ) {
500             printf("[%d %d]: Failed to allocate program memory\n", getpid(), gettid());
501             free_task(num_threads);
502             return 1;
503         }
504
505         return 0;
506     }
507
508     int run_jacoby_method_pthread(int num_threads) {
509         current_iteration_g = 0;
510         global_delta = 1.0f;
511         stop_all_threads_g = 0;
512
513         if (N_Y - 2 <= 0) {
514             printf("main: Too small N_Y size\n");
515             return 1;
516         }
517         if (N_Y - 2 < num_threads) {
518             printf("main: Fewer computable rows (%d) than threads (%d). Some threads might not compute actual data rows.\n",
    N_Y - 2, num_threads);
519             num_threads = N_Y - 2;
520             printf("Now num_threads = %d", num_threads);
521         }
522
523         if (allocate_task(num_threads) != 0) {
524             return 1;
525         }
526
527         int base_rows_per_thread = N_Y / num_threads;
528         int remainder_rows = N_Y % num_threads;
```

```
529
530     int current_global_compute_row_start_idx = 0;
531     int memory_offset = 0;
532
533     for (int t = 0; t < num_threads; t++) {
534         task_g.threads_args[t].thread_id = t;
535         task_g.threads_args[t].num_threads = num_threads;
536
537         task_g.threads_args[t].local_rows = base_rows_per_thread + ((t < remainder_rows) ? 1 : 0);
538         task_g.threads_args[t].global_start_row_idx = current_global_compute_row_start_idx;
539         current_global_compute_row_start_idx += task_g.threads_args[t].local_rows;
540
541         task_g.threads_args[t].local_phi = task_g.phi_storage + memory_offset;
542         task_g.threads_args[t].local_phi_new = task_g.phi_storage + (N_X * N_Y) + memory_offset;
543         task_g.threads_args[t].local_rho = task_g.rho_storage + memory_offset;
544         memory_offset += task_g.threads_args[t].local_rows * N_X;
545     }
546
547     pthread_t threads[num_threads];
548
549     printf("main: Starting Jacobi method with pthreads. N_X=%d, N_Y=%d, N_T=%d, NUM_THREADS=%d\n", N_X, N_Y, N_T,
    num_threads);
550     long long t1, t2;
551     double tDiff;
552     struct timespec curTime;
553     clock_gettime(CLOCK_BOOTTIME, &curTime);
554     t1 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
555
556
557     for (int t = 0; t < num_threads; t++) {
558         if (pthread_create(&threads[t], NULL, thread_worker_function, &task_g.threads_args[t])) {
559             printf("main: Failed to create thread %d\n", t);
560             free_task(num_threads);
561             return 1;
```

```
562             }
563         }
564 
565         for (int t = 0; t < num_threads; t++) {
566             if (pthread_join(threads[t], NULL)) {
567                 printf("main: Failed to join thread %d\n", t);
568                 free_task(num_threads);
569                 return 1;
570             }
571         }
572 
573 
574         clock_gettime(CLOCK_BOOTTIME, &curTime);
575         t2 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
576         tDiff = (double) (t2 - t1) / 1000000000.0;
577         printf("main: Finising Jacobi method with pthreads. Time=%gs\n", tDiff);
578 
579         free_task(num_threads);
580         return 0;
581     }
582 
583     int main(int argc, char* argv[]) {
584             int pid = getpid();
585         int tid = gettid();
586         printf("main[%d %d] Hello\n", pid, tid);
587         if (argc != 2) {
588             printf("usage: %s <num_threads>\n", argv[0]);
589             return 1;
590         }
591 
592         int num_threads = atoi(argv[1]);
593         if (num_threads <= 0) {
594             printf("main[%d %d]: num_threads = %d, It must be > 0\n", pid, tid, num_threads);
595             return 1;
```

```
596         }
597         if (num_threads > 8) {
598             printf("main[%d %d]: num_threads = %d, It must be <= 8 (my num_kernels = 8)", pid, tid, num_threads);
599             return 1;
600         }
601 
602         main_coef_m128_g  = _mm_set1_ps(MAIN_COEF);
603         first_coef_m128_g = _mm_set1_ps(FIRST_COEF);
604         second_coef_m128_g= _mm_set1_ps(SECOND_COEF);
605         third_coef_m128_g = _mm_set1_ps(THIRD_COEF);
606 
607         pthread_mutex_init(&print_mutex, NULL);
608 
609         if (run_jacoby_method_pthread(num_threads)) {
610             printf("main[%d %d]: run_jacoby_method_pthread() failed\n", pid, tid);
611             pthread_mutex_destroy(&print_mutex);
612             return 1;
613         }
614 
615         pthread_mutex_destroy(&print_mutex);
616 
617         return 0;
618     }
```