УНИВЕРСИТЕТ ИТМО

Лабораторная работа №5 "Параллельное программирование с использованием стандарта POSIX Threads"

Дисциплина

Параллельные вычисления

Автор

Дмитрий Рачковский

12 января 2021 г.

Содержание

1 Введение

1.1 Цель работы

В данной работе необходимо исследовать эффективность распараллеливания программ на языке C с помощью технологии PThreads и сравнить её с технологией OpenMP. Для этого одна и та же программа, производящая достаточное количество вычислений с использованием канонических циклов for, компилируется сначала без использования OpenMP, затем с использованием распараллеливания циклов for технологией OpenMP, а затем - с использованием распараллеливания циклов for технологией PThreads. Время выполенения полученных выполняемых файлов сравнивается. Также производится контроль результата вычислений, который не должен меняться (в пределах допустимой погрешности) при распараллеливании, гарантируя правильность выполнения.

1.2 Использованное оборудование

Для проведения экспериментов использовалась виртуальная машина с OS Debian. Виртуальная машина получила доступ к 6 ядрам (используется 64-битный процессор Intel Core i7-8750H @ 2.20 GHz с 6 физическими и 12 логическими ядрами) и 6 ГБ оперативной памяти (из 16, доступных в системе). Во время выполнения экспериментов система была подключена к источнику питания.

Версия использованного компилятора: gcc (Debian 6.3.0-18+deb9u1) 6.3.0 20170516.

Версия OpenMP: 4.5 (201511)

Версия PThreads: 2.24

2 Ход работы

2.1 Используемая программа

Для проведения экспериментов программа, использованная в лабораторной работе №4, была переписана с использованием технологии PThreads вместо технологии OpenMP.

Полный текст программы может быть найден в приложениях 4.1 и 4.2.

2.2 Получение данных

Программа на OpenMP компилируется без подключения OpenMP для проверки прямой совместимости. Затем эта же программа компилируется с использованием флага -openmp. Программа, написанная с импользованием PThreads компилируется с флагом -pthread. Команды для компиляции программы могут быть найдены в приложении 4.3.

Для экспериментов были использованы значения N, найденные в предыдущей лабораторной работе: от 3350 до 36650 с шагом в 1110.

2.3 Анализ результатов

По выполнению первого эксперимента было замечено, что PThreads работает значительно быстрее OpenMP:

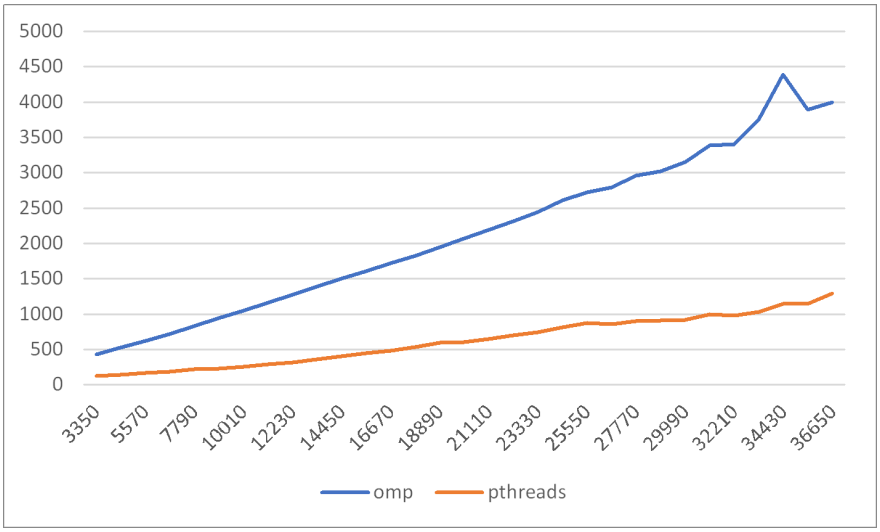

Рис. 1: Время выполнения программ на OpenMP и PThreads

Соответственно, параллельное ускорение программы, распараллеленной с помощью PThreads также растёт гораздо быстрее?

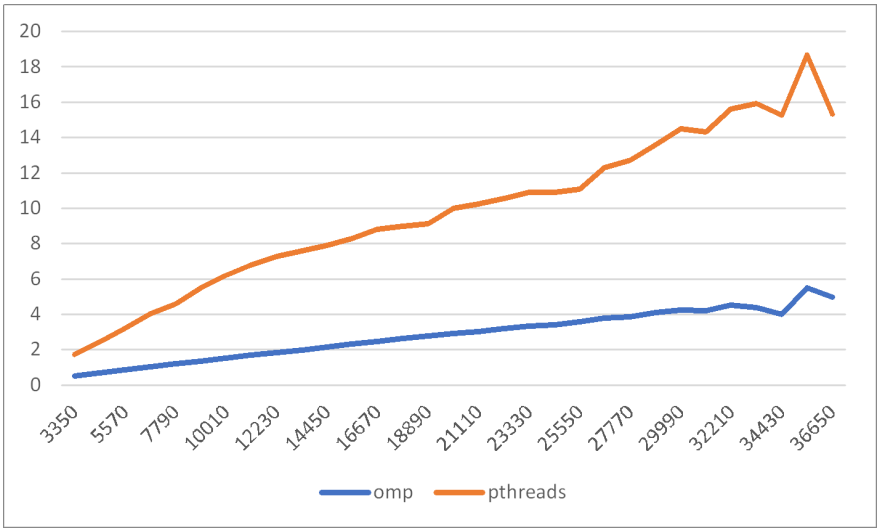

Рис. 2: Параллельное ускорение программ на OpenMP и PThreads

Помимо ускорения в работе по сравнению с OpenMP, значение N_x, при котором накладные расходы на распараллеливание прекращают превышать выигрыш от распараллеливания, также упало. При $N_x = 6700$ для OpenMP, для PThreads это значение равно $N_x = 2420$.

Другое наблюдение можно совершить, если рассмотреть долю времени, проводимого на каждом этапе вычисления.

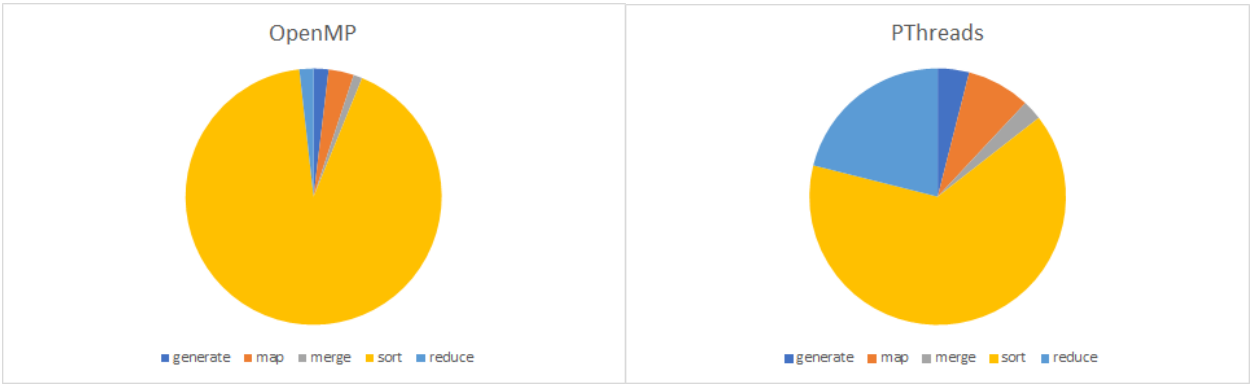

Рис. 3: Доля времени на каждом этапе вычисления

Можно заметить, что первые три этапа несильно занимают обе программы, а также что обе они тратят большую часть времени на этап Sort. Однако, программа, написанная на PThreads, тратит гораздо большее количество времени на этап reduce. Это также видно из сравнительных графиков:

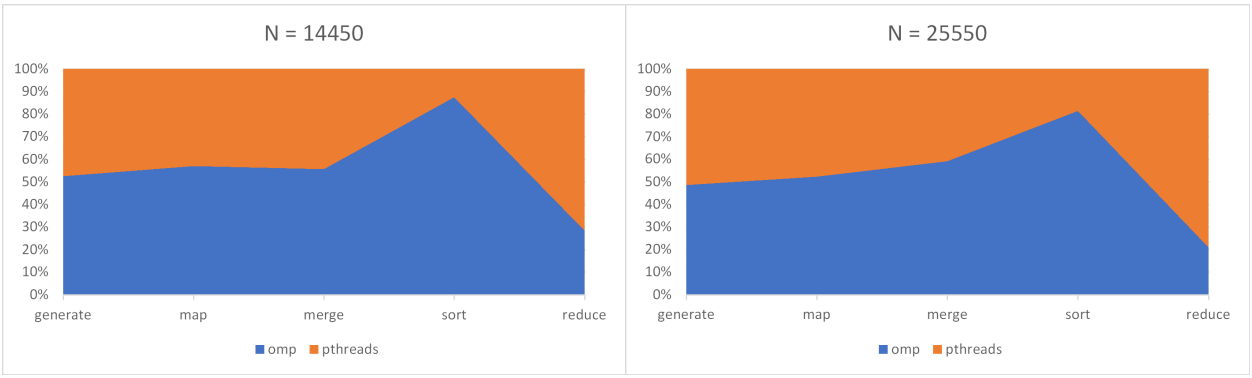

Рис. 4: Также доля времени на каждом этапе вычисления

Относительно друг друга, на первых трёх этапах программы тратят примерно равное количество времени:

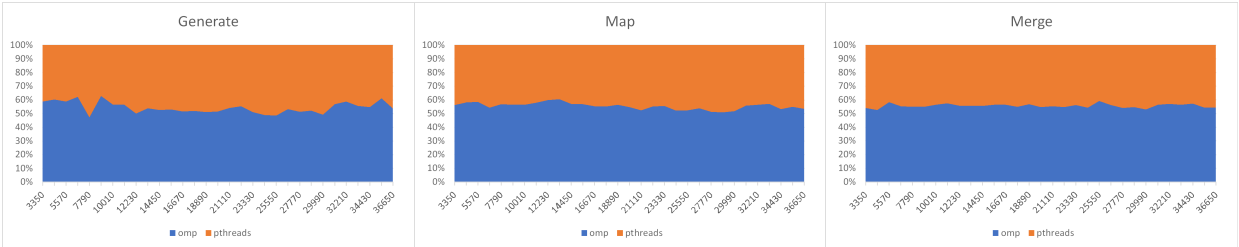

Рис. 5: Распределение времени на первых трёх этапах

На следующих же двух этапах распределение сильно отличается:

Рис. 6: Распределение времени на последних двух этапах

Можно наблюдать, что на этапе sort программа на OpenMP проводит сильно больше времени, в то время как на этапе reduce гораздо больше времени проводит программа на PThreads. Таким образом, распараллелив последний этап с помощью OpenMP, можно получить ещё большую выгоду.

3 Вывод

На основании данного эксперимента можно сделать несколько выводов:

- На изменение программы для того, чтобы она использовала стандарт POSIX Threads вместо OpenMP было потрачено примерно три часа, и хотя потребовалось большое количество изменений, было добавлено всего 75 строк кода.
- Параллельное ускорение при этом увеличивается в среднем в четыре раза, а минимальное время, при котором накладные расходы на распараллеливание прекращают превышать выигрыш от распараллеливания, упало почти в три раза. Это делает затраты времени на переделывание программы абсолютно стоящими.

4 Приложения

4.1 Исходный код программы на OpenMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <math.h>

#define A 700 //
#define NUMBER_OF_ITERATIONS 50

typedef enum stage_ {
    STAGE_UNDEFINED = 0,
    STAGE_GENERATE = 0,
    STAGE_MAP,
    STAGE_MERGE,
    STAGE_SORT,
    STAGE_REDUCE,
    STAGE_NUM_OF,
} stage_t;

int current_iteration = 0;
stage_t current_stage = STAGE_UNDEFINED;

void monitor_execution_percent();

#ifdef _OPENMP
    #include "omp.h"
    void monitor_execution_percent() {
        while (1) {
            sleep(1);
            double execution_percent = (100. / NUMBER_OF_ITERATIONS) * current_iteration + (100. / NUMBER_OF_ITERATIONS) * ((double)current_stage / STAGE_NUM_OF);
            if (execution_percent < 100) {
                printf("Current progress: %.2f%%\n", execution_percent);
            } else {
                break;
            }
        };
    }
#else
    #include <sys/time.h>
    double omp_get_wtime() { struct timeval T; gettimeofday(&T, NULL); return T.tv_sec + T.tv_usec / 1000000.; }
    int omp_get_num_procs() { return 1; }
    int omp_get_thread_num() { return 0; }
    void omp_set_nested(int n) {}
    void monitor_execution_percent() {}
#endif

long double random_on_interval(long double min, long double max, unsigned int *seed) {
    return (long double) ((rand_r(seed) % (int)(max + 1 - min)) + min);
}

unsigned int make_seed(int i, int j) {
    return 9572 + 234*i + 456*j;
}
```

```
54
55  int main(int argc, char* argv[]) {
56      int N = atoi(argv[1]);
57
58      omp_set_nested(1);
59
60      double T1, T2, last_time_measure;
61      double gen = 0;
62      double map = 0;
63      double mer = 0;
64      double sor = 0;
65      double red = 0;
66      long double M1[N];
67      long double M2[N/2];
68      long double M2_copy[N/2];
69
70      double X = 0;
71
72      int j = 0;
73      #pragma omp parallel sections
74      {
75          #pragma omp section
76          {
77              monitor_execution_percent();
78          }
79          #pragma omp section
80          {
81              T1 = omp_get_wtime();
82              last_time_measure = T1;
83              for (current_iteration=0; current_iteration<
                    NUMBER_OF_ITERATIONS; current_iteration++) {
84                  /******************** GENERATE ********************/
85                  current_stage = STAGE_GENERATE;
86                  unsigned int seed;
87
88                  #pragma omp parallel for default(none) private(j, seed)
                        shared(M1, N, current_iteration)
89                  for (j = 0; j < N; j++) {
90                      seed = make_seed(current_iteration, j);
91                      M1[j] = random_on_interval(0, A, &seed);
92                  }
93
94                  #pragma omp parallel for default(none) private(j, seed)
                        shared(M2, N, current_iteration)
95                  for (j = 0; j < N/2; j++) {
96                      seed = make_seed(current_iteration, j);
97                      M2[j] = random_on_interval(A, A*10, &seed);
98                  }
99                  // make a copy, shifting M2 one element to the right
100                 M2_copy[0] = 0;
101                 memcpy(&M2_copy[1], M2, sizeof(long double) * ((N/2)-1));
102
103                 gen += omp_get_wtime() - last_time_measure;
104                 last_time_measure = omp_get_wtime();
105                 /********************* MAP ************************/
106                 current_stage = STAGE_MAP;
107                 #pragma omp parallel for default(none) private(j) shared(
                        M1, N)
108                 for (j = 0; j < N; j++) {
109                     // operation #1, remember to convert to radians
110                     M1[j] = pow(sinhl((M1[j] * M_PI) / 180.0), 2);
111                 }
112
```

```
113                 #pragma omp parallel for default(none) private(j) shared(
                        M2, M2_copy, N)
114                 for (j = 0; j < N/2; j++) {
115                     // operation #3
116                     M2[j] = fabs(tanl(M2[j] + M2_copy[j]));
117                 }
118
119                 map += omp_get_wtime() - last_time_measure;
120                 last_time_measure = omp_get_wtime();
121                 /******************** MERGE ************************/
122                 current_stage = STAGE_MERGE;
123                 #pragma omp parallel for default(none) private(j) shared(
                        M1, M2, N)
124                 for (j = 0; j < N/2; j++) {
125                     // operation #1
126                     M2[j] = pow(M1[j], M2[j]);
127                 }
128
129                 mer += omp_get_wtime() - last_time_measure;
130                 last_time_measure = omp_get_wtime();
131                 /******************** SORT ************************/
132                 current_stage = STAGE_SORT;
133                 int elem_per_part = ((N/2) / omp_get_num_procs()) + 1;
134                 int start_locations[omp_get_num_procs()];
135                 #pragma omp parallel
136                 {
137                     int start_inc = elem_per_part * omp_get_thread_num();
138                     int finish_non_inc = (start_inc + elem_per_part) < N/2
                            ? start_inc + elem_per_part : N/2;
139                     start_locations[omp_get_thread_num()] = start_inc;
140                     for (int k = start_inc; k < finish_non_inc-1; k++)
141                     {
142                         int min_k = k;
143                         for (int l = k+1; l < finish_non_inc; l++)  {
144                             if (M2[l] < M2[min_k]) {
145                                 min_k = l;
146                             }
147                         }
148                         if (min_k != k) {
149                             long double temp = M2[k];
150                             M2[k] = M2[min_k];
151                             M2[min_k] = temp;
152                         }
153                     }
154                 }
155                 // we'll be reusing this variable
156                 memcpy(M2_copy, M2, sizeof(long double) * (N/2));
157                 for (j = 0; j < N/2; j++) {
158                     int min_interval = 0;
159                     for (int m = 1; m < omp_get_num_procs(); m++) {
160                         if ((start_locations[m] < N/2) && (M2_copy[
                                start_locations[m]] < M2_copy[start_locations[
                                min_interval]])) {
161                             min_interval = m;
162                         }
163                     }
164                     M2[j] = M2_copy[start_locations[min_interval]];
165                     M2_copy[start_locations[min_interval]] = INFINITY;
166                     start_locations[min_interval]++;
167                 }
168
169                 sor += omp_get_wtime() - last_time_measure;
170                 last_time_measure = omp_get_wtime();
```

```
171                 /******************* REDUCE *********************/
172                 current_stage = STAGE_REDUCE;
173                 int min_index;
174                 for (min_index = 0; M2[min_index] <= 0; min_index++) {}
175                 long double min = M2[min_index];
176
177                 #pragma omp parallel for default(none) private(j) shared(
                        M2, min, N, current_iteration) reduction(+:X)
178                 for (j = 0; j < N/2; j++) {
179                     if (isfinite(M2[j]) && (int)(M2[j] / min) % 2 == 0) {
180                         // remember to convert to radians
181                         X += sinl((M2[j] * M_PI) / 180.0);
182                     }
183                 }
184                 red += omp_get_wtime() - last_time_measure;
185                 last_time_measure = omp_get_wtime();
186                 current_stage = STAGE_UNDEFINED;
187             }
188             T2 = omp_get_wtime();
189         }
190     }
191
192     long delta_ms = (T2 - T1) * 1000;
193     printf("%ld\n%ld\n%ld\n%ld\n%ld\n", (long)(gen*1000), (long)(map*1000)
        , (long)(mer*1000), (long)(sor*1000), (long)(red*1000));
194     printf("%d\n", N);
195     printf("%ld\n", delta_ms);
196     printf("%.5f\n", X);
197
198     return 0;
199 }
```

4.2 Исходный код программы на PThreads

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>
#include <math.h>
#include <pthread.h>

#define A 700 //
#define NUMBER_OF_ITERATIONS 50
#define NUMBER_OF_THREADS 6

typedef enum stage_ {
    STAGE_UNDEFINED = 0,
    STAGE_GENERATE = 0,
    STAGE_MAP,
    STAGE_MERGE,
    STAGE_SORT,
    STAGE_REDUCE,
    STAGE_NUM_OF,
} stage_t;

typedef struct arg_struct_ {
    long start;
    long finish;
} arg_struct_t;

pthread_t comp_threads[NUMBER_OF_THREADS];
pthread_t progress_thread;
static pthread_mutex_t reduce_mutex = PTHREAD_MUTEX_INITIALIZER;

long double *M1;
long double *M2;
long double *M2_copy;

long double min;
double X = 0;

long current_iteration = 0;
stage_t current_stage = STAGE_UNDEFINED;

double get_time_ms() { struct timeval T; gettimeofday(&T, NULL); return T.
    tv_sec + T.tv_usec / 1000000.; }

void *monitor_execution_percent() {
    while (1) {
        sleep(1);
        double execution_percent = (100. / NUMBER_OF_ITERATIONS) *
            current_iteration + (100. / NUMBER_OF_ITERATIONS) * ((double)
            current_stage / STAGE_NUM_OF);
        if (execution_percent < 100) {
            printf("Current progress: %.2f%%\n", execution_percent);
        } else {
            break;
        }
    }
    pthread_exit(NULL);
}

long double random_on_interval(long double min, long double max, unsigned
    int *seed) {
```

```
58      return (long double) ((rand_r(seed) % (int)(max + 1 - min)) + min);
59  }
60
61  unsigned int make_seed(long i, long j) {
62      return 9572 + 234*i + 456*j;
63  }
64
65  void start_threads(void * (*func)(void *), long *segments) {
66      for (int j = 0; j < NUMBER_OF_THREADS; j++) {
67          arg_struct_t *args = malloc(sizeof(arg_struct_t));
68          args->start = segments[j];
69          args->finish = segments[j+1];
70          pthread_create(&comp_threads[j], NULL, func, (void *)args);
71      }
72  }
73
74  void wait_for_threads() {
75      for (int j = 0; j < NUMBER_OF_THREADS; j++) {
76          pthread_join(comp_threads[j], NULL);
77      }
78  }
79
80  void *fill_M1(void *arguments) {
81      arg_struct_t *args = arguments;
82      for (long j = args->start; j < args->finish; j++) {
83          unsigned int seed = make_seed(current_iteration, j);
84          M1[j] = random_on_interval(0, A, &seed);
85      }
86      free(args);
87      pthread_exit(NULL);
88  }
89
90  void *fill_M2(void *arguments) {
91      arg_struct_t *args = arguments;
92      for (long j = args->start; j < args->finish; j++) {
93          unsigned int seed = make_seed(current_iteration, j);
94          M2[j] = random_on_interval(A, A*10, &seed);
95      }
96      free(args);
97      pthread_exit(NULL);
98  }
99
100 void *map_M1(void *arguments) {
101     arg_struct_t *args = arguments;
102     for (long j = args->start; j < args->finish; j++) {
103         // operation #1, remember to convert to radians
104         M1[j] = pow(sinhl((M1[j] * M_PI) / 180.0), 2);
105     }
106     free(args);
107     pthread_exit(NULL);
108 }
109
110 void *map_M2(void *arguments) {
111     arg_struct_t *args = arguments;
112     for (long j = args->start; j < args->finish; j++) {
113         // operation #3
114         M2[j] = fabs(tanl(M2[j] + M2_copy[j]));
115     }
116     pthread_exit(NULL);
117 }
118
119 void *merge(void *arguments) {
120     arg_struct_t *args = arguments;
```

```
121     for (long j = args->start; j < args->finish; j++) {
122         // operation #1
123         M2[j] = pow(M1[j], M2[j]);
124     }
125     free(args);
126     pthread_exit(NULL);
127 }
128 
129 void *sort(void *arguments) {
130     arg_struct_t *args = arguments;
131     for (long j = args->start; j < args->finish; j++) {
132         int min_j = j;
133         for (int k = j+1; k < args->finish; k++)  {
134             if (M2[k] < M2[min_j]) {
135                 min_j = k;
136             }
137         }
138         if (min_j != j) {
139             long double temp = M2[j];
140             M2[j] = M2[min_j];
141             M2[min_j] = temp;
142         }
143     }
144     free(args);
145     pthread_exit(NULL);
146 }
147 
148 void *reduce(void *arguments) {
149     arg_struct_t *args = arguments;
150     for (long j = args->start; j < args->finish; j++) {
151         if (isfinite(M2[j]) && (int)(M2[j] / min) % 2 == 0) {
152             // remember to convert to radians
153             double value_to_add = sinl((M2[j] * M_PI) / 180.0);
154             pthread_mutex_lock(&reduce_mutex);
155             X += value_to_add;
156             pthread_mutex_unlock(&reduce_mutex);
157         }
158     }
159     free(args);
160     pthread_exit(NULL);
161 }
162 
163 int main(int argc, char* argv[]) {
164     double T1, T2, last_time_measure;
165     double gen = 0;
166     double map = 0;
167     double mer = 0;
168     double sor = 0;
169     double red = 0;
170     long N = atoi(argv[1]);
171 
172     M1 = malloc(sizeof(long double) * N);
173     M2 = malloc(sizeof(long double) * (N/2));
174     M2_copy = malloc(sizeof(long double) * (N/2));
175 
176     long M1_segments[NUMBER_OF_THREADS + 1];
177     long M2_segments[NUMBER_OF_THREADS + 1];
178     for (int i = 0; i < NUMBER_OF_THREADS; i++) {
179         M1_segments[i] = ((N / NUMBER_OF_THREADS) + 1)  * i;
180         M2_segments[i] = (((N/2) / NUMBER_OF_THREADS) + 1)  * i;
181     }
182     M1_segments[NUMBER_OF_THREADS] = N;
183     M2_segments[NUMBER_OF_THREADS] = N/2;
```

```
184
185     int j = 0;
186     pthread_create(&progress_thread, NULL, monitor_execution_percent, NULL
            );
187     T1 = get_time_ms();
188     last_time_measure = T1;
189     for (current_iteration=0; current_iteration<NUMBER_OF_ITERATIONS;
            current_iteration++) {
190         /******************** GENERATE ********************/
191         current_stage = STAGE_GENERATE;
192
193         start_threads(fill_M1, M1_segments);
194         wait_for_threads();
195
196         start_threads(fill_M2, M2_segments);
197         wait_for_threads();
198
199         // make a copy, shifting M2 one element to the right
200         M2_copy[0] = 0;
201         memcpy(&M2_copy[1], M2, sizeof(long double) * ((N/2)-1));
202
203         gen += get_time_ms() - last_time_measure;
204         last_time_measure = get_time_ms();
205         /******************** MAP ************************/
206         current_stage = STAGE_MAP;
207
208         start_threads(map_M1, M1_segments);
209         wait_for_threads();
210
211         start_threads(map_M2, M2_segments);
212         wait_for_threads();
213
214         map += get_time_ms() - last_time_measure;
215         last_time_measure = get_time_ms();
216         /******************** MERGE ***********************/
217         current_stage = STAGE_MERGE;
218
219         start_threads(merge, M2_segments);
220         wait_for_threads();
221
222         mer += get_time_ms() - last_time_measure;
223         last_time_measure = get_time_ms();
224         /******************** SORT ************************/
225         current_stage = STAGE_SORT;
226
227         start_threads(sort, M2_segments);
228         wait_for_threads();
229
230         // we'll be reusing this variable
231         memcpy(M2_copy, M2, sizeof(long double) * (N/2));
232         long start_locations[NUMBER_OF_THREADS];
233         memcpy(start_locations, M2_segments, sizeof(long) *
                NUMBER_OF_THREADS);
234         for (j = 0; j < N/2; j++) {
235             int min_interval = 0;
236             for (int m = 1; m < NUMBER_OF_THREADS; m++) {
237                 if ((start_locations[m] < N/2) && (M2_copy[start_locations
                        [m]] < M2_copy[start_locations[min_interval]])) {
238                     min_interval = m;
239                 }
240             }
241             M2[j] = M2_copy[start_locations[min_interval]];
242             M2_copy[start_locations[min_interval]] = INFINITY;
```

```
243             start_locations[min_interval]++;
244         }
245 
246         sor += get_time_ms() - last_time_measure;
247         last_time_measure = get_time_ms();
248         /******************** REDUCE ***********************/
249         current_stage = STAGE_REDUCE;
250         int min_index;
251         for (min_index = 0; M2[min_index] <= 0; min_index++) {}
252         min = M2[min_index];
253 
254         start_threads(reduce, M2_segments);
255         wait_for_threads();
256 
257         red += get_time_ms() - last_time_measure;
258         last_time_measure = get_time_ms();
259         current_stage = STAGE_UNDEFINED;
260     }
261     T2 = get_time_ms();
262 
263     free(M1);
264     free(M2);
265     free(M2_copy);
266     long delta_ms = (T2 - T1) * 1000;
267     printf("%ld\n%ld\n%ld\n%ld\n%ld\n", (long)(gen*1000), (long)(map*1000)
        , (long)(mer*1000), (long)(sor*1000), (long)(red*1000));
268     printf("%ld\n", N);
269     printf("%ld\n", delta_ms);
270     printf("%.5f\n", X);
271 
272     pthread_exit(NULL);
273 
274     return 0;
275 }
```

4.3 Скрипт для компиляции программ

```
gcc -O3 -Wall -lm -o lab4-seq lab4.c
gcc -O3 -Wall -Werror -fopenmp -lm -o lab4 lab4.c
gcc -O3 -Wall -Werror -pthread -lm -o lab5 lab5.c
```

4.4 Скрипт для запуска программм

```
#!/bin/bash
set -eu

declare -a N_set=("3350" "4460" "5570" "6680" "7790" "8900" "10010" "11120" "12230" "13340" "14450" "15560" "16670" "17780" "18890" "20000" "21110" "22220" "23330" "24440" "25550" "26660" "27770" "28880" "29990" "31100" "32210" "33320" "34430" "35540" "36650")
TEMPLATE="lab5_template.csv"
OUTPUT="lab5_output.csv"
N_LINE=1
SEQ_TIME_LINE_NUM_START=2
OMP_TIME_LINE_NUM_START=4
PTHREADS_TIME_LINE_NUM_START=11
SEQ_X_LINE_NUM=18
OMP_X_LINE_NUM=19
PTHREADS_X_LINE_NUM=20
EXECUTABLE_SEQ="lab4-seq"
EXECUTABLE_OMP="lab4"
EXECUTABLE_PTHREADS="lab5"

NUM_OF_TRIES=3

function add_to_line() {
  FILE_NAME="${1}"
  LINE_NUM="${2}"
  TEXT="${3}"
  sed -e "${LINE_NUM}s/\$/${TEXT};/" -i "${FILE_NAME}"
}

function execute() {
    EXECUTABLE="${1}"
    N="${2}"
    time_line_num="${3}"
    x_line_num="${4}"

    X=$(./${EXECUTABLE} ${N})
    y=(${X//$'\n'/ })
    min_time=${y[-2]}
    for (( meh=2; meh<=${NUM_OF_TRIES}; meh++ ))
    do
      X=$(./${EXECUTABLE} ${N})
      y=(${X//$'\n'/ })
      if [ "${y[-2]}" -lt "${min_time}" ]
      then
        min_time=${y[-2]}
      fi
    done
    add_to_line ${OUTPUT} ${time_line_num} ${min_time}
    if [[ "${EXECUTABLE}" != "${EXECUTABLE_SEQ}" ]]; then
      ((time_line_num=time_line_num+1))
      add_to_line ${OUTPUT} ${time_line_num} ${y[-8]}
      ((time_line_num=time_line_num+1))
```

```
50          add_to_line ${OUTPUT} ${time_line_num} ${y[-7]}
51          ((time_line_num=time_line_num+1))
52          add_to_line ${OUTPUT} ${time_line_num} ${y[-6]}
53          ((time_line_num=time_line_num+1))
54          add_to_line ${OUTPUT} ${time_line_num} ${y[-5]}
55          ((time_line_num=time_line_num+1))
56          add_to_line ${OUTPUT} ${time_line_num} ${y[-4]}
57      fi
58      add_to_line ${OUTPUT} ${x_line_num} $(sed 's/\./,/g' <<< ${y[-1]})
59  }
60
61  cp "./${TEMPLATE}" "./${OUTPUT}"
62
63  for N in "${N_set[@]}"
64  do
65      add_to_line ${OUTPUT} ${N_LINE} ${N}
66
67      execute ${EXECUTABLE_SEQ} ${N} ${SEQ_TIME_LINE_NUM_START} ${
            SEQ_X_LINE_NUM}
68      execute ${EXECUTABLE_OMP} ${N} ${OMP_TIME_LINE_NUM_START} ${
            OMP_X_LINE_NUM}
69      execute ${EXECUTABLE_PTHREADS} ${N} ${PTHREADS_TIME_LINE_NUM_START} ${
            PTHREADS_X_LINE_NUM}
70      echo ${N}
71  done
72
73  exit 0
```