

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

Под задачей фильтрации изображений в широком смысле понимают любые процедуры обработки изображений, при которых на вход процедуры подается растровое изображение и на выходе формируется растровое изображение. Однако чаще под «фильтрацией» понимают так называемую помеховую фильтрацию. Главная цель помеховой фильтрации заключается в такой обработке изображений, при которой результат оказывается более подходящим с точки зрения конкретного применения.

Цифровой шум изображения – дефект изображения, вносимый фотосенсорами и электроникой устройств, которые их используют. Для его подавления используют методы линейной фильтрации изображений.

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



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 3×3 . 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

1	2	1
2	4	2
1	2	1

$\ast 1/16$

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.

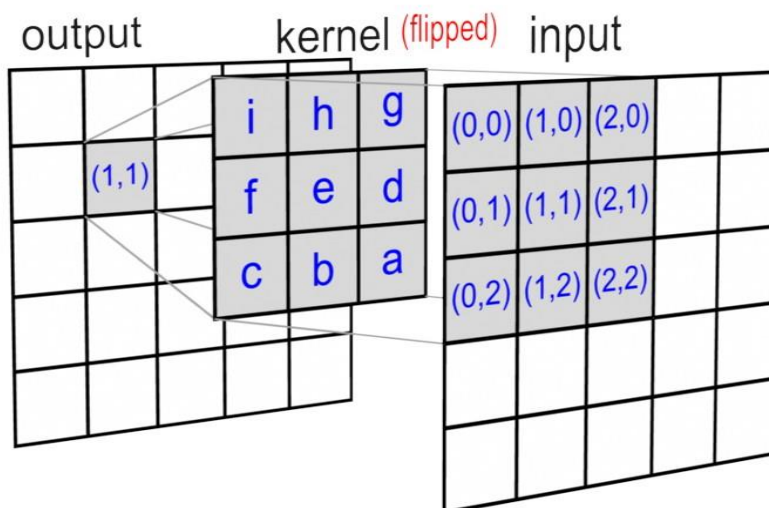


Рисунок 3. Пример «накладывания» матрицы свёртки на изображение

Входное изображение Матрица

12	14	41
43	84	24
2	1	43

×

0,5	0,75	0,5
0,75	1,0	0,75
0,5	0,75	0,5

=

Результат

$$\begin{array}{c} \text{---} \\ \text{---} \end{array}
 \left(\begin{array}{l} 12 * 0,5 + 14 * 0,75 + 41 * 0,5 + \\ 43 * 0,75 + 84 * 1,0 + 24 * 0,75 + \\ 2 * 0,5 + 1 * 0,75 + 43 * 0,5 \end{array} \right)
 \times \frac{1}{\text{div}}
 \begin{array}{c} \text{---} \\ \text{---} \end{array}
 32,41667$$

div = 6

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:

$$\begin{aligned} n1 &= \text{rows} / \text{commSize} \\ n2 &= \text{rows} \% \text{commSize} \end{aligned} \tag{1}$$

где commSize – количество процессов, “/” – операция целочисленного деления, “%” – операция деления с остатком.

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 нужно часть изображения с дополнительными, прилегающие к нему, двумя соседними строками. В случае, если оно последнее, т.е. конец изображения, выслать нужно только верхнюю прилегающую строчку.

Данное дополнение необходимо в следствии вычисления пикселя с помощью матрицы свертки 3×3 , которая задействует 8 соседних пикселей (3 на верхней строке, 3 на нижней строке и 2 на строке обрабатываемого пикселя)

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 10000 x 10000 пикселей.

Количество процессов	Время работы последовательного алгоритма	Время работы параллельного алгоритма
1	3.332сек	3.405 сек
2	3.238 сек	2.076 сек
4	3.355 сек	1.901 сек
8	3.268 сек	1.919 сек

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.

Размер изображения	Время работы последовательного алгоритма	Время работы параллельного алгоритма
500x500	0.0146 сек	0.0046 сек
1000x1000	0.054 сек	0.0319 сек
5000x5000	1.385 сек	0.463 сек
10000x10000	3.269 сек	1.935 сек

Таблица 2. Время работы алгоритма с фиксированным количеством процессов.

Смотря на первый эксперимент можно выделить, что максимальная эффективность параллельного метода достигается на 4 процессах. На 8 разница во времени незначительна относительно выделенных ресурсов.

Исходя из второго эксперимента можно увидеть, что параллельный алгоритм работает в разы эффективней последовательного. На малых размерах изображения видно, что параллельный алгоритм почти что не отличается по времени от последовательного из-за затрат на пересылку данных между процессами. Например, с изображением 500 на 500 или 1000 на 1000 пикселей параллельный алгоритм незначительно эффективней. Но на больших размерах его эффективность становится очевидной. Параллельный алгоритм работает 2-3 раза быстрее последовательного.

7. Заключение

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

Итоговая программа была протестирована, а результаты экспериментов были проанализированы относительно входных данных и количества процессов, на которых выполняется программа.

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
    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 startAddress = (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] + startAddress, 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
}

```

```

    }
#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();
}

```