МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
Высшего образования
«Нижегородский государственный университет им. Н.И. Лобачевского»
Национальный исследовательский университет

Институт информационных технологий, математики и механики
Кафедра математического обеспечения и суперкомпьютерных технологий

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

«Маркировка компонент на бинарном изображении»

Выполнил:

студент группы 381706-1
Корнев Никита Алексеевич
_____ Подпись

Принял:

Доцент кафедры МОСТ, кандидат технических наук

_____Сысоев А. В.

Нижний Новгород

2019.

Содержание

Введение

Бинарное изображение (двухуровневое, двоичное) — разновидность цифровых растровых изображений, когда каждый пиксел может представлять только один из двух цветов.

Значения каждого пиксела условно кодируются как «0» и «1». Значение «0» условно называют задним планом или фоном (англ. background), а «1» — передним планом (англ. foreground).

В данной программе бинарное изображение будет представлено в виде отдельного класса, поля которого содержат ширину, высоту изображения и двумерный целочисленный массив, необходимый для хранения данных изображения.

Задача маркировки компонент на бинарном изображении предполагает выделение связных объектов и последующую маркировку уникальным индексом (цветом). Таким образом, результатом работы программы, получившей на вход бинарное изображение, будет размеченное изображение (матрица, элементами которой являются «0» и индексы).

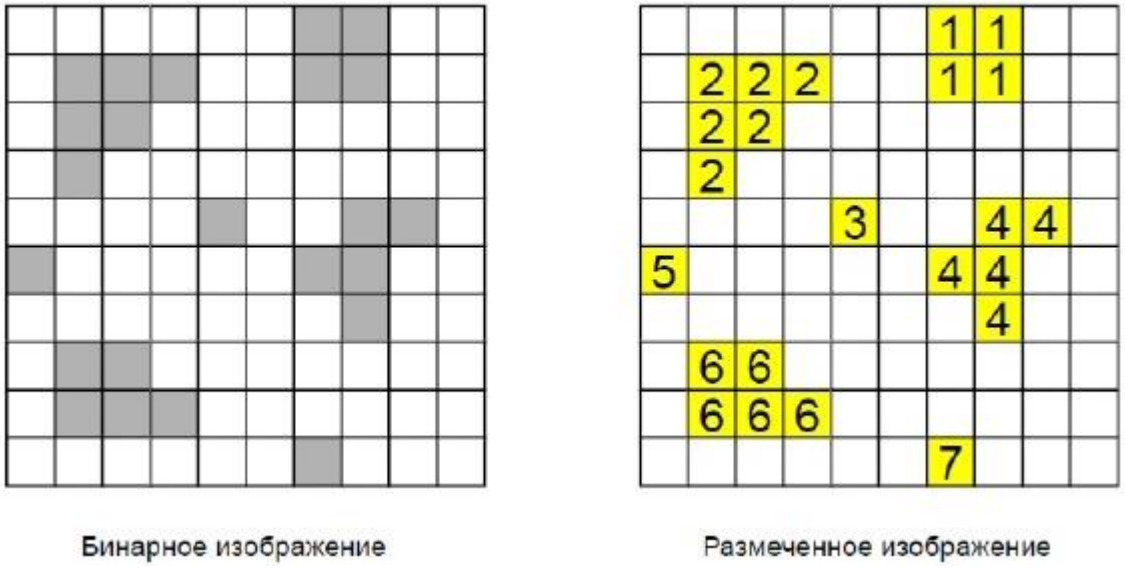

Рисунок 1. Результат работы алгоритма

Постановка задачи

В данной лабораторной работе необходимо выполнить следующие задачи:

1. Разработать алгоритм параллельной маркировки бинарного изображения некоторым числом процессов.

2. Реализовать класс, представляющий бинарное изображение.

3. Написать программу, реализующую данный алгоритм.

4. Написать тесты для проверки работы алгоритма.

5. Провести вычислительные эксперименты.

6. Проанализировать полученные результаты.

Описание алгоритмов

Алгоритм маркировки:

1. Если индекс условного пикселя (элемента матрицы) равен «1», маркируем его, то есть присваиваем уникальный индекс.

2. Если индекс пикселя, расположенного под найденным, равен «1» маркируем его тем же индексом, двигаемся вниз, пока не найдём пиксель с индексом «0».

3. При нахождении каждого пикселя ниже предыдущего, проходим по пикселям слева и справа от него, пока не найдём «0». Все найденные пиксели маркируем.

4. Если все пиксели вокруг первоначального исследованы, увеличиваем счетчик фигур на 1.

Программную реализацию данного алгоритма можно найти в приложении.

Схема распараллеливания

Принцип распараллеливания в данной программе заключается в том, что бинарное изображение разбивается на некоторое число областей, зависящее от числа процессов, затем эти области передаются процессам для маркировки, после чего, размеченные области возвращаются корневому процессу. Он проходит по границам областей, размечая фигуры, расположенные на границах, одним из индексов. Здесь появляется проблема, заключающаяся в том, что число процессов не всегда пропорционально размеру изображения, так что нужно вычислить остаток и увеличить область, обрабатываемую нулевым процессом, чтобы всё изображение было промаркировано. Рассмотрим алгоритм более подробно:

1. Бинарное изображение, представляющее собой матрицу, трансформируется в строку нулевым процессом.

2. Строка разбивается на подстроки некоторой длины, подстроки передаются нулевым процессом всем остальным процессам.

3. Процесс, получив подстроку, преобразует её в локальное бинарное изображение.

4. Каждый процесс проводит маркировку своего изображения.

5. Затем преобразует изображение назад в подстроку.

6. Передает сформированную подстроку нулевому процессу.

7. Нулевой процесс формирует из полученных подстрок строку и строит новое изображение.

8. Затем проходит по всем границам областей, если находит связные фигуры на границе разных областей, маркирует их одним из индексов.

Программную реализацию данного алгоритма можно найти в приложении.

Подтверждение корректности

Для проверки корректности результата работы алгоритма сравним значения заданного количества фигур и полученное значение счетчика. Таким образом, можно сделать вывод, что алгоритм нашел и промаркировал все заданные фигуры на изображении. Для наглядности реализован небольшой алгоритм, выводящий полученное изображение на экран, так что наблюдатель может убедиться, что изображение размечено правильно.

В программе реализованы тесты на основе Google tests для проверки корректности работы алгоритма. Для каждого теста задано некоторое бинарное изображение. Результаты тестов выглядят следующим образом:

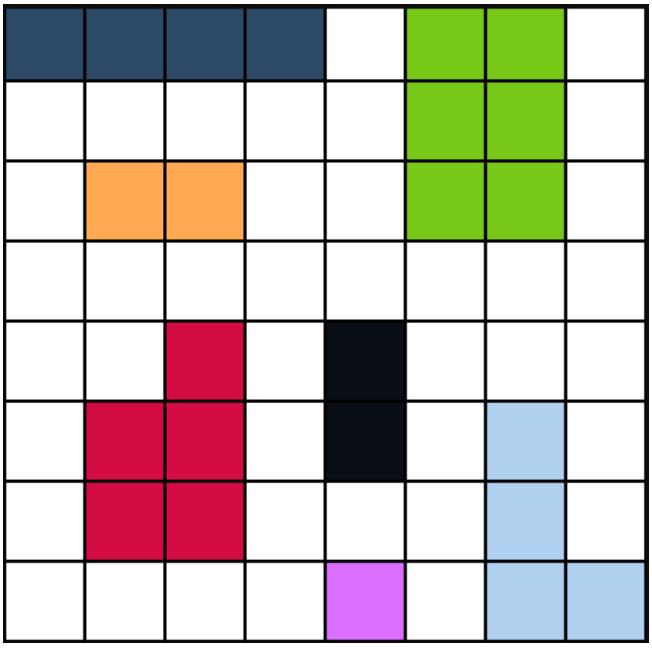

Рисунок 2. Тест 1

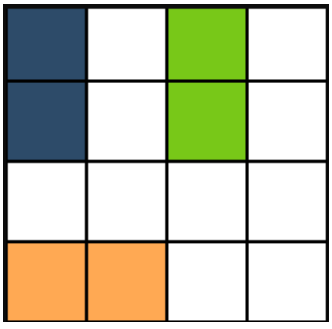

Рисунок 3. Тест 2

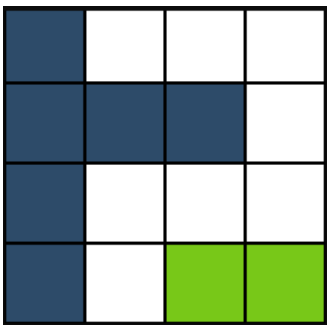

Рисунок 4. Тест 3

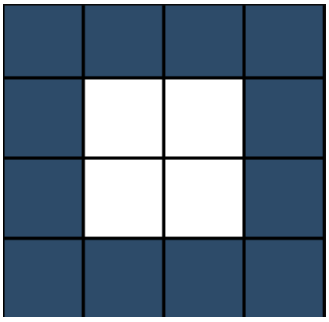

Рисунок 5. Тест 4

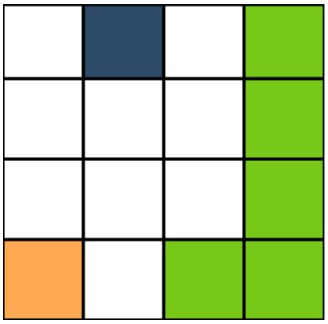

Рисунок 6. Тест 5

Эксперименты

Эксперименты проводились на ПК со следующими характеристиками:

- Процессор: AMD Ryzen 3 3200U
- Оперативная память: 4Gb DDR4 2133 GHz
- Операционная система: Windows 10 Домашняя
- Microsoft Visual Studio 2019
- MPI 10.0

Задано случайное бинарное изображение. Проведем замеры времени работы алгоритма на различном числе процессов и получим среднее.

Число процессов / сторона изображения	1	2	4	8
1024	0,027071 с	0,017335 с	0,016961 с	0,019742 с
2048	0,112905 с	0,081043 с	0,071152 с	0,064605 с
4096	0,512412 с	0,356986 с	0,296069 с	0,264022 с

При небольшом размере изображения 1024 * 1024 наблюдается прирост производительности в 1,562 раза при выполнении на 2 процессах, в 1,596 при выполнении на 4. Выполнение на 8 процессах дает прирост в 1,371 раз. Дальнейшее увеличение числа процессов не дает прироста производительности, напротив, увеличивает время выполнения алгоритма. Это объясняется большим возрастанием накладных расходов на разбиение изображения на области и их рассылку всем процессам. Таким образом, оптимальным числом процессов для этого алгоритма на малом изображении является 4 процесса.

При среднем размере изображения 2048 * 2048 наблюдается прирост производительности в 1,393 раза при выполнении на 2 процессах, в 1,586 при выполнении на 4. Выполнение на 8 процессах дает прирост в 1,748 раз. Дальнейшее увеличение числа процессов не дает прироста производительности. Таким образом, оптимальным числом процессов для этого алгоритма на среднем изображении является 8 процессов.

При большом размере изображения 4096 * 4096 наблюдается прирост производительности в 1,435 раза при выполнении на 2 процессах, в 1,731 при выполнении на 4. Выполнение на 8 процессах дает прирост в 1,942 раз. Дальнейшее увеличение числа процессов не дает прироста производительности. Таким образом, оптимальным числом процессов для этого алгоритма на малом изображении является 8 процессов.

Таким образом, в среднем выполнение на 2 процессах дает ускорение в 1,528 раз, на 4 в 1,637 раз, а на 8 в 1,687 раз. Дальнейшее увеличение числа процессов не имеет смысла, так как сильно возрастают накладные расходы, замедляющие работу алгоритма. Можно сделать вывод, что для небольших изображений оптимальнее использовать 4 процесса, для остальных – 8.

Заключение

В результате данной лабораторной работы успешно выполнены следующие действия:

1. Разработан и применен алгоритм «labelling», предназначенный для параллельной маркировки бинарного изображения некоторым числом процессов.

2. Реализован класс «image» с полями m (высота), n (ширина) и data, а также перегруженным оператором сдвига «<<», представляющий бинарное изображение.

3. Написана программа, успешно реализующая данный алгоритм маркировки на практике.

4. Разработаны и реализованы Google-тесты для проверки корректности работы алгоритма, результаты их работы приведены выше.

5. Проведены вычислительные эксперименты, получено среднее время выполнения алгоритма на разном числе процессов и размерах исходного изображения, полученные результаты занесены в таблицу.

6. Сделаны следующие выводы на основе полученных результатов:

- для малых изображений (со сторонами до 1024) эффективнее задействовать 4 процесса при выполнении алгоритма;

- для всех остальных оптимальным вариантом является 8 процессов.

Таким образом, все поставленные в данной лабораторной работе задачи успешно выполнены.

Литература

1. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. 184 с.

2. Википедия: свободная электронная энциклопедия: на русском языке [Электронный ресурс] - Режим доступа: https://ru.wikipedia.org/wiki/Бинарное_изображение, свободный.

Приложение

Img_labelling.h

```
#ifndef MODULES_TASK_3_KORNEV_N_BIN_IMG_LABELING_BIN_IMG_LABELING_H_
#define MODULES_TASK_3_KORNEV_N_BIN_IMG_LABELING_BIN_IMG_LABELING_H_
// Copyright 2019 Kornev Nikita

#include <iostream>
#include "./mpi.h"

struct image {
  int m, n, count;
  int** data;

  image(int _m, int _n) : m(_m), n(_n), count(1) {
    data = new int*[m];
    for (int i = 0; i < m; i++) {
      data[i] = new int[n];
      for (int j = 0; j < n; j++) {
        data[i][j] = 0;
      }
    }
  }

  friend std::ostream& operator<< (std::ostream& os, const image& img) {
    for (int i = 0; i < img.m; i++) {
      for (int j = 0; j < img.n; j++) {
        os << img.data[i][j] << "\t";
      }
      os << std::endl;
    }
    return os;
  }
};

void labeling(image* img);
void draw(image* img);

#endif  // MODULES_TASK_3_KORNEV_N_BIN_IMG_LABELING_BIN_IMG_LABELING_H_
```

Img_labelling.cpp

```
// Copyright 2019 Kornev Nikita

#include "../../../modules/task_3/kornev_n_bin_img_labeling/bin_img_labeling.h"
#include <Windows.h>
#include <cmath>
#include <iostream>
using namespace std;

COLORREF color(int label) {
  int hash = pow(label, 3) + pow(label, 2) + label + 1;
  return RGB(3 * hash % 256, 5 * hash % 256, 7 * hash % 256);
}

void labeling(image* img) {
  int rank, size, label, res = 0, label_count = 0, string_count, new_string_count, rest;
```

```
  bool flag = 0;
  int* global_img_arr, *local_img_arr;
  image* local_img;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Status status;
  string_count = img->m / size;
  rest = img->m % size;

  if (rank == 0) {
    new_string_count = string_count + rest;
  } else {
    new_string_count = string_count;
  }

  label = rank * 100 + 2;

  // alloc mem for recv buf & local_img & res_buf
  global_img_arr = new int[img->m * img->n];
  local_img_arr = new int[new_string_count * img->n];
  local_img = new image(new_string_count, img->n);

  // clone img matrix values into arr & send parts to other ranks
  if (rank == 0) {
    for (int i = 0; i < img->m; i++) {
      for (int j = 0; j < img->n; j++) {
        global_img_arr[i * img->n + j] = img->data[i][j];
      }
    }

    // put first part of img arr in 0-rank's local img
    for (int i = 0; i < (new_string_count) * img->n; i++) {
      local_img_arr[i] = global_img_arr[i];
    }

    // send other parts of img arr to other ranks
    for (int i = 1; i < size; i++) {
      MPI_Send(&global_img_arr[i * img->n * string_count + rest * img->n], string_count
* img->n,
        MPI_INT, i, 0, MPI_COMM_WORLD);
    }
  } else {
    MPI_Recv(&local_img_arr[0], string_count * img->n, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);
  }

  // clone part of img arr into cur rank's img
  for (int i = 0; i < new_string_count; i++) {
    for (int j = 0; j < img->n; j++) {
      local_img->data[i][j] = local_img_arr[i * img->n + j];
    }
  }

  // img labeling
  for (int i = 0; i < local_img->m; i++) {
    for (int j = 0; j < img->n; j++) {
      if (local_img->data[i][j] == 1) {
        flag = 1;
        local_img->data[i][j] = label;
        if (i + 1 < new_string_count && local_img->data[i + 1][j] == 1) {
          int k = i + 1;
          while (k < new_string_count && local_img->data[k][j] == 1) {
            local_img->data[k][j] = label;
            if (j - 1 >= 0 && local_img->data[k][j - 1]) {
              int l = j - 1;
```

```
            while (l >= 0 && local_img->data[k][l]) {
              local_img->data[k][l] = label;
              l--;
            }
          }
          if (j + 1 < img->n && local_img->data[k][j + 1] == 1) {
            int l = j + 1;
            while (l < img->n && local_img->data[k][l]) {
              local_img->data[k][l] = label;
              l++;
            }
          }
          k++;
        }
      }
    } else {
      if (flag) {
        flag = 0;
        label++;
        label_count++;
      }
    }
  }
}

// now clone cur rank's img back to img arr
for (int i = 0; i < new_string_count; i++) {
  for (int j = 0; j < img->n; j++) {
    local_img_arr[i * img->n + j] = local_img->data[i][j];
  }
}

// put 0-ranks's local arr into global img arr
if (rank == 0) {
  for (int i = 0; i < new_string_count * img->n; i++) {
    global_img_arr[i] = local_img_arr[i];
  }
}

// then send local img arrs to 0-rank
if (rank != 0) {
    MPI_Send(&local_img_arr[0], new_string_count * img->n,
      MPI_INT, 0, 0, MPI_COMM_WORLD);
} else {
  for (int i = 1; i < size; i++) {
    MPI_Recv(&global_img_arr[i * img->n * string_count + rest * img->n],
new_string_count * img->n,
      MPI_INT, i, 0, MPI_COMM_WORLD, &status);
  }
}

MPI_Reduce(&label_count, &res, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

// now build new img
if (rank == 0) {
  for (int i = 0; i < img->m; i++) {
    for (int j = 0; j < img->n; j++) {
      img->data[i][j] = global_img_arr[i * img->n + j];
    }
  }

  // borders
  flag = 0;
  for (int i = new_string_count - 1; i < img->m - string_count; i += string_count) {
    for (int j = 0; j < img->n; j++) {
```

```
        if (img->data[i][j] != 0 && img->data[i + 1][j] != 0) {
          flag = 1;
          int k = i + 1;
          while (k < img->m && img->data[k][j] != 0) {
            img->data[k][j] = img->data[i][j];
            if (j > 0 && img->data[k][j - 1] != 0) {
              int l = j - 1;
              while (l >= 0 && img->data[k][l] != 0) {
                img->data[k][l] = img->data[i][j];
                l--;
              }
            }

            if (j < img->n - 1 && img->data[k][j + 1] != 0) {
              int l = j + 1;
              while (l < img->n && img->data[k][l] != 0) {
                img->data[k][l] = img->data[i][j];
                img->data[k][l] = img->data[i][j];
                l++;
              }
            }
            k++;
          }
        } else {
          if (flag) {
            flag = 0;
            res--;
          }
        }
      }
    }
  }
  img->count = res;
}

void draw(image* img) {
  HWND hwnd = GetConsoleWindow(); // Берём ориентир на консольное окно (В нём будем
рисовать)
  HDC dc = GetDC(hwnd); // Цепляемся к консольному окну
  HBRUSH brush, bg = CreateSolidBrush(RGB(255, 255, 255)), black = CreateSolidBrush(0);
// Переменная brush - это кисть, она будет использоваться для закрашивания

  bool flag = 0;
  for (int i = 0; i < img->m; i++) {
    for (int j = 0; j < img->n; j++) {
      if (img->data[i][j] != 0) {
        if (flag == 0) {
          flag = 1;
          if (img->data[i][j] == 1) {
            SelectObject(dc, black);
          }
          else {
            brush = CreateSolidBrush(color(img->data[i][j]));
            SelectObject(dc, brush);
          }
        }
        Rectangle(dc, 50 + j * 50, 175 + i * 50, 100 + j * 50, 225 + i * 50);
      } else {
        if (flag == 1)
          flag = 0;
        SelectObject(dc, bg);
        Rectangle(dc, 50 + j * 50, 175 + i * 50, 100 + j * 50, 225 + i * 50);
      }
    }
```

Main.cpp

```
// Copyright 2019 Kornev Nikita

#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <time.h>
#include <iostream>
#include "./bin_img_labeling.h"
#include <random>

TEST(broadcast, test1) {
  int rank, size, exp_count = 7;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  image img(8, 8);

  if (rank == 0) {
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 2; j++) {
        img.data[0][0] = 1;
        img.data[0][1] = 1;
        img.data[0][2] = 1;
        img.data[0][3] = 1;
        img.data[0][5] = 1;
        img.data[0][6] = 1;
        img.data[1][5] = 1;
        img.data[1][6] = 1;
        img.data[2][5] = 1;
        img.data[2][6] = 1;
        img.data[2][1] = 1;
        img.data[2][2] = 1;
        img.data[4][2] = 1;
        img.data[4][4] = 1;
        img.data[5][2] = 1;
        img.data[5][4] = 1;
        img.data[5][6] = 1;
        img.data[5][1] = 1;
        img.data[6][1] = 1;
        img.data[6][2] = 1;
        img.data[6][6] = 1;
        img.data[7][4] = 1;
        img.data[7][6] = 1;
        img.data[7][7] = 1;
      }
    }
  }

  labeling(&img);

  if (rank == 0) {
    draw(&img);
    ASSERT_EQ(exp_count, 7);
  }
}

TEST(broadcast, test2) {
  int rank, size, exp_count = 3;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  image img(4, 4);

  if (rank == 0) {
    img.data[0][0] = 1;
```

```
    img.data[1][0] = 1;
    img.data[3][0] = 1;
    img.data[0][2] = 1;
    img.data[0][2] = 1;
    img.data[1][2] = 1;
    img.data[3][1] = 1;

  }

  labeling(&img);

  if (rank == 0) {
     draw(&img);
    ASSERT_EQ(exp_count, 3);
  }
}

TEST(broadcast, test3) {
  int rank, size, exp_count = 1;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  image img(4, 4);

  if (rank == 0) {
    img.data[0][0] = 1;
    img.data[1][0] = 1;
    img.data[2][0] = 1;
    img.data[3][0] = 1;
    img.data[3][2] = 1;
    img.data[3][3] = 1;
    img.data[1][1] = 1;
    img.data[1][2] = 1;
  }

  labeling(&img);

  if (rank == 0) {

    ASSERT_EQ(exp_count, 1);
  }
}

TEST(broadcast, test4) {
  int rank, size, exp_count = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  image img(4, 4);

  if (rank == 0) {
    img.data[0][1] = 1;
    img.data[3][0] = 1;
    img.data[3][3] = 1;
    img.data[3][2] = 1;
    img.data[0][3] = 1;
    img.data[1][3] = 1;
    img.data[2][3] = 1;
  }

  labeling(&img);

  if (rank == 0) {
     draw(&img);
    ASSERT_EQ(exp_count, 0);
  }
}
```

```
TEST(broadcast, test5) {
  int rank, size, exp_count = 2;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  image img(4, 4);

  if (rank == 0) {
    img.data[0][0] = 1;
    img.data[0][1] = 1;
    img.data[0][2] = 1;
    img.data[0][3] = 1;
    img.data[1][0] = 1;
    img.data[1][3] = 1;
    img.data[2][0] = 1;
    img.data[2][3] = 1;
    img.data[3][0] = 1;
    img.data[3][1] = 1;
    img.data[3][2] = 1;
    img.data[3][3] = 1;
  }

  labeling(&img);

  if (rank == 0) {
     draw(&img);
    ASSERT_EQ(exp_count, 2);
  }
}


int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  MPI_Init(&argc, &argv);

  ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
  ::testing::TestEventListeners& listeners =
    ::testing::UnitTest::GetInstance()->listeners();

  listeners.Release(listeners.default_result_printer());
  listeners.Release(listeners.default_xml_generator());

  listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
  return RUN_ALL_TESTS();
}
```