

Санкт-Петербургский политехнический университет Петра Великого  
Институт компьютерных наук и технологий  
Кафедра компьютерных систем и программных технологий

## Отчет по курсовой работе

по дисциплине «Параллельные вычисления»

тема работы: «Определение площади набора кругов, заданных массивом с координатами центров и радиусами, методом Монте-Карло.»

**Работу выполнил:**

Косолапов С.А.

Группа: 53501/3

**Преподаватель:**

Стручков И.В.

Санкт-Петербург  
2016

# Содержание

<b>1</b>	<b>Постановка задачи</b>	<b>2</b>
<b>2</b>	<b>Реализация</b>	<b>2</b>
2.1	Реализация последовательной программы . . . . .	2
2.2	Реализация параллельной программы с использованием PThreads . . . . .	5
2.3	Реализация параллельной программы с использованием MPI . . . . .	6
<b>3</b>	<b>Тестирование производительности многозадачных программ</b>	<b>7</b>
<b>4</b>	<b>Анализ результатов</b>	<b>10</b>
<b>5</b>	<b>Выводы</b>	<b>10</b>
<b>6</b>	<b>Листинги</b>	<b>10</b>

# 1 Постановка задачи

1. Реализовать последовательную программу, позволяющую определить суммарную площадь набора кругов, заданных массивом с координатами центров и радиусами, методом Монте-Карло.
2. Провести тестирование последовательной программы.
3. Реализовать параллельные программы с использованием PThreads и MPI.
4. Провести тестирование производительности параллельных программ в зависимости от количества используемых ядер (процессоров).
5. Проанализировать полученные результаты и сделать выводы.

# 2 Реализация

Поставленная задача была декомпозирована на следующие этапы:

1. Загрузка координат центра круга и его радиуса из файла. Результатом является контейнер объектов, содержащих координаты и радиус кругов.
2. Определение площади с использованием метода Монте-Карло. Определяются граничные точки заданных кругов, образующие прямоугольную область. В соответствии с заданной плотностью случайным образом сгенерированных точек внутри прямоугольной области, а также её площадью, устанавливается их количество. Каждая точка проверяется на принадлежность хотя бы одному из кругов. Таким образом, предельное отношение точек, принадлежащих хотя бы одному из кругов, к общему количеству сгенерированных точек, примерно равняется суммарной площади кругов.
3. Вывод результата и статистических показателей, таких как математическое ожидание результата и его дисперсия.

## 2.1 Реализация последовательной программы

В соответствии с решаемыми задачами, исходный код программы содержит в себе следующие файлы:

1. types.h - содержит структуры и переопределённые типы, используемые в программе

```
1 typedef double T;
2
3 class Point
4 {
5 public:
6     T x;
7     T y;
8
9     Point(T x, T y)
10    {
11        this->x = x;
12        this->y = y;
13    }
14
15    Point(const Point *point)
16    {
17        this->x = point->x;
18        this->y = point->y;
19    }
20 };
21
22 class Circle
23 {
24 private:
25     inline static T sqr(T x){ return x*x; }
26 public:
27     Point *center;
28     T radius;
29
30     Circle(Point *center, T radius)
31     {
32         this->center = center;
33         this->radius = radius;
```

```

34     }
35
36     ~Circle()
37     {
38         delete center;
39     }
40
41     bool containsPoint(const Point *point)
42     {
43         return this->sqr(this->center->x - point->x) +
44                this->sqr(this->center->y - point->y) <= this->sqr(this->radius);
45     }
46 };
47
48
49 class Rect
50 {
51 public:
52     Point *tl;
53     Point *br;
54
55     Rect(Point *tl, Point *br)
56     {
57         this->tl = tl;
58         this->br = br;
59     }
60
61     Rect(T x0, T y0, T x1, T y1)
62     {
63         this->tl = new Point(x0, y0);
64         this->br = new Point(x1, y1);
65     }
66
67     ~Rect()
68     {
69         delete tl;
70         delete br;
71     }
72
73     T area() const
74     {
75         return std::fabs((tl->x - br->x)*(tl->y - br->y));
76     }
77 };
78
79 typedef Point* (*RandomFunction)(const Rect*, int);

```

В файле содержатся структуры Point, Circle и Rect, а также определён тип RandomFunction, позволяющий использовать различные варианты генерации точек.

2. parser.h и parser.cpp - содержат в себе объявление и реализацию функции, позволяющей считать вектор объектов типа Circle из файла

```

1 std::vector< Circle* > *parseCirclesFromFile(const std::string &filename, int max_circles
    ↪ ) throw(FileException);

```

3. solver.h и solver.cpp - содержат функциональность для решения задачи

```

1 T solve(const std::vector< Circle* > *circles, int density, RandomFunction random);
2 std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
3     const std::vector< Circle* > *circles, const Rect *rect, int density,
    ↪ RandomFunction random);
4 Rect *getFigureRect(const std::vector< Circle* > *circles);
5 bool isPointInsideCircles(const std::vector< Circle* > *circles,

```

Функция solve позволяет решить задачу для кругов, указанных в параметре circles. Также указываются параметры density - количество точек на единицу площади (плотность генерации) и random - функция для генерации точек со случайными координатами. Сама функция solve вызывает функцию getFigureRect, которая позволяет определить границы прямоугольной области, в которой находятся круги, а затем - функцию generateAndCheckPoints, возвращающую общее количество точек и количество точек, входящих в круги.

Функция getFigureRect:

```

1 Rect *getFigureRect(const std::vector< Circle* > *circles)
2 {
3     typename std::vector< Circle* >::const_iterator it = circles->begin();
4
5     Point *min = new Point(
6         (*it)->center->x - (*it)->radius,
7         (*it)->center->y - (*it)->radius
8     );
9     Point *max = new Point(
10        (*it)->center->x + (*it)->radius,
11        (*it)->center->y + (*it)->radius
12    );
13
14    while(++it != circles->end())
15    {
16        if((*it)->center->x - (*it)->radius < min->x)
17        {
18            min->x = (*it)->center->x - (*it)->radius;
19        }
20        else if((*it)->center->x + (*it)->radius > max->x)
21        {
22            max->x = (*it)->center->x + (*it)->radius;
23        }
24
25        if((*it)->center->y - (*it)->radius < min->y)
26        {
27            min->y = (*it)->center->y - (*it)->radius;
28        }
29        else if((*it)->center->y + (*it)->radius > max->y)
30        {
31            max->y = (*it)->center->y + (*it)->radius;
32        }
33    }
34    return new Rect(min, max);
35 }

```

В функции generateAndCheckPoints производится непосредственное решение задачи. Изначально определяется количество точек для генерации:

```

1 T area = rect->area();
2 unsigned long count = density*static_cast<unsigned long>(area);
3 unsigned long result = 0;

```

Затем с помощью функции генерации random, передаваемой как параметр функции, генерируются точки и проверяются на принадлежность кругам.

```

1 for(unsigned long i = 0; i < count; ++i)
2 {
3     Point *point = random(rect, density);
4     if (isPointInsideCircles(circles, point)) {
5         ++result;
6     }
7     delete point;
8 }

```

Функция isPointInsideCircles позволяет проверить, находится ли точка внутри окружностей:

```

1 bool isPointInsideCircles(const std::vector< Circle* > *circles, const Point *point)
2 {
3     for(typename std::vector< Circle* >::const_iterator it = circles->begin(); it !=
4     ↪ circles->end(); it++)
5     {
6         if((*it)->containsPoint(point))
7         {
8             return true;
9         }
10    }
11    return false;

```

4. randomizer.h и randomizer.cpp - содержат функции для генерации точек со случайными координатами.

Рассмотрено два варианта реализации функций генерации точек.

- (a) `random_simple` - позволяет решить задачу с помощью функции `rand()` и последующего масштабирования на заданную прямоугольную область. У данной реализации имеются две основные проблемы - распределение отлично от равномерного, что создаёт дополнительную погрешность, а также невозможность одновременного использования функции несколькими потоками, что затрудняет использование данной функции при работе с PThreads и OpenMP.
- (b) `random_uniform_real_distribution` - позволяет решить задачу с использованием функции, генерирующей равномерное распределение (`std::uniform_real_distribution`). Этот вариант хорошо работает в параллельных приложениях, а также позволяет получить действительно равномерное распределение. Основной проблемой является достаточно медленное выполнение функции.

Таким образом, после первоначального тестирования и выявления недостатков функции `random_simple` было решено в дальнейшем использовать `random_uniform_real_distribution`.

5. `main.cpp` соединяет воедино функциональность программы, содержит тесты.

## 2.2 Реализация параллельной программы с использованием PThreads

В данном случае дополнительно создаются  $N-1$  потоков, если считать  $N$  количеством ядер. Количество генерируемых точек `COUNT` делится на  $N$  частей, и созданные потоки генерируют и проверяют принадлежность окружностям для `COUNT/N` точек. Оставшиеся `COUNT - (N-1)*COUNT/N` точек генерируются и проверяются главным потоком. Затем главный поток выполняет `pthread_join` для всех созданных потоков и, когда дождётся, может сложить полученные всеми потоками результаты. Результаты хранятся в массиве, адрес элементов которого передаётся каждому потоку.

Так как в `pthread_create` тип передаваемых параметров `void*`, в функцию передаётся указатель на следующую структуру, позволяющую потоку использовать необходимые значения:

```

1 struct PointGeneratorStruct{
2     unsigned long count;
3     int density;
4     const Rect *rect;
5     RandomFunction random;
6     const std::vector< Circle* > *circles;
7     unsigned long result;
8
9     PointGeneratorStruct(unsigned long count,
10                          int density,
11                          const Rect *rect,
12                          RandomFunction random,
13                          const std::vector< Circle* > *circles)
14     {
15         this->count = count;
16         this->density = density;
17         this->rect = rect;
18         this->random = random;
19         this->circles = circles;
20     }
21 };

```

Функция, реализующая функциональность одного потока:

```

1 void *generatePointsThread(void *params)
2 {
3     PointGeneratorStruct *data = (PointGeneratorStruct*)params;
4     data->result = 0;
5     for(unsigned long i = 0; i < data->count; ++i)
6     {
7         Point* point = data->random(data->rect, data->density);
8         if(isPointInsideCircles(data->circles, point))
9         {
10             ++(data->result);
11         }
12         delete point;
13     }
14     return NULL;

```

Таким образом, для каждого потока генерируется и проверяется на принадлежность кругам заданное количество точек.

В функции `generateAndCheckPoints` сначала создаются потоки:

```

1 PointGeneratorStruct *params[max_cores];
2
3 pthread_t threads[max_cores - 1];
4 for(int i = 0; i < max_cores - 1; ++i)
5 {
6     params[i] = new PointGeneratorStruct(count/max_cores, density, rect, random, circles);
7     pthread_create(
8         &threads[i],
9         NULL,
10        generatePointsThread,
11        params[i]
12    );
13 }

```

Затем производятся вычисления на главном потоке:

```

1 unsigned long real_count = count - count*(max_cores - 1)/max_cores;
2 params[max_cores - 1] =
3     new PointGeneratorStruct(real_count + real_count % max_cores, density, rect,
4     ↪ random, circles);
5 generatePointsThread(params[max_cores - 1]);

```

После этого главный поток ожидает созданные потоки, либо удостоверится в их завершении:

```

1 for(int i = 0; i < max_cores - 1; ++i)
2 {
3     pthread_join(threads[i], NULL);
4 }

```

Сделав это, главный поток проводит редукцию результатов:

```

1 unsigned long result = params[0]->result;
2
3 for(int i = 1; i < max_cores; ++i)
4 {
5     result += params[i]->result;
6     delete params[i];
7 }

```

Полученное значение делится на общее количество точек, и получается суммарная площадь, занимаемая кругами.

В данном случае удалось избежать совместного использования потоками ресурсов для записи, а следовательно, и использования средств синхронизации.

## 2.3 Реализация параллельной программы с использованием MPI

В данном случае создаются не потоки, а процессы. Запуск программы в этом случае должен осуществляться с помощью программы `mpirun`. Использована реализация OpenMPI.

Это создаёт дополнительные шаги и при использовании MPI в программе. В частности, необходимо произвести инициализацию и финализацию с помощью функций `MPI_Init` и `MPI_Finalize`. Причём в `MPI_Init` необходимо также передать аргументы командной строки, поэтому её использование вне функции `main` затруднено в связи с необходимостью передачи дополнительных параметров.

MPI использует понятия группы и коммунитатора. Группа процессов - это упорядоченная коллекция процессов. Коммунитатор же позволяет общаться процессам внутри группы или между группами.

Также в программе используется функция `MPI_Comm_size`, позволяющая получить размер коммунитатора, а также стандартный коммунитатор `MPI_COMM_WORLD`, содержащий все созданные процессы.

С помощью функции `MPI_Comm_rank` можно узнать номер процесса в текущем коммунитаторе.

Функция `generateAndCheckPoints` видоизменилась, и теперь её часть, решающая задачу, выглядит следующим образом:

```

1 unsigned long result;
2
3 // ——— MPI section ———
4
5 int mpi_rank;
6 const int rank_main = 0;
7
8 MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
9
10 if(mpi_rank == rank_main)

```

```

11 {
12     unsigned long real_count = count - count*(max_cores - 1)/max_cores;
13     result = generatePointsProcess(circles, real_count, density, rect, random);
14 }
15 else
16 {
17     result = generatePointsProcess(circles, count/max_cores, density, rect, random);
18     sendResult(result, rank_main);
19     return NULL;
20 }
21
22 for(int rank = 1; rank < max_cores; ++rank)
23 {
24     unsigned long current_result;
25     recvResult(&current_result, rank);
26     result += current_result;
27 }
28
29 // — /MPI —

```

Таким образом, в зависимости от ранга процесса, мы определяем, главный это процесс или нет. Он должен ожидать результат от остальных процессов и выполнить редукцию.

Передача и получение результата выделены в отдельные функции:

```

1 void sendResult(unsigned long result, int to)
2 {
3     MPI_Send(&result, 1, MPI_UNSIGNED_LONG, to, 3, MPI_COMM_WORLD);
4 }
5
6 void recvResult(unsigned long *result, int from)
7 {
8     MPI_Status status;
9     MPI_Recv(result, 1, MPI_UNSIGNED_LONG, from, 3, MPI_COMM_WORLD, &status);
10 }

```

Непосредственное решение задачи также вынесено в отдельную функцию:

```

1 unsigned long generatePointsProcess(const std::vector< Circle* > *circles,
2                                     unsigned long count, int density, const Rect rect, RandomFunction random)
3 {
4     unsigned long result = 0;
5     for(unsigned long i = 0; i < count; ++i)
6     {
7         Point* point = random(rect, density);
8         if(isPointInsideCircles(circles, point))
9         {
10             ++result;
11         }
12         delete point;
13     }
14     return result;
15 }

```

### 3 Тестирование производительности многозадачных программ

Для обеих многозадачных модификаций программ были проведены тестовые испытания, состоящие из 50 тестов для различного количества ядер от 1 до 6.

#### 1. Многопоточная программа с использованием POSIX Threads

Результат, в зависимости от количества задействованных потоков:

##### • 1 поток

```

1 [Result]
2 Mean value = 31416.11368
3 Dispersion = 11.07398086
4
5 [Time]
6 Mean value = 14.21
7 Dispersion = 0.015213

```

##### • 2 потока



```
1 [Result]
2 Mean value = 31413.79668
3 Dispersion = 12.39041898
4
5 [Time]
6 Mean value = 11.11
7 Dispersion = 0.05943
```

• 3 потока

```
1 [Result]
2 Mean value = 31414.81872
3 Dispersion = 13.91885692
4
5 [Time]
6 Mean value = 8.9818
7 Dispersion = 0.54635
```

• 4 потока

```
1 [Result]
2 Mean value = 31413.73856
3 Dispersion = 12.62161521
4
5 [Time]
6 Mean value = 8.1528
7 Dispersion = 0.10264
```

• 5 потоков

```
1 [Result]
2 Mean value = 31413.39316
3 Dispersion = 17.33276977
4
5 [Time]
6 Mean value = 7.408
7 Dispersion = 0.20057
```

• 6 потоков

```
1 [Result]
2 Mean value = 31413.01008
3 Dispersion = 14.11241007
4
5 [Time]
6 Mean value = 7.2377
7 Dispersion = 0.076595
```

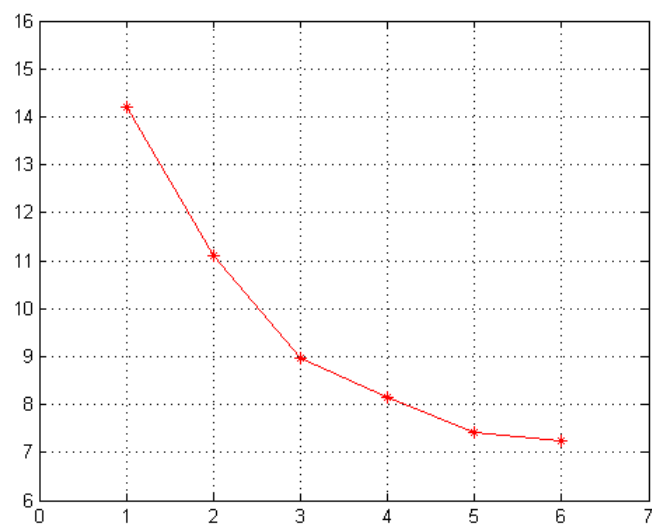


Рис. 1: График зависимости времени выполнения программы с PThreads от количества ядер

В данном случае с увеличением количества ядер наблюдается лишь незначительное увеличение производительности. Так, при использовании 6 ядер программа выполняется быстрее всего в 2 раза.

## 2. Многопроцессная программа с использованием MPI

Тестирование программы осуществляется с помощью shell-скрипта runtest.sh:

```
1 #!/bin/sh
2
3 max_cores=6
4
5 for i in `seq 1 $max_cores`
6 do
7   mpirun -np $i ./build/dist/CirclesFigureArea
8 done
```

Результат, в зависимости от количества задействованных процессов:

### • 1 процесс

```
1 [Result]
2 Mean value = 31416
3 Dispersion = 14.248
4
5 [Time]
6 Mean value = 14.485
7 Dispersion = 0.075995
```

### • 2 процесса

```
1 [Result]
2 Mean value = 31416
3 Dispersion = 12.289
4
5 [Time]
6 Mean value = 7.2112
7 Dispersion = 0.00044222
```

### • 3 процесса

```
1 [Result]
2 Mean value = 31416
3 Dispersion = 10.529
4
5 [Time]
6 Mean value = 4.8756
7 Dispersion = 0.0049912
```

### • 4 процесса

```
1 [Result]
2 Mean value = 31416
3 Dispersion = 16.05
4
5 [Time]
6 Mean value = 3.8008
7 Dispersion = 0.0063037
```

### • 5 процессов

```
1 [Result]
2 Mean value = 31416
3 Dispersion = 12.64
4
5 [Time]
6 Mean value = 3.377
7 Dispersion = 0.0082743
```

### • 6 процессов

```
1 [Result]
2 Mean value = 31416
3 Dispersion = 8.79
4
5 [Time]
6 Mean value = 2.8819
7 Dispersion = 0.044522
```

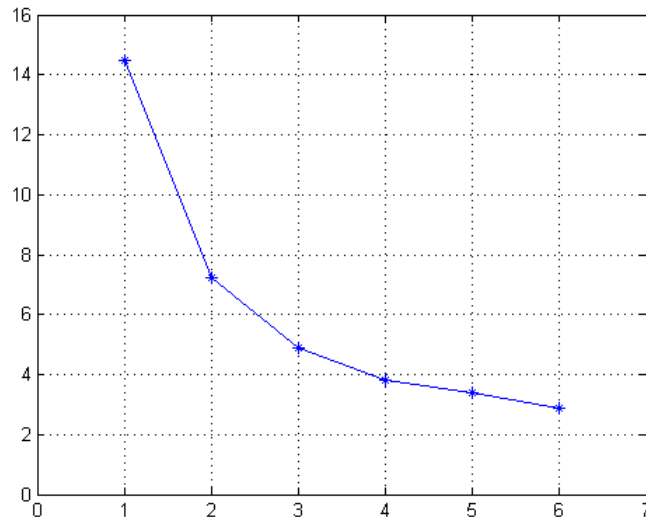


Рис. 2: График зависимости времени выполнения программы с MPI от количества ядер

Как видно из графика и листингов, в данном случае наблюдается существенное ускорение выполнения программы. На 6 ядрах, в сравнении с одним, программа выполняется на 80,1% быстрее.

## 4 Выводы

В работе исследованы варианты преобразования последовательной программы в параллельную с использованием технологий POSIX Threads и MPI. PThreads позволяют использовать параллельно несколько потоков. Главным достоинством такого подхода является отсутствие разделения адресных пространств взаимодействующих задач, что избавляет от необходимости задумываться о межпроцессном взаимодействии, а также позволяет локализовать изменения программы в случае преобразования её реализации из последовательной в параллельную. Вместе с тем, данный способ дал не очень большое повышение производительности. Скорее всего, это связано с использованием функции генерации случайных значений, обращение к которой невозможно произвести параллельно для одного процесса.

Технология MPI, наоборот, позволила реализовать приложение, производительность которого существенно улучшается с увеличением числа ядер, на которых оно исполняется. Межпроцессное взаимодействие внутри данной технологии осуществляется путём передачи сообщений между задачами и удобно при использовании простых типов передаваемых данных. Вместе с тем, чтобы передать сложные структуры данных между процессами, их придётся декомпонировать до простых типов, которые можно использовать в функциях MPI\_Send и MPI\_Recv. Определённым недостатком технологии можно считать необходимость изменения функции main с целью вставки туда MPI\_Init и MPI\_Finalize. В остальном изменения, связанные с распараллеливанием программы с использованием MPI локальны.

## 5 Листинги

Файлы parser.h, parser.cpp, randomizer.h и randomizer.cpp являются общими для всех программ. Файлы main.cpp, types.h, solver.h и solver.cpp изменены, в соответствии с реализациями.

### 1. Последовательная программа

- main.cpp

```

1 #include <iostream>
2 #include <stdlib.h>
3
4 #include "solver.h"
5 #include "parser.h"
6 #include "randomizer.h"
7
8 #define TEST
9

```

```

10 namespace def {
11     int MIN_ARGS_COUNT = 2;
12     int MAX_CIRCLES_COUNT = 100;
13     int DENSITY = 500;
14 };
15
16 void test_statistics(std::vector< Circle* > *circles){
17     int tests_count = 10;
18     std::cout << ">>>Running_statistics_test.This_test_consists_of_" << tests_count
19     → << "_iterations" << std::endl;
20     T results[tests_count];
21     for(int i = 0; i < tests_count; ++i){
22         results[i] = solve(circles, def::DENSITY, &random_uniform_real_distribution);
23         std::cout << "Result[" << i << "]_" << results[i] << std::endl;
24     }
25     std::cout << std::endl;
26
27     T mean = results[0];
28     for(int i = 1; i < tests_count; ++i){
29         mean += results[i];
30     }
31     mean /= tests_count;
32
33     std::cout << "Mean_value_" << mean << std::endl;
34
35     auto sqr = [](T x){return x*x;};
36     T disp = sqr(results[0] - mean);
37     for(int i = 1; i < tests_count; ++i){
38         disp += sqr(results[i] - mean);
39     }
40     disp /= tests_count;
41
42     std::cout << "Dispersion_" << disp << std::endl;
43 }
44
45 void run_tests(std::vector< Circle* > *circles){
46     test_statistics(circles);
47 }
48
49 int main() {
50     int status = -1;
51     std::cout.precision(10);
52
53     std::vector< Circle* > *circles = NULL;
54     const std::string filename("/home/oglandx/circles.txt");
55
56     try{
57         circles = parseCirclesFromFile(filename, def::MAX_CIRCLES_COUNT);
58     }
59     catch (FileException e)
60     {
61         std::cout << e.what();
62     }
63
64     if(NULL != circles)
65     {
66         srand((unsigned)time(NULL));
67
68 #ifdef TEST
69     run_tests(circles);
70 #else
71     T result = solve(circles, def::DENSITY, &random_simple);
72     std::cout << "Result_" << result << std::endl;
73 #endif
74     status = 0;
75 }
76
77 for(unsigned long i = 0; circles != NULL && i < circles->size(); ++i)
78 {
79     delete circles->at(i);
80 }
81 delete circles;
82
83 return status;
84 }

```

- types.h

```

1 //
2 // Created by oglandx on 3/16/16.
3 //
4
5 #ifndef TYPES_H
6 #define TYPES_H
7
8 #include <cmath>
9
10 typedef double T;
11
12 class Point
13 {
14 public:
15     T x;
16     T y;
17
18     Point(T x, T y)
19     {
20         this->x = x;
21         this->y = y;
22     }
23
24     Point(const Point *point)
25     {
26         this->x = point->x;
27         this->y = point->y;
28     }
29 };
30
31 class Circle
32 {
33 private:
34     inline static T sqr(T x){ return x*x; }
35 public:
36     Point *center;
37     T radius;
38
39     Circle(Point *center, T radius)
40     {
41         this->center = center;
42         this->radius = radius;
43     }
44
45     ~Circle()
46     {
47         delete center;
48     }
49
50     bool containsPoint(const Point *point)
51     {
52         return this->sqr(this->center->x - point->x) +
53                this->sqr(this->center->y - point->y) <= this->sqr(this->radius);
54     }
55 };
56
57
58 class Rect
59 {
60 public:
61     Point *tl;
62     Point *br;
63
64     Rect(Point *tl, Point *br)
65     {
66         this->tl = tl;
67         this->br = br;
68     }
69
70     Rect(T x0, T y0, T x1, T y1)
71     {
72         this->tl = new Point(x0, y0);
73         this->br = new Point(x1, y1);
74     }
75

```

```

76     ~Rect()
77     {
78         delete tl;
79         delete br;
80     }
81
82     T area() const
83     {
84         return std::fabs((tl->x - br->x)*(tl->y - br->y));
85     }
86 };
87
88 typedef Point* (*RandomFunction)(const Rect*, int);
89
90 #endif /* TYPES_H */

```

- parse.h

```

1  //
2  // Created by oglandx on 3/16/16.
3  //
4
5  #ifndef CIRCLESFIGUREAREA_PARSER_H
6  #define CIRCLESFIGUREAREA_PARSER_H
7
8  #include <string>
9  #include <vector>
10 #include <exception>
11 #include <cstring>
12 #include <sstream>
13 #include <iostream>
14 #include <fstream>
15
16 #include "types.h"
17
18 class FileException : std::exception
19 {
20 private:
21     char *reason;
22 public:
23     FileException(const char *what)
24     {
25         this->reason = new char[std::strlen(what)];
26         strcpy(this->reason, what);
27     }
28
29     const char* what() throw()
30     {
31         return this->reason;
32     }
33 };
34
35 std::vector< Circle* > *parseCirclesFromFile(const std::string &filename, int
    ↪ max_circles) throw(FileException);
36
37 #endif //CIRCLESFIGUREAREA_PARSER_H

```

- parse.cpp

```

1  /*
2   * File:    parser.h
3   * Author:  oglandx
4   *
5   * Created on February 17, 2016, 1:54 AM
6   */
7
8  #ifndef PARSER_H
9  #define PARSER_H
10
11 #include "parser.h"
12
13 std::vector< Circle* > *parseCirclesFromFile(const std::string &filename, int
    ↪ max_circles) throw(FileException)
14 {
15     std::vector< Circle* > *circles = new std::vector< Circle* >();
16

```

```

17     std::ifstream file;
18     file.open(filename.c_str());
19
20     if(!file.is_open())
21     {
22         std::stringstream reason;
23         reason << "Cannot_open_file_<" << filename << ">" << std::endl;
24         throw FileException(reason.str().c_str());
25     }
26
27     for(unsigned long i = 0; i < max_circles && !file.eof(); ++i)
28     {
29         std::string line;
30         std::getline(file, line);
31
32         std::istringstream line_to_parse(line);
33         Circle* circle = new Circle(new Point(0, 0), 0);
34         line_to_parse >> circle->center->x >> circle->center->y >> circle->radius;
35
36         if(!file.eof() && !file.bad())
37         {
38             if(!line_to_parse.bad())
39             {
40                 circles->push_back(circle);
41             }
42             else
43             {
44                 std::stringstream reason;
45                 reason << "Error_in_line_" << i + 1 << ":_not_enough_symbols" << std
→ ::endl;
46                 throw FileException(reason.str().c_str());
47             }
48         }
49         else
50         {
51             break;
52         }
53     }
54
55     file.close();
56
57     return circles;
58 }
59
60 #endif /* PARSER_H */

```

#### • solve.h

```

1  /*
2  * File:    solver.h
3  * Author:  oglandx
4  *
5  * Created on February 17, 2016, 1:54 AM
6  */
7
8  #ifndef CIRCLESFIGUREAREA_SOLVER_H
9  #define CIRCLESFIGUREAREA_SOLVER_H
10
11 #include "types.h"
12 #include <vector>
13 #include <stdlib.h>
14 #include <pthread.h>
15 #include <iostream>
16 #include <chrono>
17
18
19 T solve(const std::vector< Circle* > *circles, int density, RandomFunction random);
20 std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
21     const std::vector< Circle* > *circles, const Rect *rect, int density,
→ RandomFunction random);
22 Rect *getFigureRect(const std::vector< Circle* > *circles);
23 bool isPointInsideCircles(const std::vector< Circle* > *circles,
24     const Point *point);
25
26
27 #endif //CIRCLESFIGUREAREA_SOLVER_H

```

• solve.cpp

```

1 #include "solver.h"
2
3 T solve(const std::vector< Circle* > *circles, int density, RandomFunction random)
4 {
5     Rect *rect = getFigureRect(circles);
6     std::pair<unsigned long, unsigned long> *result = generateAndCheckPoints(circles,
7     ↪ rect, density, random);
8     if(NULL == result)
9     {
10         return (T)(-1);
11     }
12     unsigned long count = result->first;
13     unsigned long intersects = result->second;
14
15     delete result;
16
17     return static_cast<T>(intersects)/static_cast<T>(count)* rect->area();
18 }
19
20 std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
21     const std::vector< Circle* > *circles, const Rect *rect, int density,
22     ↪ RandomFunction random)
23 {
24     T area = rect->area();
25     unsigned long count = density*static_cast<unsigned long>(area);
26     unsigned long result = 0;
27
28     std::chrono::milliseconds start_time =
29     ↪ std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
30     ↪ system_clock::now().time_since_epoch());
31     std::cout << "$start_generation ...." << std::endl;
32
33     for(unsigned long i = 0; i < count; ++i)
34     {
35         Point *point = random(rect, density);
36         if (isPointInsideCircles(circles, point)) {
37             ++result;
38         }
39         delete point;
40     }
41
42     std::chrono::milliseconds end_time =
43     ↪ std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
44     ↪ system_clock::now().time_since_epoch());
45     std::cout << "end_generation_(time_=" << end_time.count() - start_time.count()
46     ↪ << ")" << std::endl;
47
48     return new std::pair<unsigned long, unsigned long>(count, result);
49 }
50
51 Rect *getFigureRect(const std::vector< Circle* > *circles)
52 {
53     typename std::vector< Circle* >::const_iterator it = circles->begin();
54
55     Point *min = new Point(
56         (*it)->center->x - (*it)->radius,
57         (*it)->center->y - (*it)->radius
58     );
59     Point *max = new Point(
60         (*it)->center->x + (*it)->radius,
61         (*it)->center->y + (*it)->radius
62     );
63
64     while(++it != circles->end())
65     {
66         if((*it)->center->x - (*it)->radius < min->x)
67         {
68             min->x = (*it)->center->x - (*it)->radius;
69         }
70         else if((*it)->center->x + (*it)->radius > max->x)
71         {
72             max->x = (*it)->center->x + (*it)->radius;
73         }
74     }
75
76     return new Rect(min, max);
77 }

```



```

71         if((*it)->center->y - (*it)->radius < min->y)
72         {
73             min->y = (*it)->center->y - (*it)->radius;
74         }
75         else if((*it)->center->y - (*it)->radius > max->y)
76         {
77             max->y = (*it)->center->y + (*it)->radius;
78         }
79     }
80     return new Rect(min, max);
81 }
82
83
84 bool isPointInsideCircles(const std::vector< Circle* > *circles, const Point *point)
85 {
86     for(typename std::vector< Circle* >::const_iterator it = circles->begin(); it !=
87         ↪ circles->end(); it++)
88     {
89         if((*it)->containsPoint(point))
90         {
91             return true;
92         }
93     }
94     return false;
95 }

```

- randomize.h

```

1  /*
2  * File:    randomizer.h
3  * Author:  oglandx
4  *
5  * Created on March 30, 2016, 1:10 AM
6  */
7
8  #ifndef RANDOMIZER_H
9  #define RANDOMIZER_H
10
11 #include <stdlib.h>
12 #include <random>
13
14 #include "types.h"
15
16 Point *random_simple(const Rect *rect, int density);
17 Point *random_uniform_real_distribution(const Rect *rect, int density);
18
19 #endif /* RANDOMIZER_H */

```

- randomize.cpp

```

1  /*
2  * File:    randomizer.cpp
3  * Author:  oglandx
4  *
5  * Created on March 30, 2016, 1:10 AM
6  */
7
8  #include "randomizer.h"
9
10 Point *random_simple(const Rect *rect, int density)
11 {
12     // ax = (x1-x0)/xm; ay = (y1-y0)/ym;
13     // bx = x0; by = y0;
14     // x = ax*rand + bx; y = ay*rand + by
15
16     Point norms = Point(
17         static_cast<T>(density),
18         static_cast<T>(density));
19     Point transform_a = Point(
20         (rect->br->x - rect->tl->x)/norms.x,
21         (rect->br->y - rect->tl->y)/norms.y);
22     Point *transform_b = rect->tl;
23
24     Point *result = new Point(
25         static_cast<T>(rand() % static_cast<int>(norms.x))*transform_a.x +
        ↪ transform_b->x,

```

```

26         static_cast<T>(rand() % static_cast<int>(norms.y))*transform_a.y +
    ↪ transform_b->y);
27     return result;
28 }
29
30 Point *random_uniform_real_distribution(const Rect *rect, int density){
31     static std::random_device seed;
32     static std::mt19937 generator(seed());
33     std::uniform_real_distribution<T> distribution_x(rect->tl->x, rect->br->x);
34     std::uniform_real_distribution<T> distribution_y(rect->tl->y, rect->br->y);
35
36     return new Point(distribution_x(generator), distribution_y(generator));
37 }

```

## 2. Модифицированная с использованием PThreads программа

### • main.cpp

```

1  #include <iostream>
2  #include <stdlib.h>
3
4  #include "solver.h"
5  #include "parser.h"
6  #include "randomizer.h"
7
8  #define TEST
9
10 namespace def {
11     int MAX_CIRCLES_COUNT = 100;
12     int DENSITY = 500;
13     int TESTS_COUNT = 50;
14     int MAX_CORES = 6;
15 };
16
17 void show_circles(std::vector< Circle* > *circles)
18 {
19     int i = 0;
20     std::cout << "[Loaded_circles]" << std::endl;
21     for(std::vector< Circle* >::const_iterator it = circles->begin(); it != circles->
    ↪ end(); it++)
22     {
23         std::cout << "Circle(" << i++ << ")_r_=" << (*it)->radius <<
24         ",_center_=" << (*it)->center->x << ",_=" << (*it)->center->y << ")" << std
    ↪ ::endl;
25     }
26     std::cout << std::endl << std::endl;
27 }
28
29 template<typename __type>
30 void print_statistics(const std::string &name, __type results[], int count)
31 {
32     std::cout << "[" << name << "]" << std::endl;
33
34     __type mean = results[0];
35     for(int i = 1; i < count; ++i)
36     {
37         mean += results[i];
38     }
39     mean /= count;
40
41     std::cout << "Mean_value_=" << mean << std::endl;
42
43     auto sqr = []( __type x){return x*x;};
44
45     __type disp = sqr(results[0] - mean);
46
47     for(int i = 1; i < count; ++i)
48     {
49         disp += sqr(results[i] - mean);
50     }
51     disp /= count;
52
53     std::cout << "Dispersion_=" << disp << std::endl << std::endl;
54 }
55
56 void test_statistics(std::vector< Circle* > *circles)

```

```

57 {
58     int tests_count = def::TESTS_COUNT;
59
60     show_circles(circles);
61
62     T results[tests_count];
63     double times[tests_count];
64
65     for(int i = 1; i < def::MAX_CORES + 1; ++i)
66     {
67         std::cout << ">>>Running_statistics_test.This_test_consists_of_" <<
        ↪ tests_count <<
68             "_iterations_for_" << i << "_cores" << std::endl << std::endl;
69         for(int j = 0; j < tests_count; ++j)
70         {
71             double *result_time = new double;
72             results[j] = solve(circles, def::DENSITY,
        ↪ random_uniform_real_distribution, i, result_time);
73             times[j] = *result_time;
74             delete result_time;
75
76             std::cout.precision(10);
77             std::cout << "Result[" << j << "]=" << results[j] << std::endl << std::
        ↪ endl;
78         }
79         std::cout << std::endl;
80
81         print_statistics<T>(std::string("Result"), results, tests_count);
82
83         std::cout.precision(5);
84         print_statistics<double>(std::string("Time"), times, tests_count);
85     }
86 }
87
88 void run_tests(std::vector< Circle* > *circles){
89     test_statistics(circles);
90 }
91
92 int main() {
93     int status = -1;
94
95     std::vector< Circle* > *circles = NULL;
96     const std::string filename("/home/oglandx/circles.txt");
97
98     try{
99         circles = parseCirclesFromFile(filename, def::MAX_CIRCLES_COUNT);
100     }
101     catch (FileException e)
102     {
103         std::cout << e.what();
104     }
105
106     if(NULL != circles)
107     {
108         srand((unsigned)time(NULL));
109
110 #ifndef TEST
111     run_tests(circles);
112 #else
113     T result = solve(circles, def::DENSITY, &random_simple);
114     std::cout << "Result=" << result << std::endl;
115 #endif
116     status = 0;
117 }
118
119
120 for(unsigned long i = 0; circles != NULL && i < circles->size(); ++i)
121 {
122     delete circles->at(i);
123 }
124
125 delete circles;
126
127 return status;
128 }

```

- types.h

```

1 //
2 // Created by oglandx on 3/16/16.
3 //
4
5 #ifndef TYPES_H
6 #define TYPES_H
7
8 #include <cmath>
9 #include <vector>
10 #include <pthread.h>
11
12 typedef double T;
13
14 class Point
15 {
16 public:
17     T x;
18     T y;
19
20     Point(T x, T y)
21     {
22         this->x = x;
23         this->y = y;
24     }
25
26     Point(const Point *point)
27     {
28         this->x = point->x;
29         this->y = point->y;
30     }
31 };
32
33 class Circle
34 {
35 private:
36     inline static T sqr(T x){ return x*x; }
37 public:
38     Point *center;
39     T radius;
40
41     Circle(Point *center, T radius)
42     {
43         this->center = center;
44         this->radius = radius;
45     }
46
47     ~Circle()
48     {
49         delete center;
50     }
51
52     bool containsPoint(const Point *point)
53     {
54         return this->sqr(this->center->x - point->x) +
55                this->sqr(this->center->y - point->y) <= this->sqr(this->radius);
56     }
57 };
58
59
60 class Rect
61 {
62 public:
63     Point *tl;
64     Point *br;
65
66     Rect(Point *tl, Point *br)
67     {
68         this->tl = tl;
69         this->br = br;
70     }
71
72     Rect(T x0, T y0, T x1, T y1)
73     {
74         this->tl = new Point(x0, y0);
75         this->br = new Point(x1, y1);

```

```

76     }
77
78     ~Rect()
79     {
80         delete tl;
81         delete br;
82     }
83
84     T area() const
85     {
86         return static_cast<T>(std::fabs((tl->x - br->x)*(tl->y - br->y)));
87     }
88 };
89
90 typedef Point* (*RandomFunction)(const Rect*, int);
91
92 struct PointGeneratorStruct{
93     unsigned long count;
94     int density;
95     const Rect *rect;
96     RandomFunction random;
97     const std::vector< Circle* > *circles;
98     unsigned long result;
99
100     PointGeneratorStruct(unsigned long count,
101                          int density,
102                          const Rect *rect,
103                          RandomFunction random,
104                          const std::vector< Circle* > *circles)
105     {
106         this->count = count;
107         this->density = density;
108         this->rect = rect;
109         this->random = random;
110         this->circles = circles;
111     }
112 };
113
114 #endif /* TYPES_H */

```

- solve.h

```

1  /*
2  * File:    solver.h
3  * Author:  oglandx
4  *
5  * Created on February 17, 2016, 1:54 AM
6  */
7
8  #ifndef CIRCLESFIGUREAREA_SOLVER_H
9  #define CIRCLESFIGUREAREA_SOLVER_H
10
11  #include "types.h"
12  #include <vector>
13  #include <stdlib.h>
14  #include <pthread.h>
15  #include <iostream>
16  #include <chrono>
17
18
19  T solve(const std::vector< Circle* > *circles, int density, RandomFunction random,
20         ↪ int max_cores,
21         double *result_time = NULL);
22  std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
23         ↪ const std::vector< Circle* > *circles, const Rect *rect, int density,
24         ↪ RandomFunction random, int max_cores,
25         double *result_time = NULL);
26  Rect *getFigureRect(const std::vector< Circle* > *circles);
27  bool isPointInsideCircles(const std::vector< Circle* > *circles,
28                           const Point *point);
29
30 #endif //CIRCLESFIGUREAREA_SOLVER_H

```

- solve.cpp

```

1 #include "solver.h"
2
3 T solve(const std::vector< Circle* > *circles, int density, RandomFunction random,
4         ↪ int max_cores, double *result_time)
5 {
6     Rect *rect = getFigureRect(circles);
7     std::pair<unsigned long, unsigned long> *result =
8         generateAndCheckPoints(circles, rect, density, random, max_cores,
9         ↪ result_time);
10
11     if(NULL == result)
12     {
13         return (T)(-1);
14     }
15
16     unsigned long count = result->first;
17     unsigned long intersects = result->second;
18
19     return static_cast<T>(intersects)/static_cast<T>(count)* rect->area();
20 }
21
22 void *generatePointsThread(void *params)
23 {
24     PointGeneratorStruct *data = (PointGeneratorStruct*)params;
25     data->result = 0;
26     for(unsigned long i = 0; i < data->count; ++i)
27     {
28         Point* point = data->random(data->rect, data->density);
29         if(isPointInsideCircles(data->circles, point))
30         {
31             ++(data->result);
32         }
33         delete point;
34     }
35     return NULL;
36 }
37
38 std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
39     const std::vector< Circle* > *circles, const Rect *rect, int density,
40     ↪ RandomFunction random, int max_cores,
41     double *result_time)
42 {
43     if(max_cores < 1)
44     {
45         return NULL;
46     }
47
48     T area = rect->area();
49     unsigned long count = density*static_cast<unsigned long>(area);
50
51     std::chrono::milliseconds start_time =
52         std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
53         ↪ system_clock::now().time_since_epoch());
54     std::cout << "$start_generation...." << std::endl;
55
56     PointGeneratorStruct *params[max_cores];
57
58     pthread_t threads[max_cores - 1];
59     for(int i = 0; i < max_cores - 1; ++i)
60     {
61         params[i] = new PointGeneratorStruct(count/max_cores, density, rect, random,
62         ↪ circles);
63         pthread_create(
64             &threads[i],
65             NULL,
66             generatePointsThread,
67             params[i]
68         );
69     }
70
71     unsigned long real_count = count - count*(max_cores - 1)/max_cores;
72     params[max_cores - 1] =
73         new PointGeneratorStruct(real_count + real_count % max_cores, density,
74         ↪ rect, random, circles);
75     generatePointsThread(params[max_cores - 1]);
76 }

```

```

71     for(int i = 0; i < max_cores - 1; ++i)
72     {
73         pthread_join(threads[i], NULL);
74     }
75
76     std::cout << "Using_" << max_cores << "_cores" << std::endl;
77
78     unsigned long result = params[0]->result;
79
80     for(int i = 1; i < max_cores; ++i)
81     {
82         result += params[i]->result;
83         delete params[i];
84     }
85
86     std::chrono::milliseconds end_time =
87         std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
88         ↪ system_clock::now().time_since_epoch());
89     double time = static_cast<double>(end_time.count() - start_time.count())/1000;
90
91     std::cout << "end_generation_(time=_" << time << "s)" << std::endl;
92
93     if(NULL != result_time)
94     {
95         *result_time = time;
96     }
97
98     return new std::pair<unsigned long, unsigned long>(count, result);
99 }
100 Rect *getFigureRect(const std::vector< Circle* > *circles)
101 {
102     typename std::vector< Circle* >::const_iterator it = circles->begin();
103
104     Point *min = new Point(
105         (*it)->center->x - (*it)->radius,
106         (*it)->center->y - (*it)->radius
107     );
108     Point *max = new Point(
109         (*it)->center->x + (*it)->radius,
110         (*it)->center->y + (*it)->radius
111     );
112
113     while(++it != circles->end())
114     {
115         if((*it)->center->x - (*it)->radius < min->x)
116         {
117             min->x = (*it)->center->x - (*it)->radius;
118         }
119         else if((*it)->center->x + (*it)->radius > max->x)
120         {
121             max->x = (*it)->center->x + (*it)->radius;
122         }
123
124         if((*it)->center->y - (*it)->radius < min->y)
125         {
126             min->y = (*it)->center->y - (*it)->radius;
127         }
128         else if((*it)->center->y + (*it)->radius > max->y)
129         {
130             max->y = (*it)->center->y + (*it)->radius;
131         }
132     }
133     return new Rect(min, max);
134 }
135
136
137 bool isPointInsideCircles(const std::vector< Circle* > *circles, const Point *point)
138 {
139     for(typename std::vector< Circle* >::const_iterator it = circles->begin(); it !=
140     ↪ circles->end(); it++)
141     {
142         if((*it)->containsPoint(point))
143         {
144             return true;
145         }
146     }

```

```

145     }
146     return false;
147 }

```

### 3. Модифицированная с использованием MPI программа

- main.cpp

```

1  #include <iostream>
2  #include <mpi.h>
3
4  #include "solver.h"
5  #include "parser.h"
6  #include "randomizer.h"
7
8  #define TEST
9
10 namespace def {
11     int MAX_CIRCLES_COUNT = 100;
12     int DENSITY = 500;
13     int TESTS_COUNT = 50;
14 };
15
16 void show_circles(std::vector< Circle* > *circles)
17 {
18     int i = 0;
19     std::cout << "[Loaded_circles]" << std::endl;
20     for(std::vector< Circle* >::const_iterator it = circles->begin(); it != circles->
21         ↪ end(); it++)
22     {
23         std::cout << "Circle(" << i++ << ")_r=_ " << (*it)->radius <<
24         ↪ ",_center=_(" << (*it)->center->x << ",_ " << (*it)->center->y << ")" << std
25         ↪ ::endl;
26     }
27     std::cout << std::endl << std::endl;
28 }
29
30 template<typename __type>
31 void print_statistics(const std::string &name, __type results[], int count)
32 {
33     std::cout.precision(5);
34
35     std::cout << "[" << name << "]" << std::endl;
36
37     __type mean = results[0];
38     for(int i = 1; i < count; ++i)
39     {
40         mean += results[i];
41     }
42     mean /= count;
43
44     std::cout << "Mean_value=_ " << mean << std::endl;
45
46     auto sqr = []( __type x){return x*x;};
47
48     __type disp = sqr(results[0] - mean);
49
50     for(int i = 1; i < count; ++i)
51     {
52         disp += sqr(results[i] - mean);
53     }
54     disp /= count;
55
56     std::cout << "Dispersion=_ " << disp << std::endl << std::endl;
57 }
58
59 void test_statistics(std::vector< Circle* > *circles)
60 {
61     int tests_count = def::TESTS_COUNT;
62
63     int world_size;
64     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
65
66     int mpi_rank;
67     MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

```



```

67     const int main_rank = 0;
68
69     if(mpi_rank == main_rank)
70     {
71         std::cout << ">>>_Running_tests" << std::endl;
72
73         show_circles(circles);
74
75         std::cout << "[Running_statistics_test]_This_test_consists_of_" <<
76         tests_count << "_iterations_for_" << world_size << "_cores" << std::endl <<
77         std::endl;
78     }
79     T results[tests_count];
80     double times[tests_count];
81
82     for(int i = 0; i < tests_count; ++i)
83     {
84         std::chrono::milliseconds start_time =
85             std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
86             ↪ system_clock::now().time_since_epoch());
87
88         results[i] = solve(circles, def::DENSITY, &random_uniform_real_distribution,
89         ↪ world_size);
90
91         std::chrono::milliseconds end_time =
92             std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
93             ↪ system_clock::now().time_since_epoch());
94         times[i] = static_cast<double>(end_time.count() - start_time.count())/1000;
95
96         std::cout << "end_generation_(process_=" << mpi_rank << ",_time_=" << times
97         ↪ [i] << "s)" << std::endl;
98
99         MPI_Barrier(MPI_COMM_WORLD);
100
101         if((T)(-1) != results[i])
102         {
103             std::cout.precision(10);
104             std::cout << "Result[" << i << "]_=" << results[i] << std::endl << std::
105             ↪ endl;
106         }
107     }
108     if((T)(-1) != results[0]){
109         std::cout << std::endl;
110         print_statistics<T>(std::string("Result"), results, tests_count);
111     }
112     MPI_Barrier(MPI_COMM_WORLD);
113
114     if(mpi_rank == 0){
115         double full_times[tests_count*world_size];
116         memcpy(full_times, times, sizeof(double)*tests_count);
117
118         MPI_Status status;
119         for(int rank = 1; rank < world_size; ++rank)
120         {
121             MPI_Recv(times, tests_count, MPI_DOUBLE, rank, 4, MPI_COMM_WORLD, &status
122             ↪ );
123             memcpy(full_times + tests_count*rank, times, sizeof(double)*tests_count);
124         }
125         print_statistics<double>(std::string("Time"), times, tests_count);
126     }
127     else
128     {
129         MPI_Send(times, tests_count, MPI_DOUBLE, 0, 4, MPI_COMM_WORLD);
130     }
131     std::cout << std::endl;
132 }
133
134 void run_tests(std::vector< Circle* > *circles){
135     test_statistics(circles);
136 }
137
138 int main(int argc, char *argv[]) {
139     int status = -1;
140
141     std::vector< Circle* > *circles = NULL;

```

```

136     const std::string filename("/home/oglandx/circles.txt");
137
138     try{
139         circles = parseCirclesFromFile(filename, def::MAX_CIRCLES_COUNT);
140     }
141     catch (FileException e)
142     {
143         std::cout << e.what();
144     }
145
146     // ——— MPI section ———
147
148     int rc = MPI_Init(&argc, &argv);
149
150     if(rc)
151     {
152         std::cout << "MPI_error_occurred" << std::endl;
153         MPI_Abort(MPI_COMM_WORLD, rc);
154         return status;
155     }
156
157     int world_size;
158     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
159
160     if(NULL != circles)
161     {
162         #ifdef TEST
163
164         run_tests(circles);
165
166         #else
167
168         T result = solve(circles, def::DENSITY, &random_uniform_real_distribution,
169 ↪ world_size);
170         if((T)(-1) != result){
171             std::cout << "Result_=_ " << result << std::endl;
172         }
173
174         #endif
175         status = 0;
176     }
177
178     MPI_Finalize();
179
180     // ——— /MPI ———
181
182     for(unsigned long i = 0; circles != NULL && i < circles->size(); ++i)
183     {
184         delete circles->at(i);
185     }
186     delete circles;
187
188     return status;
189 }

```

- types.h

```

1 //
2 // Created by oglandx on 3/16/16.
3 //
4
5 #ifndef TYPES_H
6 #define TYPES_H
7
8 #include <cmath>
9 #include <vector>
10 #include <pthread.h>
11
12 typedef double T;
13
14 class Point
15 {
16 public:
17     T x;
18     T y;
19
20     Point(T x, T y)

```

```

21     {
22         this->x = x;
23         this->y = y;
24     }
25
26     Point(const Point *point)
27     {
28         this->x = point->x;
29         this->y = point->y;
30     }
31 };
32
33 class Circle
34 {
35 private:
36     inline static T sqr(T x){ return x*x; }
37 public:
38     Point *center;
39     T radius;
40
41     Circle(Point *center, T radius)
42     {
43         this->center = center;
44         this->radius = radius;
45     }
46
47     ~Circle()
48     {
49         delete center;
50     }
51
52     bool containsPoint(const Point *point)
53     {
54         return this->sqr(this->center->x - point->x) +
55             this->sqr(this->center->y - point->y) <= this->sqr(this->radius);
56     }
57 };
58
59
60 class Rect
61 {
62 public:
63     Point *tl;
64     Point *br;
65
66     Rect(Point *tl, Point *br)
67     {
68         this->tl = tl;
69         this->br = br;
70     }
71
72     Rect(T x0, T y0, T x1, T y1)
73     {
74         this->tl = new Point(x0, y0);
75         this->br = new Point(x1, y1);
76     }
77
78     ~Rect()
79     {
80         delete tl;
81         delete br;
82     }
83
84     T area() const
85     {
86         return std::fabs((tl->x - br->x)*(tl->y - br->y));
87     }
88 };
89
90 typedef Point* (*RandomFunction)(const Rect*, int);
91
92 struct PointGeneratorStruct{
93     unsigned long count;
94     int density;
95     const Rect *rect;
96

```

```

97     PointGeneratorStruct(unsigned long count,
98                           int density,
99                           const Rect *rect)
100 {
101     this->count = count;
102     this->density = density;
103     this->rect = rect;
104 }
105 };
106
107 #endif /* TYPES_H */

```

- solve.h

```

1  /*
2  * File:    solver.h
3  * Author:  oglandx
4  *
5  * Created on February 17, 2016, 1:54 AM
6  */
7
8  #ifndef CIRCLESFIGUREAREA_SOLVER_H
9  #define CIRCLESFIGUREAREA_SOLVER_H
10
11 #include "types.h"
12 #include <vector>
13 #include <stdlib.h>
14 #include <iostream>
15 #include <chrono>
16
17
18 T solve(const std::vector< Circle* > *circles, int density, RandomFunction random,
19         ↪ int max_cores);
20 std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
21     const std::vector< Circle* > *circles, const Rect *rect, int density,
22     ↪ RandomFunction random, int max_cores);
23 Rect *getFigureRect(const std::vector< Circle* > *circles);
24 bool isPointInsideCircles(const std::vector< Circle* > *circles,
25                           const Point *point);
26
27
28 #endif //CIRCLESFIGUREAREA_SOLVER_H

```

- solve.cpp

```

1  #include "solver.h"
2
3  #include <mpi.h>
4
5  T solve(const std::vector< Circle* > *circles, int density, RandomFunction random,
6         ↪ int max_cores)
7  {
8      Rect *rect = getFigureRect(circles);
9      std::pair<unsigned long, unsigned long> *result = generateAndCheckPoints(circles,
10      ↪ rect, density, random, max_cores);
11      if(NULL == result)
12      {
13          return (T)(-1);
14      }
15
16      unsigned long count = result->first;
17      unsigned long intersects = result->second;
18
19      return static_cast<T>(intersects)/static_cast<T>(count) * rect->area();
20  }
21
22 unsigned long generatePointsProcess(const std::vector< Circle* > *circles,
23                                     unsigned long count, int density, const Rect rect, RandomFunction
24                                     ↪ random)
25 {
26     unsigned long result = 0;
27     for(unsigned long i = 0; i < count; ++i)
28     {
29         Point* point = random(rect, density);
30         if(isPointInsideCircles(circles, point))
31         {

```

```

29         ++result;
30     }
31     delete point;
32 }
33 return result;
34 }
35
36 void sendResult(unsigned long result, int to)
37 {
38     MPI_Send(&result, 1, MPI_UNSIGNED_LONG, to, 3, MPI_COMM_WORLD);
39 }
40
41 void recvResult(unsigned long *result, int from)
42 {
43     MPI_Status status;
44     MPI_Recv(result, 1, MPI_UNSIGNED_LONG, from, 3, MPI_COMM_WORLD, &status);
45 }
46
47 std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
48     const std::vector< Circle* > *circles, const Rect *rect, int density,
49     ↪ RandomFunction random, int max_cores)
50 {
51     if(max_cores < 1)
52     {
53         return NULL;
54     }
55
56     T area = rect->area();
57     unsigned long count = density*static_cast<unsigned long>(area);
58
59     unsigned long result;
60
61     // —— MPI section ——
62
63     int mpi_rank;
64     const int rank_main = 0;
65
66     MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
67
68     if(mpi_rank == rank_main)
69     {
70         unsigned long real_count = count - count*(max_cores - 1)/max_cores;
71         result = generatePointsProcess(circles, real_count, density, rect, random);
72     }
73     else
74     {
75         result = generatePointsProcess(circles, count/max_cores, density, rect,
76         ↪ random);
77         sendResult(result, rank_main);
78         return NULL;
79     }
80
81     for(int rank = 1; rank < max_cores; ++rank)
82     {
83         unsigned long current_result;
84         recvResult(&current_result, rank);
85         result += current_result;
86     }
87
88     // —— /MPI ——
89
90     return new std::pair<unsigned long, unsigned long>(count, result);
91 }
92
93 Rect *getFigureRect(const std::vector< Circle* > *circles)
94 {
95     typename std::vector< Circle* >::const_iterator it = circles->begin();
96
97     Point *min = new Point(
98         (*it)->center->x - (*it)->radius,
99         (*it)->center->y - (*it)->radius
100     );
101     Point *max = new Point(
102         (*it)->center->x + (*it)->radius,
103         (*it)->center->y + (*it)->radius
104     );

```

```

103
104 while(++it != circles->end())
105 {
106     if((*it)->center->x - (*it)->radius < min->x)
107     {
108         min->x = (*it)->center->x - (*it)->radius;
109     }
110     else if((*it)->center->x + (*it)->radius > max->x)
111     {
112         max->x = (*it)->center->x + (*it)->radius;
113     }
114
115     if((*it)->center->y - (*it)->radius < min->y)
116     {
117         min->y = (*it)->center->y - (*it)->radius;
118     }
119     else if((*it)->center->y + (*it)->radius > max->y)
120     {
121         max->y = (*it)->center->y + (*it)->radius;
122     }
123 }
124 return new Rect(min, max);
125 }
126
127
128 bool isPointInsideCircles(const std::vector< Circle* > *circles, const Point *point)
129 {
130     for(typename std::vector< Circle* >::const_iterator it = circles->begin(); it !=
131     ↪ circles->end(); it++)
132     {
133         if((*it)->containsPoint(point))
134         {
135             return true;
136         }
137     }
138     return false;
139 }

```