Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

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

**Выполнил:**

студент группы 1608

Агликов И.А.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc533512866)

[Метод решения 5](#_Toc533512867)

[Схема распараллеливания 6](#_Toc533512870)

[Описание программной реализации 8](#_Toc533512871)

[Подтверждение корректности](#_Toc533512872) 9

[Результаты экспериментов 1](#_Toc533512873)0

[Заключение 1](#_Toc533512874)1

[Приложение 1](#_Toc533512876)2

# Постановка задачи.

В рамках данной лабораторной работы ставится задача реализации линейного фильтра с ядром Гаусса размерностью 3х3. Требуется реализовать как параллельную, так и линейную версию программы. Массив генерируется в начале работы программы на нулевом процессе. По завершении выполнения программа выводит на консоль время работы линейной и параллельной версий программы в секундах.

Где – исходное изображение, – выходное изображение, – ядро фильтра. Таким образом, значение каждого пикселя получаемого изображения определяется окрестными пикселями исходного изображения .

Ядро фильтра, заданное на прямоугольной области, может рассматриваться как матрица некоторой размерности. Эта матрица называется матрицей свёртки. Она содержит коэффициенты, которые покомпонентно умножаются на значения пикселей исходного изображения для получения требуемого результата.

В данной лабораторной работе по умолчанию используется матрица свёртки размерностью 3х3, значения коэффициентов которой заполняются по закону нормального распределения (закону Гаусса):

При нормальном распределении значения быстро убывают по мере отдаления от «центра». Таким образом, влияние цветов пикселей окрестности тем больше, чем они ближе расположены к обрабатываемому пикселю, при этом исходный цвет пикселя будет иметь наибольший «вес».

Таким образом, новый цвет пикселя рассчитывается как сумма произведений соответствующих коэффициентов матрицы свёртки на значения цветов пикселей окрестности. Эта операция применяется ко всем пикселям исходного изображения путём перемещения матрицы.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Стоит упомянуть о случае, который касается обработки границ изображения. У пикселей верхней строки нет соседей сверху, у пикселей нижней строки нет соседей снизу. Аналогичные ограничения имеются для крайних столбцов и «угловых» пикселей.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Имеется несколько способов решения данной проблемы. В работе используется подход, при котором результат обработки граничных пикселей рассчитывается, как если бы изображение дополнялось нулями.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 |  |  |  |  |  |  | 0 |
| 0 |  |  |  |  |  |  | 0 |
| 0 |  |  |  |  |  |  | 0 |
| 0 |  |  |  |  |  |  | 0 |
| 0 |  |  |  |  |  |  | 0 |
| 0 |  |  |  |  |  |  | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Метод решения.

Изображение задаётся как массив байт. Ядро фильтра представляет собой матрицу заданной размерности, коэффициенты которой заполняются по нормальному закону (закону Гаусса). Цвет каждого пикселя выходного изображения рассчитывается как линейная комбинация цветов пикселей окрестности размера 3х3 (размер матрицы свёртки) (цвет каждого пикселя умножается на соответствующий коэффициент матрицы свёртки). Обработка границ изображения осуществляется таким образом, как если бы оно было дополнено нулями. Для применения фильтра ко всему изображению необходимо передвигать матрицу по всем пикселям. Выходное изображение так же представляет собой массив байт.

### Последовательный алгоритм

### Последовательный алгоритм состоит из двух простых этапов:

1. Генерация массива байт заданного пользователем размера (задание изображения);
2. Последовательное применение фильтра к каждому пикселю;

Последовательное применение фильтра к каждому пикселю состоит в следующем:

* Осуществить проход по всему изображению с помощью цикла;
* Для каждого пикселя выполнить:
  + Получить значение цвета пикселя из окрестности;
  + Умножить полученное значение на соответствующий коэффициент матрицы свёртки;
  + Прибавить полученную величину к получаемому значению нового цвета.

Вышеперечисленные шаги для каждого пикселя осуществляются до тех пор, пока не будут пройдены все пиксели окрестности.

# Схема распараллеливания.

Введём следующие обозначения:

* – высота изображения;
* – ширина изображения;
* – число процессов;
* – ранг процесса;
* Результирующий блок и дополнительный столбец – используются для решения проблемы границ передаваемых частей изображения. Дополнительный столбец необходим результирующему блоку для вычисления границ.

Для параллельной работы используются 3 массива:

1. – Количество столбцов, отправляемых каждому процессу, результат которых важен. Отправляется part[pN] + 1 / part[pN] + 2, в зависимости от того, крайний это процесс или нет. Если ширина изображения кратна числу процессов, все элементы part равны Иначе остаток от этого деления расслаивается между процессами с рангами от 1 до .
2. – Смещение результирующих блоков (суммирование элементов part предшествующих по номеру процессов).
3. – Количество отправляемых элементов (кол-во столбцов \* ).

Для решения проблемы обработки границ передаваемых частей изображения было создано 8 пользовательских типов данных: mpi\_LANE, где **∈** [1; 8]. Суть этих типов заключается в том, что вместе с результирующим блоком, который в итоге попадёт в финальное изображение, мы передаём дополнительные столбы. Использование типов зависит от нахождения процесса (первый и последний / в середине), а также от кратности ширины оригинального изображения на количество процессов.

|  |  |  |  |
| --- | --- | --- | --- |
| mpi\_LANE1 | mpi\_LANE2 | mpi\_LANE3 | mpi\_LANE4 |
| |  | | --- | | step[0] | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |     Дополнительный столбец  mpi\_LANE1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |
| На данном типе базируются все остальные. | Имеет 1 дополнительный столбец. Всегда используется первым процессом. Используется последним, если он требует столько же столбцов, сколько и первый. | Имеет дополнительные столбцы по обеим сторонам. | Используется процессами, part которых имеет на 1 столбец больше, чем part[0]. |

Типы mpi\_LANE, где **∈** [5; 8] не были внесены в таблицу, так как отличаются от вышепредставленных не количеством столбцов, а смещением элементов для столбцов в изображении.

Вычисление новых цветов на каждом процессе происходит по описанному в предыдущем разделе алгоритму. После того, как каждый процесс обработает свою часть изображения, осуществляется отправка на процесс результирующих изображений и их слияние.

# Описание программной реализации.

**Руководство пользователя**

Для запуска используется обычный способ, характерный для всех программ, написанных с использованием технологии MPI. Команда для запуска через командную строку имеет вид:

path/to/mpiexec.exe –n <число\_процессов> <название\_программы> par1 par2 par3 par4

par1 – ширина, по умолчанию равна 10;

par2 – высота, по умолчанию равна 10;

par3 – сигма, по умолчанию равна 1.0;

par4 – радиус ядра, по умолчанию равен 1.

Программа выводит время работы последовательного и параллельного алгоритмов, а также исходное и получившееся изображение, если оно имеет площадь меньше либо равно 100.

**Руководство программиста**

Программа состоит из следующих функций:

* printImage() – Вывод принимаемого массива в виде матрицы по принимаемым ширине и высоте;
* createKernel() – Создание матрицы свёртки заданной размерности, коэффициенты уже являются нормированными.
* processImage() – Применение фильтра к изображению. Результат работы – новое изображение;
* Clamp() – Принимает значение пикселя и не даёт ему выйти за пределы [0; 255];
* equality\_check() – Сравнение результатов выполнения последовательного и параллельного алгоритмов.
* main() – Основная функция программы;

Устройство функции main() можно разделить на несколько частей:

1. Объявление переменных, используемых в программе;
2. Считывание данных корневым процессом и задание матрицы изображения;
3. Создание матрицы свёртки и её заполнение;
4. Выполнение корневым процессом последовательного алгоритма;
5. Организация параллельных вычислений:
   1. Корневой процесс передаёт другим процессам информацию о размерах изображения и ядро;
   2. Вычисление процессами объёмов работ и смещений для передач;
   3. Создание буферов на каждом процессе для хранения части изображения;
   4. Рассылка частей изображения каждому процессу;
   5. Обработка каждым процессом своей части изображения;
   6. Сбор обработанных частей на корневом процессе, формирование выходного изображения;
   7. Сравнение последовательного и параллельного результатов;

Исходный код программы можно просмотреть в разделе «[Приложение](#_Приложение)».

# Подтверждение корректности.

Для подтверждения корректности работы параллельного алгоритма в программе предусмотрена функция equality\_check(), принимающая на вход результат работы последовательного и параллельного алгоритмов. Т.к. изображение представляется в виде массива, для проверки корректности осуществляется поэлементное сравнение двух массивов. В случае не нахождения отличий (т.е. результаты совпали), функция печатает, что программа работает корректно. Иначе не печатает ничего.

# Результаты экспериментов.

Эксперименты проводились на машине со следующими характеристиками:

* Процессор: Intel Core i3-6100 3.70GHz;
* Оперативная память: 16,00GB Dual-Channel DDR4 2133MHz;
* ОС: Windows 10.

Полученные значения являются средними по результатам 10 экспериментов. Время указано в секундах.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер  изображения | Последовательный алгоритм | Параллельный алгоритм | | | |
| 2 процесса | | 4 процесса | |
| Время | Ускорение | Время | Ускорение |
| 1920x1080 | 0.0996003 | 0.054356 | 1.8323 | 0.0381826 | 2.6085 |
| 5000х5000 | 1.1687153 | 0.713367 | 1.6383 | 0.434145 | 2.6919 |
| 10000х10000 | 4.6585154 | 2.8262243 | 1.6483 | 1.731354 | 2.6906 |
| 15000х15000 | 10.4963579 | 6.2998922 | 1.6661 | 3.894022 | 2.6955 |
|  |  |  |  |  |  |
| Размер изображения | Последовательный алгоритм | Параллельный алгоритм | | | |
| 6 процессов | | 8 процессов | |
| Время | Ускорение | Время | Ускорение |
| 1920x1080 | 0.0996003 | 0.045151 | 2.2059 | 0.039522 | 2.5201 |
| 5000х5000 | 1.1687153 | 0.452174 | 2.5846 | 0.437461 | 2.6715 |
| 10000х10000 | 4.6585154 | 2.075123 | 2.2449 | 1.740542 | 2.6764 |
| 15000х15000 | 10.4963579 | 4.691849 | 2.2371 | 3.938060 | 2.6653 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Размер изображения | Последовательный алгоритм | Параллельный алгоритм | |  |  |
| 10 процессов | |  |  |
| Время | Ускорение |  |  |
| 1920x1080 | 0.0996003 | 0.044761 | 2.2251 |  |  |
| 5000х5000 | 1.1687153 | 0.488837 | 2.3908 |  |  |
| 10000х10000 | 4.6585154 | 1.972019 | 2.3623 |  |  |
| 15000х15000 | 10.4963579 | 4.437048 | 2.3656 |  |  |

# Заключение.

По представленным выше результатам экспериментов видно, что параллельная программа работает быстрее последовательной. Рост ускорения быстро достигает насыщения, что объясняется накладными расходами на обмен данными между процессами. Кроме того, сама фильтрация изображения осуществляется каждым процессом последовательно и занимает большую часть общего времени работы.

# Приложение.

|  |
| --- |
| // 31. Linear image filtering (vertical split). The Gaussian kernel 3x3.  #include <iostream>  #include <cmath>  #include <ctime>  #include <cstdlib>  #include <mpi.h>  void printImage(unsigned char\* image, int width, int height);  void createKernel(float\* kernel, int radius, int size, float sigma);  void processImage(unsigned char\* originIm, unsigned char\* checkImage,  float\* kernel, int height, int width, int size, int radius);  unsigned char Clamp(float value);  void equality\_check(unsigned char\* res, unsigned char\* res2, int height, int width);  int main(int argc, char\*\* argv) {  srand(time(NULL));  int ProcNum, ProcRank;  int width, height, kernelRadius, size;  float\* kernel = nullptr;  unsigned char\* image = nullptr;  unsigned char\* checkImage = nullptr; // For checking  unsigned char\* partImage = nullptr;  unsigned char\* res = nullptr;  unsigned char\* result = nullptr;  double sequentialTime, parallelTime;    MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcNum);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);  ///////////////////////////////////////////////////////////////////////////  // Non-parallel algorithm  ///////////////////////////////////////////////////////////////////////////  if (ProcRank == 0) {  // Input parametrs  width = ((argc >= 2) && (atoi(argv[1]) > 0)) ? atoi(argv[1]) : 10;  height = ((argc >= 3) && (atoi(argv[2]) > 0)) ? atoi(argv[2]) : 10;  float sigma = (argc >= 4) ? atof(argv[3]) : 1.0;  kernelRadius = ((argc >= 5) && (argv[4] > 0)) ? atoi(argv[4]) : 1;  size = 2 \* kernelRadius + 1;  image = new unsigned char[width \* height];  kernel = new float[size \* size];  checkImage = new unsigned char[width \* height];  // Image initialization  for (int i = 0; i < height; i++)  for (int j = 0; j < width; j++)  image[i \* width + j] = rand() % 255;  if (height \* width <= 100) {  std::cout << "Original image:\n";  printImage(image, width, height); // Print image  }  createKernel(kernel, kernelRadius, size, sigma); // Kernel initialization  // Sequential process  sequentialTime = MPI\_Wtime();  processImage(image, checkImage, kernel, height, width, size, kernelRadius);  sequentialTime = MPI\_Wtime() - sequentialTime;  // Image for checking  if (height \* width <= 100) {  std::cout << "Image for checking:\n";  printImage(checkImage, width, height); // Print image for checking  }  }  ///////////////////////////////////////////////////////////////////////////  // Parallel algorithm  ///////////////////////////////////////////////////////////////////////////  ///////////////////////////////////////////////////////////////////////////  // Preparing  MPI\_Bcast(&width, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Bcast(&height, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Bcast(&kernelRadius, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Bcast(&size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  if (ProcRank != 0)  kernel = new float[size \* size];  MPI\_Bcast(kernel, size \* size, MPI\_FLOAT, 0, MPI\_COMM\_WORLD);  int\* part = new int[ProcNum];  int\* partStep = new int[ProcNum];  int\* step = new int[ProcNum];  if (ProcRank == 0) {  for (int i = 0; i < ProcNum; i++) // step  step[i] = width / ProcNum;  int x = width % ProcNum;  while (x > 0)  step[x--]++;  partStep[0] = 0; // partStep  for (int i = 1; i < ProcNum; i++)  partStep[i] = partStep[i - 1] + step[i - 1];  for (int i = 0; i < ProcNum; i++) // part  part[i] = step[i] \* height;  }  MPI\_Bcast(step, ProcNum, MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Bcast(part, ProcNum, MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Bcast(partStep, ProcNum, MPI\_INT, 0, MPI\_COMM\_WORLD);    MPI\_Datatype mpi\_LANE1;  MPI\_Type\_vector(height, step[0], width, MPI\_UNSIGNED\_CHAR, &mpi\_LANE1);  MPI\_Type\_commit(&mpi\_LANE1);  MPI\_Datatype mpi\_LANE2; // LANE1 + 1  MPI\_Type\_vector(height, step[0] + 1, width, MPI\_UNSIGNED\_CHAR, &mpi\_LANE2);  MPI\_Type\_commit(&mpi\_LANE2);  MPI\_Datatype mpi\_LANE3;  MPI\_Type\_vector(height, step[0] + 2, width, MPI\_UNSIGNED\_CHAR, &mpi\_LANE3);  MPI\_Type\_commit(&mpi\_LANE3);  MPI\_Datatype mpi\_LANE4;  MPI\_Type\_vector(height, step[0] + 3, width, MPI\_UNSIGNED\_CHAR, &mpi\_LANE4);  MPI\_Type\_commit(&mpi\_LANE4);  MPI\_Datatype mpi\_LANE5;  MPI\_Type\_vector(height, step[0], step[0] + 1, MPI\_UNSIGNED\_CHAR, &mpi\_LANE5);  MPI\_Type\_commit(&mpi\_LANE5);  MPI\_Datatype mpi\_LANE6;  MPI\_Type\_vector(height, step[0], step[0] + 2, MPI\_UNSIGNED\_CHAR, &mpi\_LANE6);  MPI\_Type\_commit(&mpi\_LANE6);  MPI\_Datatype mpi\_LANE7;  MPI\_Type\_vector(height, step[0] + 1, step[0] + 2, MPI\_UNSIGNED\_CHAR, &mpi\_LANE7);  MPI\_Type\_commit(&mpi\_LANE7);  MPI\_Datatype mpi\_LANE8;  MPI\_Type\_vector(height, step[0] + 1, step[0] + 3, MPI\_UNSIGNED\_CHAR, &mpi\_LANE8);  MPI\_Type\_commit(&mpi\_LANE8);  result = new unsigned char[width \* height];  if (ProcRank == 0 || ProcRank == (ProcNum - 1))  partImage = new unsigned char[part[ProcRank] + height];  else  partImage = new unsigned char[part[ProcRank] + 2 \* height];  /////////////////////////////////////////////////////////////////////////////  if (ProcRank == 0)  parallelTime = MPI\_Wtime();  if (ProcRank == 0) {  MPI\_Sendrecv(image, 1, mpi\_LANE2, 0, 0,  partImage, part[0] + height, MPI\_INT, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  for (int i = 1; i < ProcNum - 1; i++)  if (step[i] == step[0])  MPI\_Send(image + (partStep[i] - 1), 1, mpi\_LANE3, i, 0, MPI\_COMM\_WORLD);  else  MPI\_Send(image + (partStep[i] - 1), 1, mpi\_LANE4, i, 0, MPI\_COMM\_WORLD);  if (step[ProcNum - 1] == step[0])  MPI\_Send(image + (partStep[ProcNum - 1] - 1), 1, mpi\_LANE2, ProcNum - 1, 0, MPI\_COMM\_WORLD);  else  MPI\_Send(image + (partStep[ProcNum - 1] - 1), 1, mpi\_LANE3, ProcNum - 1, 0, MPI\_COMM\_WORLD);  }  else {  if (ProcRank == ProcNum - 1)  MPI\_Recv(partImage, part[ProcRank] + height, MPI\_UNSIGNED\_CHAR, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  else  MPI\_Recv(partImage, part[ProcRank] + 2 \* height , MPI\_UNSIGNED\_CHAR, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  }  /////////////////////////////////////////////////////////////////////////////  if (ProcRank == 0 || ProcRank == (ProcNum - 1))  res = new unsigned char[part[ProcRank] + height];  else  res = new unsigned char[part[ProcRank] + 2 \* height];  if (ProcRank == 0 || ProcRank == ProcNum - 1)  processImage(partImage, res, kernel, height, step[ProcRank] + 1, size, kernelRadius);  else  processImage(partImage, res, kernel, height, step[ProcRank] + 2, size, kernelRadius);  /////////////////////////////////////////////////////////////////////////////  if (ProcRank == 0) {  MPI\_Sendrecv(res, 1, mpi\_LANE5, 0, 0,  result, 1, mpi\_LANE1, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  for (int i = 1; i < ProcNum - 1; i++)  if (step[i] == step[0])  MPI\_Recv(result + partStep[i], 1, mpi\_LANE1, i, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  else  MPI\_Recv(result + partStep[i], 1, mpi\_LANE2, i, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  if (step[ProcNum - 1] == step[0])  MPI\_Recv(result + partStep[ProcNum - 1], 1, mpi\_LANE1, ProcNum - 1, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  else  MPI\_Recv(result + partStep[ProcNum - 1], 1, mpi\_LANE2, ProcNum - 1, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);  }  else {  if (ProcRank == ProcNum - 1)  if (step[ProcNum - 1] == step[0])  MPI\_Send(res + 1, 1, mpi\_LANE5, 0, 0, MPI\_COMM\_WORLD);  else  MPI\_Send(res + 1, 1, mpi\_LANE7, 0, 0, MPI\_COMM\_WORLD);  else  if (step[ProcRank] == step[0])  MPI\_Send(res + 1, 1, mpi\_LANE6, 0, 0, MPI\_COMM\_WORLD);  else  MPI\_Send(res + 1, 1, mpi\_LANE8, 0, 0, MPI\_COMM\_WORLD);  }  /////////////////////////////////////////////////////////////////////////////  if (ProcRank == 0) { // Print conclusion  parallelTime = MPI\_Wtime() - parallelTime;  if (height \* width <= 100) {  std::cout << "Image made with parallel algorithm:\n";  printImage(result, width, height);  }  equality\_check(result, checkImage, height, width); // Test for equality  std::cout << std::fixed << "Non-parallel algorithm time: "  << sequentialTime << "\nParallel algorithm time: "  << parallelTime << std::endl;  }  MPI\_Finalize();  return 0;  }  void printImage(unsigned char\* image, int width, int height) {  for (int i = 0; i < height; i++) {  for (int j = 0; j < width; j++)  std::cout << image[i \* width + j] << ' ';  std::cout << std::endl;  }  std::cout << std::endl;  }  void createKernel(float\* kernel, int radius, int size, float sigma) {  float norm = 0;  for (int i = -radius; i <= radius; i++)  for (int j = -radius; j <= radius; j++) {  kernel[(i + radius) \* size + (j + radius)] = exp(-(i \* i + j \* j) / (sigma \* sigma));  norm += kernel[(i + radius) \* size + (j + radius)];  }  for (int i = 0; i < size; i++) // Normalization  for (int j = 0; j < size; j++)  kernel[i \* size + j] /= norm;  // Print kernel  std::cout << "Kernel:\n";  for (int i = 0; i < size; i++) {  for (int j = 0; j < size; j++)  std::cout << std::fixed << kernel[i \* size + j] << ' ';  std::cout << std::endl;  }  std::cout << std::endl;  }  void processImage(unsigned char\* originIm, unsigned char\* checkImage, float\* kernel, int height, int width, int size, int radius) {  float tmp;  for (int i = 0; i < height; i++)  for (int j = 0; j < width; j++) {  tmp = 0;  for (int y = -radius; y <= radius; y++)  for (int x = -radius; x <= radius; x++)  if ((i + y) >= 0 && (i + y) < height && (j + x) >= 0 && (j + x) < width)  tmp += originIm[(i + y) \* width + j + x] \* kernel[(y + radius) \* size + x + radius];  checkImage[i \* width + j] = Clamp(roundf(tmp));  }  }  unsigned char Clamp(float value) {  if (value < 0) return 0;  if (value > 255) return 255;  return (unsigned char)value;  }  void equality\_check(unsigned char \* res, unsigned char \* res2, int height, int width) {  bool flag = false;  for (int i = 0; i < height \* width; i++)  if (res2[i] != res[i]) {  flag = !flag;  break;  }  if (!flag)  std::cout << "The program works correctly.\n";  } |