Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

# Параллельные вычисления

Отчет по лабораторной работе

Создание параллельной программы на C++ с испольхованием Pthreads и OpenMP

**Работу выполнил:**
Раскин А.Р.
Группа: 13541/3
**Преподаватель:**
Стручков И.В.

Санкт-Петербург
2018

# Содержание

# 1. Цель работы

Изучить основы создания параллельных программ на C++ с использованием библиотек pthreads и OpenMP. Написать параллельную программу, которая решает следующую задачу: поиск площади окружностей, с использованием метода Монте-Карло. Сравнить производительность решений.

# 2. Программа работы

Для решения задачи создана часть программы, не зависящая от использования многопоточности и средств её реализации. На основе данной (методом линкования) собираются программы последовательного выполнения и многопоточного выполнения. Количество потоков задаётся аргументом командной строки для обеих многопоточных реализации.

Проверка эффективности выполнения проводится на компьютере с процессором описанным в листинге 2

```
lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  4
On-line CPU(s) list:     0-3
Thread(s) per core:      2
Core(s) per socket:      2
Socket(s):               1
NUMA node(s):            1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   42
Model name:              Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
Stepping:                7
CPU MHz:                 799.975
CPU max MHz:             3500.0000
CPU min MHz:             800.0000
BogoMIPS:                5584.55
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                4096K
NUMA node0 CPU(s):       0-3
Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
    ↪ cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
    ↪ rdtscp lm constant_tsc arch_perfmon pebs bts nopl xtopology nonstop_tsc
    ↪ cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
    ↪  cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes
    ↪ xsave avx lahf_lm epb tpr_shadow vnmi flexpriority ept vpid xsaveopt
    ↪ dtherm ida arat pln pts
```

# 3. Теоретическая информация

Метод Монте-Карло в данном случае заключается в генерации случайных точек в прямоугольном пространстве, включающем в себя окружности, площади которых мы ищем.

Подсчёт площади прямоугольника является простой задачей, а отношение точек, вошедших в окружности, ко всем точкам примерно равно отношению площадей окружностей к площади вышеупомянутого прямоугольника.

# 4. Ход выполнения работы

Для сокращения общего количества строк кода и использования одинаковых методов вычисления площади, общий набор функций был вынесен в файл 1

Листинг 1: monte.c

```
1  #include "monte.h"
2
3  double smaller(double current, double new){
4    if(new < current) return new;
5    return current;
6  }
7
8  double bigger(double current, double new){
9    if(new > current) return new;
10   return current;
11 }
12
13 Square getBorders(Circle circle[], int numOfCircles){
14   Square borders;
15
16   borders.topLeft.x = circle[0].center.x - circle[0].radius;
17   borders.topLeft.y = circle[0].center.y + circle[0].radius;
18   borders.botRight.x  = circle[0].center.x + circle[0].radius;
19   borders.botRight.y  = circle[0].center.y - circle[0].radius;
20
21   for (int i = 1; i < numOfCircles - 1 ; i++){
22     borders.topLeft.x = smaller(borders.topLeft.x,  circle[i].center.x - circle[
     ↪ i].radius);
23     borders.topLeft.y = bigger(borders.topLeft.y,   circle[i].center.y + circle[
     ↪ i].radius);
24     borders.botRight.x  = bigger(borders.botRight.x,  circle[i].center.x +
     ↪ circle[i].radius);
25     borders.botRight.y  = smaller(borders.botRight.y, circle[i].center.y -
     ↪ circle[i].radius);
26   }
27   return borders;
28 }
29
30 int isInside(Point point, Circle circle){
31   double xLength = point.x - circle.center.x;
32   double yLength = point.y - circle.center.y;
33   double Distance = sqrt(xLength*xLength + yLength*yLength);
34   if(Distance <= circle.radius) return 0;
35   return 1;
36 }
37
38 int countCircles(int args){
39   if( args == 0) return 0;
40   if( args % 3 == 0 ) return (args / 3);
41   return 0;
42 }
43
44 int parseCircles(Circle circle[], int numberOfCircles, char* argv[]){
```

```
45    for(int i = 0; i < numberOfCircles; i++){
46      circle[i].center.x  = atof(argv[3*i+1]);
47      circle[i].center.y  = atof(argv[3*i+2]);
48      circle[i].radius  = atof(argv[3*i+3]);
49    }
50    return 0;
51  }
52
53  Point getRandomPoint(Square borders, struct random_data *buf, int32_t *result){
54    Point point;
55    double random;
56    if(random_r(buf, result) != 0){
57      fprintf( stderr, "Error occurred in attempt to generate random x for a point
        ↪ \n");
58      point.x = (double)0;
59      point.y = (double)0;
60      return point;
61    }
62    random = *result;
63    point.x = borders.topLeft.x + (borders.botRight.x - borders.topLeft.x) *
        ↪ random / (double)(RAND_MAX);
64    if(random_r(buf, result) != 0){
65      fprintf( stderr, "Error occurred in attempt to generate random y for a point
        ↪ \n");
66      point.y = (double)0;
67      return point;
68    }
69    random = *result;
70    point.y = borders.botRight.y + (borders.topLeft.y - borders.botRight.y) *
        ↪ random / (double)(RAND_MAX);
71    return point;
72  }
73
74  int hitMonteCarlo(Square borders, Circle circles[], int numOfCircles, int tries,
        ↪  int *resultingHits){
75    int hits = 0;
76    Point randomPoint;
77    struct random_data buf;
78    memset(&buf, 0, sizeof(buf));
79    int32_t result=8;
80    char state[256];
81
82    initstate_r(*resultingHits, state, sizeof(state), &buf);
83    for( int i = 0; i < tries; i++){
84      for( int j = 0; j < numOfCircles; j++){
85        randomPoint = getRandomPoint(borders, &buf, &result);
86        if(isInside(randomPoint, circles[j]) == 0 ){
87          hits++;
88          break;
89        }
90      }
91    }
92    *resultingHits = hits;
93    return 0;
94  }
```

В фрагменте кода, приведённом в листинге 1 реализованы функции:

- поиска минимальных необходимых границ прямоугольника, охватывающего все заданные окружности.

- подсчёта количества заданных окружностей
- парсинга аргументов командной строки на предмет координат окружностей
- получение потоко-безопасным путем случайной точки
- основная логика метода Монте-Карло

## 4.1. Последовательное выполнение

Первым был реализован данный метод в меру наименьшей сложности, что позволило проверить корректность основного алгоритма подсчёта площади окружностей из пункта 4 и однопоточную производительность. Код реализации приведён в линстинге 2

Листинг 2: serial.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include "monte.h"


int main(int argc, char *argv[]){
  printf("Number of args: %d\n", argc);
  int numOfCircles = countCircles(argc-1); // set first argument aside (it's program name)
  printf("Number of circles: %d\n", numOfCircles);
  if(numOfCircles == 0){
    fprintf(stderr, "You should give at least one circle (X and Y of center and
     ↪ radius)\n");
    return 1;
  }

  Point point;
  Circle* circles;
  circles = malloc(numOfCircles * sizeof(Circle));
  if (circles == NULL){
    fprintf(stderr, "Error while allocating memory\n");
    exit(4);
  }
  parseCircles(circles, numOfCircles, argv);
  Square borders;

  borders = getBorders(circles, numOfCircles);
  printf("top-left corner: %f,%f\nbot-right corner: %f,%f\n",
    borders.topLeft.x,
    borders.topLeft.y,
    borders.botRight.x,
    borders.botRight.y);

  int iterations = 2000000000;
  int* circlesHits;
  circlesHits = malloc(1);
  if (circlesHits == NULL){
    fprintf(stderr, "Error while allocating memory\n");
    exit(4);
  }
  if(hitMonteCarlo(borders, circles, numOfCircles, iterations, circlesHits + 0 )
   ↪ ){
```

```
43      fprintf(stderr, "Error in function hitMonteCarlo\n");
44      return 2;
45    }
46
47
48    double ratio, circlesArea;
49    ratio = circlesHits[0] / (double)iterations;
50    circlesArea = ratio * (borders.botRight.x-borders.topLeft.x) * (borders.
        ↪ topLeft.y-borders.botRight.y);
51
52    printf("circles area is %f \n", circlesArea);
53
54    free(circles);
55    free(circlesHits);
56    return 0;
57  }
```

Данная программа принимает тройки координат (x,y,радиус) в качестве аргументов командной строки. И запускает генерацию случайной точки и проверку на её попадание 2000000000 раз подряд.

Результат запуска для одной единичной окружности приведён в листинге 4.2

```
1 $ ./serial 1 1 1
2 Number of args: 4
3 Number of circles: 1
4 top-left corner: 0.000000,2.000000
5 bot-right corner: 2.000000,0.000000
6 circles area is 3.141584
```

А время выполнения, засечённое утилитой командной строки **time**, составило 1 минуту 35 секунд.

## 4.2. Параллелизм на основе Pthreads

Принцип распараллеливания построен на деление общего количества "выстрелов"между потоками. После выполнения всех параллельных потоков их успешные "выстрелы"суммируются, а общее количество попыток известно, что и позволяет узнать соотношение. В качестве аргументов принимается количество потоков и тройки координат кругов. Код реализации приведён в листинге 3.

Листинг 3: pthreads.c

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4 #include <ctype.h>
5 #include <unistd.h>
6 #include <pthread.h>
7 #include "monte.h"
8
9 typedef struct {
10    Square *borders;
11    Circle *circles;
12    int *numOfCircles;
13    int iterations;
14    int *circlesHits;
15 }thread_args;
16
17 static void* thread_monte(void *arg){
```

```
18      thread_args * argsStruct = arg;
19
20      if(hitMonteCarlo(*(argsStruct->borders), argsStruct->circles, *(argsStruct->
        ↪ numOfCircles), argsStruct->iterations, argsStruct->circlesHits + 0 )){
21          fprintf(stderr, "Error in function hitMonteCarlo\n");
22      }
23      pthread_exit(NULL);
24
25  }
26
27  int main(int argc, char *argv[]){
28    printf("Number of args: %d\n", argc);
29    int numOfCircles = countCircles(argc-2); // set first argument aside (it's program name) +
        second - number of threads
30    printf("Number of circles: %d\n", numOfCircles);
31    if(numOfCircles == 0){
32      fprintf(stderr, "You should give at least one circle (X and Y of center and
        ↪ radius)\n");
33      return 1;
34    }
35
36    int numberOfThreads = atof(argv[1]);
37    if ( numberOfThreads == 0 ) exit(3);
38    printf("Number of threads: %d\n", numberOfThreads);
39
40    Point point;
41    Circle* circles;
42    circles = malloc(numOfCircles * sizeof(Circle));
43    if (circles == NULL){
44      fprintf(stderr, "Error while allocating memory\n");
45      exit(4);
46    }
47    parseCircles(circles, numOfCircles, &argv[1]);
48    Square borders;
49
50    borders = getBorders(circles, numOfCircles);
51    printf("top-left corner: %f,%f\nbot-right corner: %f,%f\n",
52      borders.topLeft.x,
53      borders.topLeft.y,
54      borders.botRight.x,
55      borders.botRight.y);
56
57    pthread_t* threadsArray;
58    threadsArray = malloc(numberOfThreads * sizeof(pthread_t));
59    if (threadsArray  == NULL){
60      fprintf(stderr, "Error while allocating memory\n");
61      exit(4);
62    }
63
64    int iterations = 2000000000;
65    int* circlesHits;
66    circlesHits = malloc(numberOfThreads * sizeof(int));
67    if (circlesHits == NULL){
68      fprintf(stderr, "Error while allocating memory\n");
69      exit(4);
70    }
71
72    for (int i=0; i < numberOfThreads; i++)
73      circlesHits[i]=i+1;
74
75    thread_args *ThreadArgs;
```

```
76    ThreadArgs= malloc(sizeof(thread_args) * numberOfThreads);
77    if (ThreadArgs == NULL){
78      fprintf(stderr, "Error while allocating memory\n");
79      exit(4);
80    }
81    for(int i = 0; i < numberOfThreads; i++){
82      ThreadArgs[i].circles = circles;
83      ThreadArgs[i].borders = &borders;
84      ThreadArgs[i].numOfCircles = &numOfCircles;
85      ThreadArgs[i].iterations = iterations / numberOfThreads;
86      ThreadArgs[i].circlesHits = &circlesHits[i];
87      pthread_create(threadsArray + i , NULL, *thread_monte, &ThreadArgs[i]);
88    }
89
90    int totalCirclesHits = 0;
91    for(int i = 0; i < numberOfThreads; i++){
92      int status = 0;
93      status = pthread_join(*(threadsArray + i ), NULL);
94      if( status != 0){
95        fprintf( stderr, "Failed to join thread %i with error %i", i, status);
96        return 3;
97      }
98      totalCirclesHits = totalCirclesHits + circlesHits[i];
99    }
100   free(threadsArray);
101   free(ThreadArgs);
102
103   double ratio, circlesArea;
104   ratio = totalCirclesHits / (double)iterations;
105   circlesArea = ratio * (borders.botRight.x-borders.topLeft.x) * (borders.
        ↪ topLeft.y-borders.botRight.y);
106
107   printf("circles area is %f \n", circlesArea);
108
109   free(circles);
110   free(circlesHits);
111   return 0;
112 }
```

Результат запуска для одной единичной окружности и двух потоков приведён в листинге 4.2

```
1 $ ./pthreads 2 1 1 1
2 Number of args: 5
3 Number of circles: 1
4 Number of threads: 2
5 top-left corner: 0.000000,2.000000
6 bot-right corner: 2.000000,0.000000
7 circles area is 3.141598
```

А время выполнения, засечённое утилитой командной строки **time**, составило:

- 1 минуту 35 секунд - один поток
- 51 секунда - два потока
- 37 секунд - четыре потока
- 36 секунд - восемь потоков
- 37 секунд - шестнадцать потоков

## 4.3. Параллелизм на основе OpenMP

Принцип распараллеливания идентичен приведённому в пункте 4.2. Количество потоков тоже регулируется через аргумент командной строки. Код реализации приведён в листинге 4.

Листинг 4: omp.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include <ctype.h>
5  #include <unistd.h>
6  #include <omp.h>
7  #include "monte.h"
8
9  typedef struct {
10   Square *borders;
11   Circle *circles;
12   int *numOfCircles;
13   int iterations;
14   int *circlesHits;
15 }thread_args;
16
17 static int thread_monte(void *arg){
18     thread_args * argsStruct = arg;
19
20     if(hitMonteCarlo(*(argsStruct->borders), argsStruct->circles, *(argsStruct->
       ↪ numOfCircles), argsStruct->iterations, argsStruct->circlesHits + 0 )){
21       fprintf(stderr, "Error in function hitMonteCarlo\n");
22       return 1;
23     }
24     return 0;
25 }
26
27 int main(int argc, char *argv[]){
28   printf("Number of args: %d\n", argc);
29   int numOfCircles = countCircles(argc-2); // set first argument aside (it's program name) +
       second - number of threads
30   printf("Number of circles: %d\n", numOfCircles);
31   if(numOfCircles == 0){
32     fprintf(stderr, "You should give at least one circle (X and Y of center and
       ↪ radius)\n");
33     return 1;
34   }
35
36   Point point;
37   Circle* circles;
38   circles = malloc(numOfCircles * sizeof(Circle));
39   if (circles == NULL){
40     fprintf(stderr, "Error while allocating memory\n");
41     exit(4);
42   }
43   parseCircles(circles, numOfCircles, &argv[1]);
44
45   int numberOfThreads = atof(argv[1]);
46   if ( numberOfThreads == 0 ) exit(3);
47   printf("Number of threads: %d\n", numberOfThreads);
48   Square borders;
49
50   borders = getBorders(circles, numOfCircles);
```

```
51    printf("top-left corner: %f,%f\nbot-right corner: %f,%f\n",
52      borders.topLeft.x,
53      borders.topLeft.y,
54      borders.botRight.x,
55      borders.botRight.y);
56
57    int tid;
58
59    int iterations = 2000000000;
60    int* circlesHits;
61    circlesHits = malloc(numberOfThreads);
62    if (circlesHits == NULL){
63      fprintf(stderr, "Error while allocating memory\n");
64      exit(4);
65    }
66
67    thread_args threadArgsTemplate;
68    threadArgsTemplate.circles = circles;
69    threadArgsTemplate.borders = &borders;
70    threadArgsTemplate.numOfCircles = &numOfCircles;
71    threadArgsTemplate.iterations = iterations / numberOfThreads;
72
73    for (int i=0; i < numberOfThreads; i++)
74      circlesHits[i]=i+1;
75
76    #pragma omp parallel num_threads(numberOfThreads) private(tid)
77    {
78      thread_args *ThreadArgs = malloc(sizeof(thread_args));
79      if (ThreadArgs == NULL){
80        fprintf(stderr, "Error while allocating memory\n");
81        exit(4);
82      }
83
84        ThreadArgs->circles = circles;
85        ThreadArgs->borders = &borders;
86        ThreadArgs->numOfCircles = &numOfCircles;
87        ThreadArgs->iterations = iterations / numberOfThreads;
88
89      ThreadArgs->circlesHits = &circlesHits[omp_get_thread_num()];
90      thread_monte(ThreadArgs);
91      #pragma omp barrier
92      free(ThreadArgs);
93    }
94
95    printf("THIS \n");
96    int totalCirclesHits = 0;
97    for(int i = 0; i < numberOfThreads; i++){
98      int status = 0;
99      printf("Hits for thread %d: %d\n", i, circlesHits[i]);
100     totalCirclesHits = totalCirclesHits + circlesHits[i];
101   }
102
103   double ratio, circlesArea;
104   ratio = totalCirclesHits / (double)iterations;
105   circlesArea = ratio * (borders.botRight.x-borders.topLeft.x) * (borders.
        ↪ topLeft.y-borders.botRight.y);
106
107   printf("circles area is %f \n", circlesArea);
108
109   free(circles);
```

```
110   return 0;
111 }
```

Результат запуска для одной единичной окружности и двух потоков приведён в листинге 4.3

```
1  $ ./omp 2 1 1 1
2  Number of args: 5
3  Number of circles: 1
4  Number of threads: 2
5  top-left corner: 0.000000,2.000000
6  bot-right corner: 2.000000,0.000000
7  THIS
8  Hits for thread 0: 785399606
9  Hits for thread 1: 785416430
10 circles area is 3.141632
```

А время выполнения, засечённое утилитой командной строки **time**, составило:

- 1 минуту 35 секунд - один поток
- 61 секунду- два потока
- 36 секунд- четыре потока
- 36 секунд- восемь потоков
- 36 секунд - шестнадцать потоков

# 5. Эксперименты

Каждая из реализации использует одинаковую кодовую базу и тестировалась на единичной окружности (позволяет легко проверить корректность счёта).

Листинг 5: Набор данных

```
0 0 1
```

Программа запускалась на наборе входных данных при $N_{total} = 2000000000$. Pthread и MPI версии запускались при этом в $1, 2, 4, 8$ потоков.

Листинг 6: Скрипт запуска

```
import sys
from subprocess import Popen, PIPE

# arguments
args = list(sys.argv)
if len(args) < 4:
sys.exit("Usage: python testparallel.py *programm* *input file* *num repeats*")

programm = args[1]
inputFile = args[2]
numRepeats = int(args[3])

#run program
allAreas = 0.0
for proc in [1, 2, 4, 8]:
times = []
for i in range(numRepeats):
process = Popen([programm, inputFile, '1000000', str(proc)], stdout=PIPE)
exit_code = process.wait()
if exit_code != 0:
sys.exit("Cannot run programm")

for line in process.stdout:
if 'Circles' in line:
allAreas = allAreas + float(line.split()[-1])
if 'Elapsed' in line:
times.append(float(line.split()[-1]))

av = sum(times) / numRepeats
disp = 0.0
for val in times:
disp = disp + (val - av) ** 2
if numRepeats == 1:
disp = disp / numRepeats
else:
disp = disp / (numRepeats - 1)

maxError = 2.58 * ((disp / numRepeats) ** (1.0 / 2.0))

print("{} threads: average = {}, dispersion = {}".format(proc, av, disp))
print("99% interval: {} +- {}".format(av, maxError))

print("Average area = {}".format(allAreas / (4 * numRepeats)))
```

Данный скрипт принимает путь к исполняемому файлу, путь к файлу с входными данными и число повторных запусков каждой программы. В результате он выводит оценки мат. ожидания и дисперсии времени работы программы для каждой конфигурации

запуска. Оценки мат. ожидания и дисперсии вычисляются по следующим формулам:

$$M = \frac{\sum_i x_i}{n}$$

$$D = \frac{1}{n-1}\sum_i (x_i - M)^2$$

По оценкам мат. ожидания и дисперсии вычисляется 99% доверительный интервал для времени работы программы. Доверительный интервал вычисляется по следующей формуле:

$$I = M \pm t_\alpha \sqrt{\frac{D}{n}}$$

В данной формуле $t_\alpha$ — это критерий Стьюдента для вероятности $\alpha$. При $\alpha = 0.99$, $t_\alpha = 2.58$.

Так же данный скрипт выводит вычисленную среднюю площадь фигуры.

Запуск программы был повторен 50 раз. Результаты экспериментов вы можете видеть в таблице 5.1.

Таблица 5.1

**Результаты**

| **Кол-во потоков** | **Однопоточная** | **PThread** | **MPI** |
|---:|---|---|---|
| **1** | $0.39714 \pm 0.0008$ | $0.3699 \pm 0.0027$ | $0.3719 \pm 0.0008$ |
| **2** | — | $0.1912 \pm 0.0018$ | $0.2025 \pm 0.0013$ |
| **4** | — | $\mathbf{0.1002 \pm 0.0017}$ | $0.1246 \pm 0.0024$ |
| **8** | — | $0.1067 \pm 0.0011$ | $0.1727 \pm 0.0052$ |

Проблемные места данного тестирования:

1. Короткие тесты
2. Одинарный запуск каждого из тестов

Первая проблема делает разницу между тестами менее заметной, и увеличивает влияние сторонних факторов (например, конкуренцию за процессорное время с другими процессами). И хоть проблема решаемая увеличением количества "выстрелов и при имеющихся результатах видно, что OpenMP справляется с задачей эффективнее, чем Pthreads, а распараллеливание позволяет достичь ощутимый прирост производительности в хорошо распаралеливаемых алгоритмах.

Вторая проблема остаётся не решённой в рамках данной работы, однако система, на которой проводились замеры, обладает малым количеством фоновых процессов и не была со стороны пользователя загружена другими процессами.

Спортивного интереса ради, каждая из приведённых программ была запущена на большем наборе данных с несколькими кругами:

Листинг 7: Набор данных 2

```
1 0  0  10
2 18  16  100
3 5  0.1  0.1
```

Результаты приведены в таблице 5.2.

Таблица 5.2

**Результаты на втором наборе данных**

| **Кол-во потоков** | **Однопоточная** | **PThread** | **MPI** |
|---:|---|---|---|
| **1** | $0.4002 \pm 0.0018$ | $0.3906 \pm 0.0029$ | $0.4139 \pm 0.0036$ |
| **2** | — | $0.2021 \pm 0.0016$ | $0.2276 \pm 0.0041$ |
| **4** | — | $\mathbf{0.1097 \pm 0.0022}$ | $0.1329 \pm 0.0031$ |
| **8** | — | $0.1143 \pm 0.0017$ | $0.1816 \pm 0.0051$ |

# 6. Выводы

Решаемая задача имела минимум проблем с рапараллеливанием - почти полное отсутствие конфликтов по данным. Что не позволило более детально взглянуть на гибкость рассматриваемых библиотек: Pthreads и OpenMP.

Однако, во уже время изучения библиотек для решения данной проблемы было понятно, что OpenMP - более гибкий, настраиваемый инструмент, нежели Pthreads, так как изначально(без необходимости самостоятельной реализации) имеет более богатый инструментарий.

Тесты показали, что в данной задаче OpenMP справляется с задачей быстрее, чем Pthreads. А распараллеленые решения могут помочь достичь почти 3х кратного ускорения для данной задачи на конкретном компьютере.