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

«Линейная фильтрация изображений (горизонтальное разбиение). Ядро Гаусса 3x3.»

Выполнил:

студент группы 381706-1
Соколов Андрей Дмитриевич
_____ Подпись

Принял:

Доцент кафедры МОСТ, кандидат технических наук

_____Сысоев А. В.

Нижний Новгород

2019.

Содержание

1. Введение 3
2. Постановка задачи 4
3. Описание алгоритмов 5
4. Схема распараллеливания 7
5. Описание MPI-версии 8
6. Эксперименты 9
7. Заключение 11
8. Литература 12

1. Введение

Линейная фильтрация изображения с помощью фильтра Гаусса – алгоритм обработки изображения в конце работы которого получается размытое изображение без шумов.

Рисунок 1. Пример использования фильтра гаусса для изображения в оттенках серого

2. Постановка задачи

В этой лабораторной работе нужно реализовать алгоритм линейной фильтрации изображения с помощью ядра Гаусса 3x3. Изначально нам даётся изображение в оттенках серого, размер пикселя - 1 байт. На выходе должно получаться изображение таких же размеров с размытием и устраненным шумом.

Для более эффективной работы алгоритма необходимо реализовать горизонтальное разбиение изображения для реализации параллельной работы алгоритма с произвольным количеством процессов.

3. Описание алгоритмов

Линейная фильтрация изображения с помощью фильтра Гаусса 3x3

Реализация алгоритма линейной фильтрации изображения заключается в последовательном применении ядра Гаусса к каждому пикселю.

Ядро, или матрица свёртки, Гаусса в нашей задаче – матрица размера 3 на 3

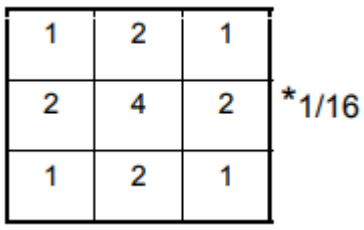

Рисунок 2. Ядро Гаусса

Это матрица коэффициентов, которая «умножается» на значение пикселей изображения для получения требуемого результата. Число «16» - это коэффициент нормирования, для того чтобы средняя интенсивность оставалась не изменой. Матрица свертки «Накладывается» на каждый вычисляемый пиксель изображения, которому соответствует середине ядра – 4 и считается новое значение за счет своего значения и 8-ми соседних пикселей.

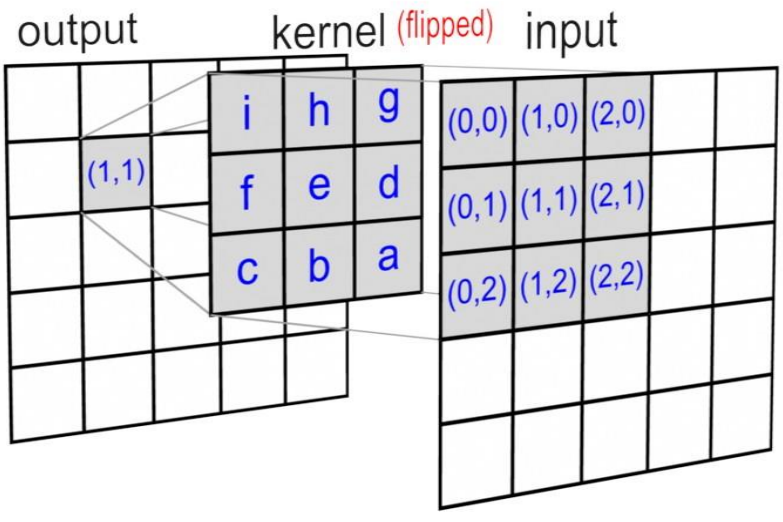

Рисунок 3. Пример «накладывания» матрицы свёртки на изображение

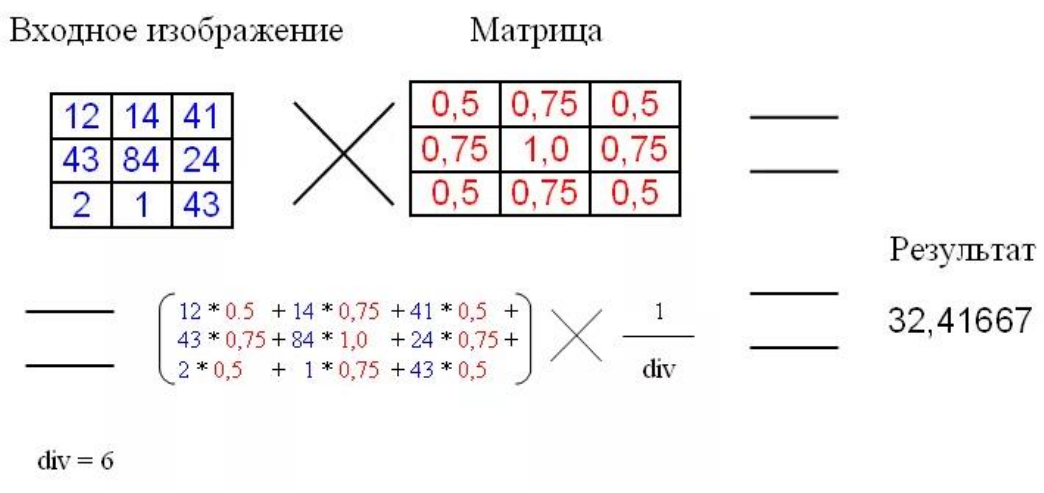

Рисунок 4. Пример вычисления значения пикселя с помощью матрицы свёртки 3x3

Так же в работе алгоритма есть особые пиксели, находящиеся на границе изображения. В таких случаях изображение расширяется за счет собственных значений и свертывание происходит как обычно.

4. Схема распараллеливания

Организация параллельной работы алгоритма происходит путём горизонтального разбиения изображения на n блоков, где n – количество процессов. Количество строк в этих блоках считаются по формулам:

$$n1 = rows / comm\_size$$
$$n2 = rows \% comm\_size \quad (1)$$

Нулевой процесс будет вычислять часть изображения размером n1 + n2 строк, а все остальные процессы – n1 строк.

Особенность распараллеливания нашего алгоритма заключается в том, что рассылать процессам изображение нужно с дополнительными строками:

- 2, если блок исходного изображения не является последним
- 1, если блок исходного изображения является последним

Данное дополнение необходимо в следствии вычисления пикселя с помощью матрицы свертки 3x3, которая задействует 8 соседних пикселей

5. Описание MPI-версии

Для реализации параллельной работы алгоритма с помощью библиотеки MPI используются стандартные функции обмена сообщениями между процессами.

В начале работы алгоритма используются функции MPI_Comm_size что бы узнать сколько процессов используется и MPI_Comm_rank что бы инициализировать ранг процесса.

Дальше нам необходимо разослать из нулевого процесса блоки исходного изображения ненулевым процессам размером, соответствующим нашей схемой распараллеливания. Это происходит с помощью функций MPI_Send и MPI_Recv.

После вычисления значений пикселей в каждом процессе программа должна разослать полученный результат обратно нулевому процессу, чтобы собрать новое изображение воедино. Это также происходит с помощью функций MPI_Send и MPI_Recv.

6. Эксперименты

Эксперименты проводились на ПК с следующими параметрами:

Операционная система: Windows 10

Процессор: Intel(R) Core™ i5-6200U CPU @ 2.30 GHz

Оперативная память: 8,00 Gb

Эксперимент 1

В этом эксперименте проверяется работа алгоритма в зависимости от количества процессов. Размер изображения 500 x 500 пикселей.

Количество процессов	Время работы последовательного алгоритма	Время работы параллельного алгоритма
1	2.55949	3.29941
2	2.54079	1.76754
4	2.36184	0.70132
8	2.58534	0.42025

Таблица 1. Время работы алгоритма с фиксированным размером изображения.

Эксперимент 2

В этом эксперименте проверяется работа алгоритма в зависимости от размера изображения. Количество процессов - 4. Количество столбцов - 100

Количество строк	Время работы последовательного алгоритма	Время работы параллельного алгоритма
10	0.000694	0.000749
50	0.0007172	0.0005217
100	0.0041704	0.002
500	2.33967	0.703265

Таблица 2. Время работы алгоритма с фиксированным количеством процессов.

Исходя из двух экспериментов можно увидеть, что параллельный алгоритм работает в разы эффективней последовательного. На малых размерах изображения видно, что параллельный алгоритм менее эффективен в следствии относительно больших затрат на пересылку между процессами. Но на больших размерах эго эффективность очевидна

7. Заключение

В ходе работы над данной лабораторной работой был реализован последовательный и параллельный алгоритм линейной фильтрации изображения с помощью ядра Гаусса 3 x 3. Был подробно изучен алгоритм обработки изображения с помощью матрицы свертки и получен опыт по распараллеливанию алгоритмов.

8. Литература

1. Википедия: свободная электронная энциклопедия на русском языке: https://ru.wikipedia.org

9. Приложение

Horizontal_gauss.h

```
// Copyright 2019 Sokolov Andrey

#ifndef MODULES_TASK_3_SOKOLOV_A_HORIZONTAL_GAUSS_HORIZONTAL_GAUSS_H_

#define MODULES_TASK_3_SOKOLOV_A_HORIZONTAL_GAUSS_HORIZONTAL_GAUSS_H_

#include <vector>


// #define DEBUG


constexpr unsigned char gaussFilter[3][3] {{1, 2, 1},
                                           {2, 4, 2},
                                           {1, 2, 1}};


constexpr unsigned char sumMask {16};


std::vector<unsigned char> getRandomImage(int _cols, int _rows);


unsigned char changePixel(std::vector<unsigned char> _source, int x, int y, int rows,
int cols);


std::vector<unsigned char> filterImageParallel(std::vector<unsigned char> source, int
rows, int cols);

std::vector<unsigned char> filterImageSequential(std::vector<unsigned char> source, int
rows, int cols);


#endif  // MODULES_TASK_3_SOKOLOV_A_HORIZONTAL_GAUSS_HORIZONTAL_GAUSS_H_
```

Horizontal_gauss.cpp

```
// Copyright 2019 Sokolov Andrey
#include <mpi.h>
#include <iostream>
#include <random>
#include <numeric>
#include <algorithm>
#include <vector>
#include <ctime>
#include <list>
#include "../../../modules/task_3/sokolov_a_horizontal_gauss/horizontal_gauss.h"

std::vector<unsigned char> getRandomImage(int _cols, int _rows) {
  std::mt19937 gen;
  gen.seed(static_cast<unsigned int>(time(0)));
  std::vector<unsigned char> image(_cols * _rows);
  for (int i = 0; i < _cols; i++)
    for (int j = 0; j < _rows; j++)
      image[i*_rows+j] = static_cast<unsigned char>(gen() % 256);

  return image;
}
```

```
unsigned char changePixel(std::vector<unsigned char> source, int _x, int _y, int rows,
int cols) {
  int sum = 0;
  for (int i = -1; i < 2; ++i) {
    for (int j = -1; j < 2; ++j) {
      int x = _x + i;
      int y = _y + j;

      if (x < 0 || x > rows - 1) {
        x = _x;
      }
      if (y < 0 || y > cols - 1) {
        y = _y;
      }
      if (x * cols + y >= cols * rows) {
        x = _x;
        y = _y;
      }
      sum += static_cast<int>(source[x*cols + y] * (gaussFilter[i + 1][j + 1]));
    }
  }
  return sum / sumMask;
}

std::vector<unsigned char> filterImageSequential(std::vector<unsigned char> source, int
rows, int cols) {
  std::vector<unsigned char> result(cols * rows);
  for (int i = 0; i < rows; ++i)
    for (int j = 0; j < cols; ++j) {
      result[i * cols + j] = changePixel(source, i, j, rows, cols);
    }
  return result;
}

std::vector<unsigned char> filterImageParallel(std::vector<unsigned char> source, int
rows, int cols) {
  int comm_size, rank;
  MPI_Status status;

  std::vector<unsigned char> globalResult(cols * rows);

  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rows < comm_size) {
    return filterImageSequential(source, rows, cols);
  }

  const int interval      = rows / comm_size;  // how many lines will send to processes
!= 0
  const int last_interval = rows % comm_size;  // residue to 0'th process

  std::vector<std::vector<unsigned char>> recvResult(comm_size - 1);

  if (rank == 0) {
#ifdef DEBUG
    std::cout << "Interval: " << interval << std::endl;
    std::cout << "Last Interval: " << last_interval << std::endl;
#endif  // DEBUG

    for (int i = 0; i < comm_size - 1; ++i) {
      recvResult[i].resize(interval * cols, 0);
    }
  }
```

```
  std::vector<unsigned char> localImage(interval * cols + 2 * cols);
  std::vector<unsigned char> localResult(interval * cols);

  if (rank == 0) {
#ifdef DEBUG
    std::cout << "Source: " << "Size: " << source.size() << std::endl;
    for (int i = 0; i < source.size(); ++i) {
      std::cout << (unsigned int)source[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
#endif  // DEBUG

    for (int proc = 1; proc < comm_size; proc++) {
      int startAdress = (proc * interval * cols) + (last_interval - 1) * cols;
      int countSend;

      if (proc != (comm_size - 1)) {
        countSend = interval * cols + 2 * cols;
      } else {
        countSend = interval * cols + cols;
      }
      MPI_Send(&source[0] + startAdress, countSend, MPI_UNSIGNED_CHAR, proc, 0,
MPI_COMM_WORLD);
    }

  } else if (rank != 0) {
    if (rank != comm_size - 1) {
      MPI_Recv(&localImage[0], (interval + 2) * cols + 2, MPI_UNSIGNED_CHAR, 0, 0,
MPI_COMM_WORLD, &status);
    } else {
      localImage.resize((interval + 1) * cols);
      MPI_Recv(&localImage[0], (interval + 1) * cols, MPI_UNSIGNED_CHAR, 0, 0,
MPI_COMM_WORLD, &status);
    }

#ifdef DEBUG
    std::cout << "{" << rank << "}" << "Local Image" << " | Size: " << localImage.size()
<< std::endl;
    for (int i = 0; i < localImage.size(); ++i) {
      std::cout << (unsigned int)localImage[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
#endif  // DEBUG
  }

  if (rank == comm_size - 1 && comm_size != 1) {
    for (int i = 1; i < interval + 1; ++i)
      for (int j = 0; j < cols; ++j)
        localResult[(i - 1) * cols + j] = changePixel(localImage, i, j, interval + 1,
cols);

#ifdef DEBUG
    std::cout << "{" << rank << "}" << "Local result" << " | Size: " <<
localResult.size() << std::endl;
    for (int i = 0; i < localResult.size(); ++i) {
      std::cout << (unsigned int)localResult[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
#endif  // DEBUG
  } else if (rank != 0) {
    for (int i = 1; i < interval + 1; ++i)
```

```
      for (int j = 0; j < cols; ++j)
        localResult[(i - 1) * cols + j] = changePixel(localImage, i, j, interval + 2,
cols);

#ifdef DEBUG
    std::cout << "{" << rank << "}" << "Local result" << " | Size: " <<
localResult.size() << std::endl;
    for (int i = 0; i < localResult.size(); ++i) {
      std::cout << (unsigned int)localResult[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
#endif  // DEBUG
  } else {
    for (int i = 0; i < interval + last_interval; ++i)
      for (int j = 0; j < cols; ++j)
        globalResult[i * cols + j] = changePixel(source, i, j, rows, cols);
  }

  if (rank != 0) {
    MPI_Send(&localResult[0], interval * cols, MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD);
  } else {
    for (int proc = 1; proc < comm_size; ++proc) {
      int start = ((interval + last_interval) * cols) + ((proc - 1) * interval * cols);
      MPI_Recv(&globalResult[0] + start, interval * cols, MPI_UNSIGNED_CHAR, proc, 0,
MPI_COMM_WORLD, &status);
    }
  }

  return globalResult;
}
```

main.cpp

```
// Copyright 2019 Sokolov Andrey
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <iostream>
#include <vector>
#include <array>
#include "../../../modules/task_3/sokolov_a_horizontal_gauss/horizontal_gauss.h"

TEST(Horizontal_Gauss_MPI, Test_Image_9_rows_9_cols) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  constexpr int rows{ 9 };
  constexpr int cols{ 9 };

  std::vector<unsigned char> src = getRandomImage(cols, rows);;
  std::vector<unsigned char> resPar;
  std::vector<unsigned char> resSeq;

  resPar = filterImageParallel(src, cols, rows);

  if (rank == 0) {
    resSeq = filterImageSequential(src, cols, rows);
#ifdef DEBUG
    std::cout << "Parallel result:" << std::endl;
    for (int i = 0; i < cols * rows; ++i) {
      std::cout << (unsigned int)resPar[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "Sequential result:" << std::endl;
```

```
    for (int i = 0; i < cols* rows; ++i) {
      std::cout << (unsigned int)resSeq[i] << " ";
    }
#endif  // DEBUG
    ASSERT_EQ(resPar, resSeq);
  }
}

TEST(Horizontal_Gauss_MPI, Test_Image_27rows_27_cols) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  constexpr int rows{ 27 };
  constexpr int cols{ 27 };

  std::vector<unsigned char> src = getRandomImage(cols, rows);;
  std::vector<unsigned char> resPar;
  std::vector<unsigned char> resSeq;

  resPar = filterImageParallel(src, cols, rows);

  if (rank == 0) {
    resSeq = filterImageSequential(src, cols, rows);
#ifdef DEBUG
    std::cout << "Parallel result:" << std::endl;
    for (int i = 0; i < cols * rows; ++i) {
      std::cout << (unsigned int)resPar[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "Sequential result:" << std::endl;
    for (int i = 0; i < cols* rows; ++i) {
      std::cout << (unsigned int)resSeq[i] << " ";
    }
#endif  // DEBUG

    ASSERT_EQ(resPar, resSeq);
  }
}

TEST(Horizontal_Gauss_MPI, Test_Image_10_rows_5_cols) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  constexpr int rows{ 10 };
  constexpr int cols{ 5 };

  std::vector<unsigned char> src = getRandomImage(cols, rows);;
  std::vector<unsigned char> resPar;
  std::vector<unsigned char> resSeq;

  resPar = filterImageParallel(src, rows, cols);

  if (rank == 0) {
    resSeq = filterImageSequential(src, rows, cols);
#ifdef DEBUG
    std::cout << "Parallel result:" << std::endl;
    for (int i = 0; i < cols * rows; ++i) {
      std::cout << (unsigned int)resPar[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
  }
    std::cout << std::endl;
    std::cout << "Sequential result:" << std::endl;
    for (int i = 0; i < cols* rows; ++i) {
      std::cout << (unsigned int)resSeq[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
```

```
    }
#endif  // DEBUG

    ASSERT_EQ(resPar, resSeq);
  }
}

TEST(Horizontal_Gauss_MPI, Test_Image_8_rows_11_cols) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  constexpr int rows{ 8 };
  constexpr int cols{ 11 };

  std::vector<unsigned char> src = getRandomImage(cols, rows);;
  std::vector<unsigned char> resPar;
  std::vector<unsigned char> resSeq;

  resPar = filterImageParallel(src, rows, cols);

  if (rank == 0) {
    resSeq = filterImageSequential(src, rows, cols);
#ifdef DEBUG
    std::cout << "Parallel result:" << std::endl;
    for (int i = 0; i < cols * rows; ++i) {
      std::cout << (unsigned int)resPar[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
    std::cout << "Sequential result:" << std::endl;
    for (int i = 0; i < cols* rows; ++i) {
      std::cout << (unsigned int)resSeq[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
#endif  // DEBUG

    ASSERT_EQ(resPar, resSeq);
  }
}


TEST(Horizontal_Gauss_MPI, Test_Image_16_rows_1_cols) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  constexpr int rows{ 16 };
  constexpr int cols{ 1 };

  std::vector<unsigned char> src = getRandomImage(cols, rows);;
  std::vector<unsigned char> resPar;
  std::vector<unsigned char> resSeq;

  resPar = filterImageParallel(src, rows, cols);

  if (rank == 0) {
    resSeq = filterImageSequential(src, rows, cols);
#ifdef DEBUG
    std::cout << "Parallel result:" << std::endl;
    for (int i = 0; i < cols * rows; ++i) {
      std::cout << (unsigned int)resPar[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
    std::cout << "Sequential result:" << std::endl;
    for (int i = 0; i < cols* rows; ++i) {
```

```
      std::cout << (unsigned int)resSeq[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
#endif  // DEBUG

    ASSERT_EQ(resPar, resSeq);
  }
}

TEST(Horizontal_Gauss_MPI, Test_Image_1_rows_16_cols) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  constexpr int rows{ 16 };
  constexpr int cols{ 1 };

  std::vector<unsigned char> src = getRandomImage(cols, rows);;
  std::vector<unsigned char> resPar;
  std::vector<unsigned char> resSeq;

  resPar = filterImageParallel(src, rows, cols);

  if (rank == 0) {
    resSeq = filterImageSequential(src, rows, cols);
#ifdef DEBUG
    std::cout << "Parallel result:" << std::endl;
    for (int i = 0; i < cols * rows; ++i) {
      std::cout << (unsigned int)resPar[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
    std::cout << "Sequential result:" << std::endl;
    for (int i = 0; i < cols* rows; ++i) {
      std::cout << (unsigned int)resSeq[i] << " ";
      if ((i + 1) % cols == 0) std::cout << std::endl;
    }
#endif  // DEBUG

    ASSERT_EQ(resPar, resSeq);
  }
}

// TEST(Horizontal_Gauss_MPI, Test_Image_500_rows_500_cols) {
// int rank;
// MPI_Comm_rank(MPI_COMM_WORLD, &rank);
//
// constexpr int rows{ 500 };
// constexpr int cols{ 500 };
//
// std::vector<unsigned char> src  = getRandomImage(cols, rows);;
// std::vector<unsigned char> resPar;
// std::vector<unsigned char> resSeq;
//
// double parTime1 = MPI_Wtime();
// resPar = filterImageParallel(src, cols, rows);
// double parTime2 = MPI_Wtime();
//
// if (rank == 0) {
//   double seqTime1 = MPI_Wtime();
//   resSeq = filterImageSequential(src, cols, rows);
//   double seqTime2 = MPI_Wtime();
// #ifdef DEBUG
//   for (int i = 0; i < cols * rows; ++i) {
//     std::cout << (unsigned int)resPar[i] << " ";
//   }
```

```
//   std::cout << std::endl;
//   for (int i = 0; i < cols* rows; ++i) {
//     std::cout << (unsigned int)resSeq[i] << " ";
//   }
// #endif  // DEBUG
//   std::cout << "ParTime " << parTime2 - parTime1 << std::endl;
//   std::cout << "SeqTime " << seqTime2 - seqTime1 << std::endl;
//   ASSERT_EQ(resPar, resSeq);
// }
// }

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();
}
```