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

Отчет по лабораторной 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

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:

```
Rect *getFigureRect(const std::vector< Circle* > *circles)
{
    typename std::vector< Circle* >::const_iterator it = circles->begin();

    Point *min = new Point(
            (*it)->center->x - (*it)->radius,
            (*it)->center->y - (*it)->radius
    );
    Point *max = new Point(
            (*it)->center->x + (*it)->radius,
            (*it)->center->y + (*it)->radius
    );

    while(++it != circles->end())
    {
        if((*it)->center->x - (*it)->radius < min->x)
        {
            min->x = (*it)->center->x - (*it)->radius;
        }
        else if((*it)->center->x + (*it)->radius > max->x)
        {
            max->x = (*it)->center->x + (*it)->radius;
        }

        if((*it)->center->y - (*it)->radius < min->y)
        {
            min->y = (*it)->center->y - (*it)->radius;
        }
        else if((*it)->center->y - (*it)->radius > max->y)
        {
            max->y = (*it)->center->y + (*it)->radius;
        }
    }
    return new Rect(min, max);
}
```

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

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

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

```
    for(unsigned long i = 0; i < count; ++i)
    {
        Point *point = random(rect, density);
        if (isPointInsideCircles(circles, point)) {
            ++result;
        }
        delete point;
    }
```

Функция isPointInsideCircles позволяет проверить, находится ли точка внутри окружностей:

```
bool isPointInsideCircles(const std::vector< Circle* > *circles, const Point *point)
{
    for(typename std::vector< Circle* >::const_iterator it = circles->begin(); it !=
    ↪ circles->end(); it++)
    {
        if((*it)->containsPoint(point))
        {
            return true;
        }
    }
    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 вдальнейшем 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:

```
struct PointGeneratorStruct{
    unsigned long count;
    int density;
    const Rect *rect;
    RandomFunction random;
    const std::vector< Circle* > *circles;
    unsigned long result;

    PointGeneratorStruct(unsigned long count,
                         int density,
                         const Rect *rect,
                         RandomFunction random,
                         const std::vector< Circle* > *circles)
    {
        this->count = count;
        this->density = density;
        this->rect = rect;
        this->random = random;
        this->circles = circles;
    }
};
```

Функция, реализующая функциональность одного потока:

```
void *generatePointsThread(void *params)
{
    PointGeneratorStruct *data = (PointGeneratorStruct*)params;
    data->result = 0;
    for(unsigned long i = 0; i < data->count; ++i)
    {
        Point* point = data->random(data->rect, data->density);
        if(isPointInsideCircles(data->circles, point))
        {
            ++(data->result);
        }
        delete point;
    }
    return NULL;
```

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

В функции generateAndCheckPoints сначала создаются потоки:

```
    PointGeneratorStruct *params[max_cores];

    pthread_t threads[max_cores - 1];
    for(int i = 0; i < max_cores - 1; ++i)
    {
        params[i] = new PointGeneratorStruct(count/max_cores, density, rect, random, circles);
        pthread_create(
                &threads[i],
                NULL,
                generatePointsThread,
                params[i]
        );
    }
```

Затем производятся вычисления на главном потоке:

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

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

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

Сделав это, главный поток проводит редукцию результатов:

```
    unsigned long result = params[0]->result;

    for(int i = 1; i < max_cores; ++i)
    {
        result += params[i]->result;
        delete params[i];
    }
```

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 коммуникатора. Группа процессов - это упорядоченная коллекция процессов. Коммуникатор же позволяет общаться процессам внутри группы или между группами.

Также в программе используется функция MPI_Comm_size, позволяющая получить размер коммуникатора, а также стандартный коммуникатор MPI_COMM_WORLD, содержащий все созданные процессы.

С помощью функции MPI_Comm_rank можно узнать номер процесса в текущем коммуникаторе.

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

```
    unsigned long result;

    // ---- MPI section ----

    int mpi_rank;
    const int rank_main = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

    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 }
```

Используется блокирующая передача сообщений, при которой процесс-отправитель ожидает, пока процесс-получатель примет переданное сообщение.

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.

3.1 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

```
[Result]
Mean value = 31413.79668
Dispersion = 12.39041898

[Time]
Mean value = 11.11
Dispersion = 0.05943
```

- 3 потока

```
[Result]
Mean value = 31414.81872
Dispersion = 13.91885692

[Time]
Mean value = 8.9818
Dispersion = 0.54635
```

- 4 потока

```
[Result]
Mean value = 31413.73856
Dispersion = 12.62161521

[Time]
Mean value = 8.1528
Dispersion = 0.10264
```

- 5 потоков

```
[Result]
Mean value = 31413.39316
Dispersion = 17.33276977

[Time]
Mean value = 7.408
Dispersion = 0.20057
```

- 6 потоков

```
[Result]
Mean value = 31413.01008
Dispersion = 14.11241007

[Time]
Mean value = 7.2377
Dispersion = 0.076595
```

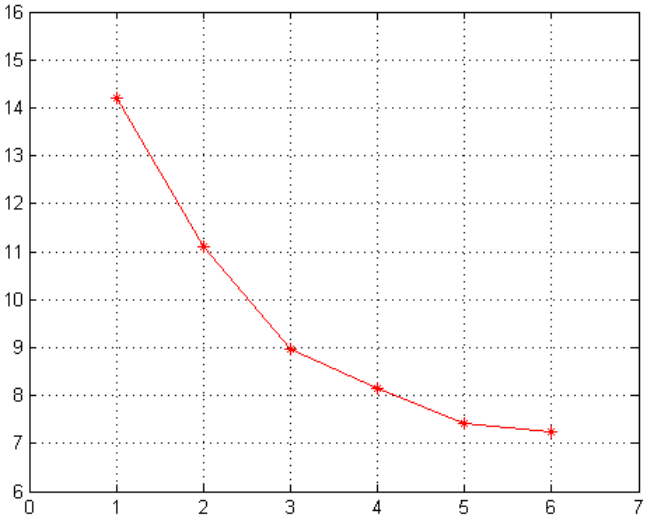

Рис. 1: График зависимости времени выполнения программы с PThreads от количества ядер

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

3.2 Многопроцессная программа с использованием MPI

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

```
#!/bin/sh

max_cores=6

for i in `seq 1 $max_cores`
do
   mpirun -np $i ./build/dist/CirclesFigureArea
done
```

Результат, в зависимости от количества задействованных процессов:

- 1 процесс

```
[Result]
Mean value = 31416
Dispersion = 14.248

[Time]
Mean value = 14.485
Dispersion = 0.075995
```

- 2 процесса

```
[Result]
Mean value = 31416
Dispersion = 12.289

[Time]
Mean value = 7.2112
Dispersion = 0.00044222
```

- 3 процесса

```
[Result]
Mean value = 31416
Dispersion = 10.529

[Time]
Mean value = 4.8756
Dispersion = 0.0049912
```

- 4 процесса

```
[Result]
Mean value = 31416
Dispersion = 16.05

[Time]
Mean value = 3.8008
Dispersion = 0.0063037
```

- 5 процессов

```
[Result]
Mean value = 31416
Dispersion = 12.64

[Time]
Mean value = 3.377
Dispersion = 0.0082743
```

- 6 процессов

```
[Result]
Mean value = 31416
Dispersion = 8.79

[Time]
Mean value = 2.8819
Dispersion = 0.044522
```

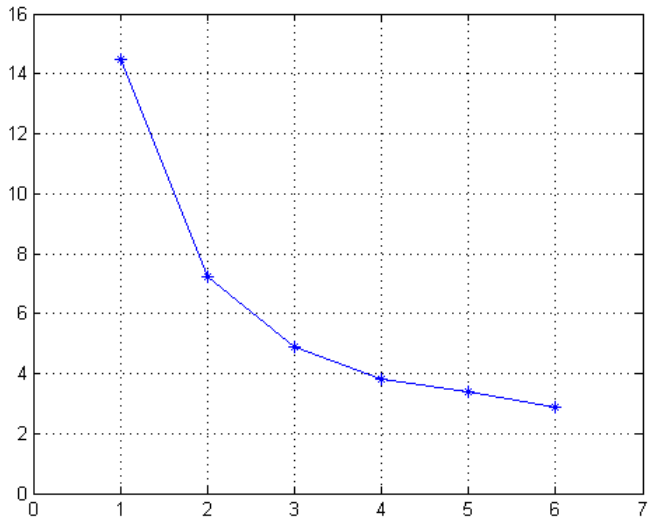

Рис. 2: График зависимости времени выполнения программы с MPI от количества ядер

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% быстрее, т.е. достигается ускорение в 5 раз, по сравнению с программой, использующей одно ядро.

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 декомпозировать 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

```cpp
#include <iostream>
#include <stdlib.h>

#include "solver.h"
#include "parser.h"
#include "randomizer.h"

#define TEST

namespace def {
    int MIN_ARGS_COUNT = 2;
    int MAX_CIRCLES_COUNT = 100;
    int DENSITY = 500;
};

void test_statistics(std::vector< Circle* > *circles){
    int tests_count = 10;
    std::cout << ">>>Running statistics test. This test consists of " << tests_count
    ↪ << " iterations" << std::endl;
    T results[tests_count];
    for(int i = 0; i < tests_count; ++i){
        results[i] = solve(circles, def::DENSITY, &random_uniform_real_distribution);
        std::cout << "Result[" << i << "] = " << results[i] << std::endl;
    }
    std::cout << std::endl;

    T mean = results[0];
    for(int i = 1; i < tests_count; ++i){
        mean += results[i];
    }
    mean /= tests_count;

    std::cout << "Mean value = " << mean << std::endl;

    auto sqr = [](T x){return x*x;};
    T disp = sqr(results[0] - mean);
    for(int i = 0; i < tests_count; ++i){
        disp += sqr(results[0] - mean);
    }
    disp /= tests_count;

    std::cout << "Dispersion = " << disp << std::endl;
}

void run_tests(std::vector< Circle* > *circles){
    test_statistics(circles);
}

int main() {
    int status = -1;
    std::cout.precision(10);
```

```
51
52      std::vector< Circle* >  *circles = NULL;
53      const std::string filename("/home/oglandx/circles.txt");
54
55      try{
56          circles = parseCirclesFromFile(filename, def::MAX_CIRCLES_COUNT);
57      }
58      catch (FileException e)
59      {
60          std::cout << e.what();
61      }
62
63      if(NULL != circles)
64      {
65          srand((unsigned)time(NULL));
66
67  #ifdef TEST
68          run_tests(circles);
69  #else
70          T result = solve(circles, def::DENSITY, &random_simple);
71          std::cout << "Result = " << result << std::endl;
72  #endif
73          status = 0;
74      }
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,
       ↪  rect, density, random);
7      if(NULL == result)
8      {
9          return (T)(-1);
10     }
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,
       ↪ RandomFunction random)
22 {
23     T area = rect->area();
24     unsigned long count = density*static_cast<unsigned long>(area);
25     unsigned long result = 0;
26
27     std::chrono::milliseconds start_time =
28             std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
       ↪ system_clock::now().time_since_epoch());
29     std::cout << "$start_generation...." << std::endl;
30
31     for(unsigned long i = 0; i < count; ++i)
32     {
33         Point *point = random(rect, density);
34         if (isPointInsideCircles(circles, point)) {
35             ++result;
36         }
37         delete point;
```

```
38      }
39
40      std::chrono::milliseconds end_time =
41              std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
        ↪ system_clock::now().time_since_epoch());
42      std::cout << "end_generation (time = " << end_time.count() - start_time.count()
        ↪ << ")" << std::endl;
43
44      return new std::pair<unsigned long, unsigned long>(count, result);
45  }
46
47  Rect *getFigureRect(const std::vector< Circle* > *circles)
48  {
49      typename std::vector< Circle* >::const_iterator it = circles->begin();
50
51      Point *min = new Point(
52              (*it)->center->x - (*it)->radius,
53              (*it)->center->y - (*it)->radius
54      );
55      Point *max = new Point(
56              (*it)->center->x + (*it)->radius,
57              (*it)->center->y + (*it)->radius
58      );
59
60      while(++it != circles->end())
61      {
62          if((*it)->center->x - (*it)->radius < min->x)
63          {
64              min->x = (*it)->center->x - (*it)->radius;
65          }
66          else if((*it)->center->x + (*it)->radius > max->x)
67          {
68              max->x = (*it)->center->x + (*it)->radius;
69          }
70
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 !=
        ↪ circles->end(); it++)
87      {
88          if((*it)->containsPoint(point))
89          {
90              return true;
91          }
92      }
93      return false;
94  }
```

- 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*xrand + bx; y = ax*yrand + 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 #ifdef 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

```
/*
 * File:   solver.h
 * Author: oglandx
 *
 * Created on February 17, 2016, 1:54 AM
 */

#ifndef CIRCLESFIGUREAREA_SOLVER_H
#define CIRCLESFIGUREAREA_SOLVER_H

#include "types.h"
#include <vector>
#include <stdlib.h>
#include <pthread.h>
#include <iostream>
#include <chrono>


T solve(const std::vector< Circle* > *circles, int density, RandomFunction random,
    ↪ int max_cores,
        double *result_time = NULL);
std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
        const std::vector< Circle* > *circles, const Rect *rect, int density,
    ↪ RandomFunction random, int max_cores,
        double *result_time = NULL);
Rect *getFigureRect(const std::vector< Circle* > *circles);
bool isPointInsideCircles(const std::vector< Circle* > *circles,
                          const Point *point);


#endif //CIRCLESFIGUREAREA_SOLVER_H
```

- solve.cpp

```
#include "solver.h"

T solve(const std::vector< Circle* > *circles, int density, RandomFunction random,
    ↪ int max_cores, double *result_time)
{
    Rect *rect = getFigureRect(circles);
    std::pair<unsigned long, unsigned long> *result =
            generateAndCheckPoints(circles, rect, density, random, max_cores,
    ↪ result_time);

    if(NULL == result)
    {
        return (T)(-1);
    }

    unsigned long count = result->first;
    unsigned long intersects = result->second;

    return static_cast<T>(intersects)/static_cast<T>(count)* rect->area();
}

void *generatePointsThread(void *params)
{
    PointGeneratorStruct *data = (PointGeneratorStruct*)params;
    data->result = 0;
    for(unsigned long i = 0; i < data->count; ++i)
    {
        Point* point = data->random(data->rect, data->density);
        if(isPointInsideCircles(data->circles, point))
        {
            ++(data->result);
        }
        delete point;
    }
    return NULL;
}

std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
        const std::vector< Circle* > *circles, const Rect *rect, int density,
    ↪ RandomFunction random, int max_cores,
        double *result_time)
{
    if(max_cores < 1)
```

```
41      {
42          return NULL;
43      }
44
45      T area = rect->area();
46      unsigned long count = density*static_cast<unsigned long>(area);
47
48      std::chrono::milliseconds start_time =
49              std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
        ↪ system_clock::now().time_since_epoch());
50      std::cout << "$start_generation...." << std::endl;
51
52      PointGeneratorStruct *params[max_cores];
53
54      pthread_t threads[max_cores - 1];
55      for(int i = 0; i < max_cores - 1; ++i)
56      {
57          params[i] = new PointGeneratorStruct(count/max_cores, density, rect, random,
        ↪ circles);
58          pthread_create(
59                  &threads[i],
60                  NULL,
61                  generatePointsThread,
62                  params[i]
63          );
64      }
65
66      unsigned long real_count = count - count*(max_cores - 1)/max_cores;
67      params[max_cores - 1] =
68              new PointGeneratorStruct(real_count + real_count % max_cores, density,
        ↪ rect, random, circles);
69      generatePointsThread(params[max_cores - 1]);
70
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::
        ↪ system_clock::now().time_since_epoch());
88      double time = static_cast<double>(end_time.count() - start_time.count())/1000;
89
90      std::cout << "end_generation (time = " << time << "s)" << std::endl;
91
92      if(NULL != result_time)
93      {
94          *result_time = time;
95      }
96
97      return new std::pair<unsigned long, unsigned long>(count, result);
98  }
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 !=
    ↪ circles->end(); it++)
140     {
141         if((*it)->containsPoint(point))
142         {
143             return true;
144         }
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->
    ↪ end(); it++)
21     {
22         std::cout << "Circle(" << i++ << ") :: r=" << (*it)->radius <<
23         ", center=(" << (*it)->center->x << "," << (*it)->center->y << ")" << std
    ↪ ::endl;
24     }
25     std::cout << std::endl << std::endl;
26 }
27 
28 template<typename __type>
29 void print_statistics(const std::string &name, __type results[], int count)
30 {
31     std::cout.precision(5);
32 
33     std::cout << "[" << name << "]" << std::endl;
```

```
34
35      __type mean = results[0];
36      for(int i = 1; i < count; ++i)
37      {
38          mean += results[i];
39      }
40      mean /= count;
41
42      std::cout << "Mean value = " << mean << std::endl;
43
44      auto sqr = [](__type x){return x*x;};
45
46      __type disp = sqr(results[0] - mean);
47
48      for(int i = 1; i < count; ++i)
49      {
50          disp += sqr(results[i] - mean);
51      }
52      disp /= count;
53
54      std::cout << "Dispersion = " << disp << std::endl << std::endl;
55  }
56
57  void test_statistics(std::vector< Circle* >  *circles)
58  {
59      int tests_count = def::TESTS_COUNT;
60
61      int world_size;
62      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
63
64      int mpi_rank;
65      MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
66
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 <<
        ↪ std::endl;
77      }
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::
        ↪ system_clock::now().time_since_epoch());
86
87          results[i] = solve(circles, def::DENSITY, &random_uniform_real_distribution,
        ↪ world_size);
88
89          std::chrono::milliseconds end_time =
90                  std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
        ↪ system_clock::now().time_since_epoch());
91          times[i] = static_cast<double>(end_time.count() - start_time.count())/1000;
92
93          std::cout << "end_generation (process = " << mpi_rank << ", time = " << times
        ↪ [i] << "s)" << std::endl;
94
95          if((T)(-1) != results[i])
96          {
97              std::cout.precision(10);
98              std::cout << "Result[" << i << "] = " << results[i] << std::endl << std::
        ↪ endl;
99          }
100     }
101     if((T)(-1) != results[0]){
102         std::cout << std::endl;
103         print_statistics<T>(std::string("Result"), results, tests_count);
```

```
104     }
105
106     if(mpi_rank == 0){
107         double full_times[tests_count*world_size];
108         memcpy(full_times, times, sizeof(double)*tests_count);
109
110         MPI_Status status;
111         for(int rank = 1; rank < world_size; ++rank)
112         {
113             MPI_Recv(times, tests_count, MPI_DOUBLE, rank, 4, MPI_COMM_WORLD, &status
    ↪ );
114             memcpy(full_times + tests_count*rank, times, sizeof(double)*tests_count);
115         }
116         print_statistics<double>(std::string("Time"), times, tests_count);
117     }
118     else
119     {
120         MPI_Send(times, tests_count, MPI_DOUBLE, 0, 4, MPI_COMM_WORLD);
121     }
122     std::cout << std::endl;
123 }
124
125 void run_tests(std::vector< Circle* > *circles){
126     test_statistics(circles);
127 }
128
129 int main(int argc, char *argv[]) {
130     int status = -1;
131
132     std::vector< Circle* > *circles = NULL;
133     const std::string filename("/home/oglandx/circles.txt");
134
135     try{
136         circles = parseCirclesFromFile(filename, def::MAX_CIRCLES_COUNT);
137     }
138     catch (FileException e)
139     {
140         std::cout << e.what();
141     }
142
143     // ----- MPI section -----
144
145     int rc = MPI_Init(&argc, &argv);
146
147     if(rc)
148     {
149         std::cout << "MPI error occurred" << std::endl;
150         MPI_Abort(MPI_COMM_WORLD, rc);
151         return status;
152     }
153
154     int world_size;
155     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
156
157     if(NULL != circles)
158     {
159         #ifdef TEST
160
161         run_tests(circles);
162
163         #else
164
165         T result = solve(circles, def::DENSITY, &random_uniform_real_distribution,
    ↪ world_size);
166         if((T)(-1) != result){
167             std::cout << "Result = " << result << std::endl;
168         }
169
170         #endif
171         status = 0;
172     }
173
174     MPI_Finalize();
175
176     // ----- /MPI -----
177
```

```
178     for(unsigned long i = 0; circles != NULL && i < circles->size(); ++i)
179     {
180         delete circles->at(i);
181     }
182     delete circles;
183 
184     return status;
185 }
```

- 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,
      ↪ int max_cores);
19 std::pair<unsigned long, unsigned long> *generateAndCheckPoints(
20          const std::vector< Circle* > *circles, const Rect *rect, int density,
      ↪ RandomFunction random, int max_cores);
21 Rect *getFigureRect(const std::vector< Circle* > *circles);
22 bool isPointInsideCircles(const std::vector< Circle* > *circles,
23                           const Point *point);
24
25
26 #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,
      ↪ int max_cores)
6 {
7     Rect *rect = getFigureRect(circles);
8     std::pair<unsigned long, unsigned long> *result = generateAndCheckPoints(circles,
      ↪  rect, density, random, max_cores);
9     if(NULL == result)
10    {
11        return (T)(-1);
12    }
13
14    unsigned long count = result->first;
15    unsigned long intersects = result->second;
16
17    return static_cast<T>(intersects)/static_cast<T>(count) * rect->area();
18 }
19
20 unsigned long generatePointsProcess(const std::vector< Circle* > *circles,
21                     unsigned long count, int density, const Rect rect, RandomFunction
      ↪  random)
22 {
23    unsigned long result = 0;
24    for(unsigned long i = 0; i < count; ++i)
25    {
26        Point* point = random(rect, density);
27        if(isPointInsideCircles(circles, point))
28        {
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,
      ↪ RandomFunction random, int max_cores)
49 {
50    if(max_cores < 1)
51    {
52        return NULL;
53    }
54
55    T area = rect->area();
56    unsigned long count = density*static_cast<unsigned long>(area);
57
58    unsigned long result;
59
60    // ---- MPI section ----
61
62    int mpi_rank;
63    const int rank_main = 0;
64
65    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
66
67    if(mpi_rank == rank_main)
68    {
69        unsigned long real_count = count - count*(max_cores - 1)/max_cores;
70        result = generatePointsProcess(circles, real_count, density, rect, random);
71    }
72    else
73    {
```

```
74          result = generatePointsProcess(circles, count/max_cores, density, rect,
        ↪ random);
75          sendResult(result, rank_main);
76          return NULL;
77      }
78
79      for(int rank = 1; rank < max_cores; ++rank)
80      {
81          unsigned long current_result;
82          recvResult(&current_result, rank);
83          result += current_result;
84      }
85
86      // ----- /MPI -----
87
88      return new std::pair<unsigned long, unsigned long>(count, result);
89  }
90
91  Rect *getFigureRect(const std::vector< Circle* > *circles)
92  {
93      typename std::vector< Circle* >::const_iterator it = circles->begin();
94
95      Point *min = new Point(
96              (*it)->center->x - (*it)->radius,
97              (*it)->center->y - (*it)->radius
98      );
99      Point *max = new Point(
100             (*it)->center->x + (*it)->radius,
101             (*it)->center->y + (*it)->radius
102     );
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 !=
        ↪ circles->end(); it++)
131     {
132         if((*it)->containsPoint(point))
133         {
134             return true;
135         }
136     }
137     return false;
138 }
```