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 (горизонтальное разбиение). Ядро Гаусса 3x3»**

**Выполнил:**

студент группы 1608

Кудалин Р.М.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc532425856)

[Метод решения 5](#_Toc532425857)

[Схема распараллеливания 6](#_Toc532425860)

[Описание программной реализации 8](#_Toc532425861)

[Подтверждение корректности 10](#_Toc532425862)

[Результаты экспериментов 11](#_Toc532425863)

[Заключение 13](#_Toc532425864)

[Список литературы 14](#_Toc532425865)

[Приложение 15](#_Toc532425866)

# Постановка задачи

При работе с изображениями часто возникает необходимость их предварительной обработки, например, выделение и фильтрация шума. При этом в процессе фильтрации необходимо обеспечить максимальное сохранение деталей исходного изображения.

Под фильтрацией изображения понимают операцию, имеющую своим результатом изображение того же размера, полученное из исходного по некоторым правилам. Правила преобразования могут быть разными. В данной лабораторной работе ставится задача реализации линейного фильтра с ядром Гаусса размерностью 3х3.

При линейной фильтрации цвет каждого пикселя выходного изображения рассчитывается как линейная комбинация цветов пикселей некоторой окрестности. Свертка изображения по какой-либо функции называется применением фильтра к изображению:

Где – исходное изображение, – выходное изображение, – ядро фильтра. Таким образом, значение каждого пикселя получаемого изображения определяется пикселями исходного изображения , которые «лежат» в окне .

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

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

При нормальном распределении значения быстро убывают по мере отдаления от «центра» (известно правило 3). Таким образом, влияние цветов пикселей окрестности тем больше, чем они ближе расположены к обрабатываемому пикселю, при этом исходный цвет пикселя будет иметь наибольший «вес». Поэтому использование ядра Гаусса позволяет в большей мере сохранять границы и края объектов исходного изображения.

Таким образом, новый цвет пикселя рассчитывается как сумма произведений соответствующих коэффициентов матрицы свертки на значения цветов пикселей окрестности. Эта операция применяется ко всем пикселям исходного изображения путем перемещения матрицы (Рис. 1).

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

Рис. 1. Применение фильтра к изображению.

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 (Рис. 2).

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

Рис. 2. Проблемы, возникающие при обработке границ изображения.

Имеется несколько способов решения данной проблемы. В работе используется подход, при котором результат обработки граничных пикселей рассчитывается, как если бы изображение дополнялось нулями (Рис. 3).

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

Рис. 3. Изображение, дополненное нулями.

Таким образом, в лабораторной работе ставится задача реализации линейного фильтра с ядром Гаусса размерностью 3х3. Необходимо:

* Реализовать последовательный алгоритм фильтрации;
* Реализовать параллельный алгоритм фильтрации с использованием технологии MPI;
* Оценить сложность и сравнить время выполнения;
* Обеспечить подтверждение корректности работы.

# Метод решения

Предполагается, что изображение задается в оттенках серого, т.е. может быть представлено в программе как массив байт. Ядро фильтра представляет собой матрицу размерности 3х3, коэффициенты которой заполняются по нормальному закону (закону Гаусса). Цвет каждого пикселя выходного изображения рассчитывается как линейная комбинация цветов пикселей окрестности размера 3х3 (цвет каждого пикселя умножается на соответствующий коэффициент матрицы свертки). 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. Загрузка изображения (генерация массива байт заданного пользователем размера);
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 свертки;
  + Прибавить полученную величину к получаемому значению нового цвета.

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

Для дальнейших рассуждений примем следующие обозначения:

– высота изображения;

– ширина изображения;

– размерность ядра;

– число процессов;

– ранг процесса;

Нетрудно убедиться в том, что для каждого пикселя исходного изображения требуется операций умножения и сложения. Таким образом, время, требуемое для фильтрации, составляет

# Схема распараллеливания

В данной лабораторной работе предполагается горизонтальное разделение исходного изображения. Таким образом, каждый процесс будет обрабатывать свою часть исходного изображения. Беря во внимание введенные выше обозначения, получим, что каждый процесс будет обрабатывать строк (Рис. 4). В случае если число строк не кратно числу процессов, тогда первые процессов (т.е. процессы с рангами от до ) получают на одну строку больше.

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

..…

Рис. 4. Распределение строк между процессами.

Каждый процесс применяет фильтр к своей части изображения. При этом возникает проблема обработки граничных строк, т.к. у процесса имеется только часть изображения. Для решения этой проблемы процессы должны обмениваться границами своих частей исходного изображения (Рис. 5).

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

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

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

Рис. 5. Обмен граничными строками между процессами.

Таким образом, процесс с рангом получает верхнюю строку от процесса с рангом и нижнюю строку от процесса с рангом , где

). Процессы с рангами 0 и получают по одной строке от процессов с рангами 1 и ) соответственно.

Стоит отметить, что эти дополнительные строки каждый процесс использует только для вычисления новых цветов пикселей граничных строк своей части изображения. Вычисление цветов пикселей этих дополнительных строк осуществляется теми процессами, частями изображения которых они являются.

Вычисление новых цветов на каждом процессе происходит по описанному в предыдущем разделе алгоритму. После того как каждый процесс обработает свою часть изображения, осуществляется слияние этих частей в единое выходное изображение.

Время выполнения параллельного алгоритма складывается из времени, которое каждый процесс затрачивает непосредственно на обработку своей части, и времени, требуемого для организации параллельных вычислений. На обработку своей части изображения каждый процесс затрачивает ). Время, требуемое на организацию параллельных вычислений (вычисление числа строк для каждого процесса, вычисление смещений, отправка частей, обмен граничными строками, сбор частей) линейно зависит от числа процессов и составляет .

Таким образом, время, требуемое для выполнения параллельного алгоритма, составляет

*.*

# Описание программной реализации

**Руководство пользователя**

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

path/to/mpiexec.exe –n <число\_процессов> <название\_программы> <высота> <ширина>

Программа выводит время работы последовательного и параллельного алгоритмов, результат сравнения их работы, а также исходное и получившееся изображение, если они имеют размер ниже порогового (по умолчанию пороговое значение равно 20 строкам).

При вводе отрицательных значений высоты и ширины для них устанавливаются значения по умолчанию (10000х10000).

Пример запуска программы представлен на Рис. 6.

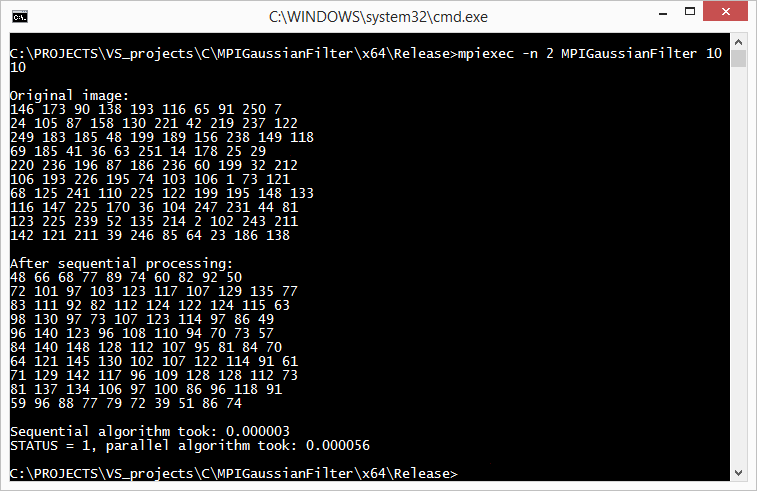


Рис. 6. Пример запуска программы.

**Руководство программиста**

Программа состоит из следующих функций:

* LoadImage() – загрузка изображения (генерация случайного массива байт);
* ShowImage() – вывод изображения;
* CreateGaussianKernel() – создание матрицы свертки и ее заполнение по закону Гаусса;
* ProcessImage() – применение фильтра к изображению. Результат работы – новое изображение;
* CheckEquality() – 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 пользовательских данных корневым процессом и загрузка изображения;
3. Создание матрицы свертки и ее заполнение;
4. Выполнение корневым процессом последовательного алгоритма;
5. Организация параллельных вычислений:
   1. Корневой процесс передает другим процессам информацию о размерах изображения и ядро (MPI\_Bcast());
   2. Вычисление процессами объемов работ и смещений для функций MPI\_Scatterv() и MPI\_Gatherv();
   3. Создание буферов на каждом процессе для хранения части изображения;
   4. Рассылка частей изображения каждому процессу (MPI\_Scatterv());
   5. Обмен краевыми строками между процессами (MPI\_Sendrecv());
   6. Обработка каждым процессом своей части изображения;
   7. Сбор обработанных частей на корневом процессе, формирование выходного изображения (MPI\_Gatherv());
   8. Проверка результатов выполнения последовательного и параллельного алгоритмов;
   9. Очистка памяти, выделенной в ходе выполнения программы.

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

# Подтверждение корректности

Для подтверждения корректности работы параллельного алгоритма в программе предусмотрена функция CheckEquality(), 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 расхождения значений каких-либо элементов функция возвращает 0. Если же массивы равны, функция возвращает 1, что говорит о том, что результаты работы параллельного и последовательного алгоритмов совпадают.

Возвращенное этой функцией значение выводится в параметре STATUS при завершении программы, как видно на Рис. 6.

С исходным кодом функции можно ознакомиться в разделе «[Приложение](#_Приложение)».

# Результаты экспериментов

Эксперименты проводились на машине со следующими характеристиками:

* Процессор: Intel Core i7-3630QM 2.40GHz;
* Оперативная память: 8,00ГБ Dual-Channel DDR3 798MHz;
* ОС: Windows 8.1

Результаты экспериментов приведены в табл. 1 и на рис. 7. 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 |
| 5000х5000 | 0,460018 | 0,2539734 | 1,8112 | 0,1434317 | 3,2072 |
| 10000х10000 | 1,8510809 | 1,0321671 | 1,7934 | 0,5661965 | 3,2693 |
| 20000х20000 | 7,4726326 | 4,0919892 | 1,8261 | 2,3704706 | 3,1523 |
|  |  |  |  |  |  |
| Размер изображения | Последовательный алгоритм | Параллельный алгоритм | | | |
| 6 процессов | | 8 процессов | |
| Время | Ускорение | Время | Ускорение |
| 5000х5000 | 0,460018 | 0,164161 | 2,8022 | 0,1401149 | 3,2831 |
| 10000х10000 | 1,8510809 | 0,597979 | 3,0955 | 0,5548884 | 3,3359 |
| 20000х20000 | 7,4726326 | 2,3419853 | 3,1907 | 2,2375884 | 3,3395 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Размер изображения | Последовательный алгоритм | Параллельный алгоритм | |  |  |
| 10 процессов | |  |  |
| Время | Ускорение |  |  |
| 5000х5000 | 0,460018 | 0,1463697 | 3,1428 |  |  |
| 10000х10000 | 1,8510809 | 0,5564459 | 3,3327 |  |  |
| 20000х20000 | 7,4726326 | 2,2302417 | 3,3505 |  |  |

Таблица 1. Результаты экспериментов.

Рис. 7. Результаты экспериментов.

# Заключение

По представленным выше результатам экспериментов видно, что параллельная программа работает быстрее последовательной. Однако ускорение не является линейным и быстро достигает насыщения. Это можно объяснить высокими накладными расходами на организацию параллельной работы: процессы должны обмениваться данными друг с другом. Кроме того, сама фильтрация изображения осуществляется каждым процессом последовательно и занимает большую часть общего времени работы.

# Список литературы

1. Richard Hartley and Andrew Zisserman Multiple View Geometry in computer vision, Cambridge Press, 2000, ISBN-0-521-62304-9 – Chapter 8, Linear Filters.
2. IFAC Large Scale Systems: Theory and Applications // -2001. –Vol. 34, No.8. –P. 349-354.
3. Gaussian Blur: [https://en.wikipedia.org/wiki/Gaussian\_blur].

# Приложение

|  |
| --- |
| #include <mpi.h> #include <stdlib.h> #include <stdio.h> #include <time.h> #include <math.h>  #define KERNEL\_SIZE 3 #define PI\_NUM 3.14159265358979323846 #define VIEW\_THRESHOLD 20 #define DEFAULT\_IMAGE\_SIZE 10000   unsigned char\* LoadImage(int height, int width) {  unsigned char\* image =  (unsigned char\*)malloc(height \* width \* sizeof(unsigned char));   for (int i = 0; i < height \* width; ++i) {  image[i] = rand() % 256;  }   return image; }  void ShowImage(unsigned char\* image, int height, int width) {  for (int i = 0; i < height; ++i) {  for (int j = 0; j < width; ++j) {  printf("%hhu ", image[j + i \* width]);  }  printf("\n");  } }  void ShowKernel(double\* kernel) {  for (int i = 0; i < KERNEL\_SIZE; ++i) {  for (int j = 0; j < KERNEL\_SIZE; ++j) {  printf("%.5f ", kernel[j + i \* KERNEL\_SIZE]);  }  printf("\n");  } }  double\* CreateGaussianKernel(double sigma) {  double\* kernel =  (double\*)malloc(KERNEL\_SIZE \* KERNEL\_SIZE \* sizeof(double));   double coeff = 1 / (sigma \* sigma \* 2 \* PI\_NUM);  int radius = KERNEL\_SIZE / 2;  for (int i = -radius; i <= radius; ++i) {  for (int j = -radius; j <= radius; ++j) {  kernel[j + radius + (i + radius) \* KERNEL\_SIZE] =  coeff \* exp(-(i \* i + j \* j) / (2 \* sigma \* sigma));  }  }   return kernel; }  unsigned char\* ProcessImage(unsigned char\* image, double\* kernel,  int height, int width, int start, int end) {   unsigned char\* result\_image =  (unsigned char\*)malloc((end - start) \* width \* sizeof(unsigned char));   int radius = KERNEL\_SIZE / 2;  for (int i = start; i < end; ++i) {  for (int j = 0; j < width; ++j) {  double result\_color = .0;  unsigned char adjacent\_color = 0;  for (int k = -radius; k <= radius; ++k) {  for (int q = -radius; q <= radius; ++q) {  if ((j + q) >= 0 && (j + q) < width &&  (i + k) >= 0 && (i + k) < height) {   adjacent\_color = image[j + q + (i + k) \* width];   result\_color += adjacent\_color \* kernel[q + radius  + (k + radius) \* KERNEL\_SIZE];  }  }  }  result\_image[j + (i - start) \* width] =  (unsigned char)((result\_color <= 255) ? result\_color : 255);  }  }   return result\_image; }  int CheckEquality(unsigned char\* seq\_res\_img, unsigned char\* par\_res\_img,  int height, int width) {   for (int i = 0; i < height \* width; ++i) {  if (seq\_res\_img[i] != par\_res\_img[i]) {  return 0;  }  }   return 1; }  int main(int argc, char\*\* argv) {  srand(time(NULL));   int proc\_num, proc\_rank;  const int root = 0;  double par\_begin, par\_end;  int\* counts = NULL;  int\* displs = NULL;   double seq\_begin, seq\_end;   int height, width, original\_height;  int extended\_img\_part\_height;  int remaining\_rows;  int sent\_rows\_num;  int start\_row\_index, end\_row\_index;  int offset;   unsigned char\* image = NULL;  unsigned char\* seq\_res\_img = NULL;  unsigned char\* par\_res\_img = NULL;  unsigned char\* proc\_img\_part = NULL;  unsigned char\* proc\_res\_img = NULL;  double\* kernel = NULL;   MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);   if (root >= proc\_num || proc\_num == 1) {  MPI\_Finalize();  exit(EXIT\_FAILURE);  }   if (proc\_rank == root) {  if (argc != 3) {  height = width = DEFAULT\_IMAGE\_SIZE;  } else {  height = atoi(argv[1]);  width = atoi(argv[2]);  }  if (height <= 0) {  height = DEFAULT\_IMAGE\_SIZE;  }  if (width <= 0) {  width = DEFAULT\_IMAGE\_SIZE;  }   original\_height = height;   image = LoadImage(height, width);   if (height <= VIEW\_THRESHOLD) {  printf("\nOriginal image:\n");  ShowImage(image, height, width);  }   kernel = CreateGaussianKernel(1);   // Execute sequential algorithm  seq\_begin = MPI\_Wtime();  seq\_res\_img = ProcessImage(image, kernel, height, width, 0, height);  seq\_end = MPI\_Wtime();   if (height <= VIEW\_THRESHOLD) {  printf("\nAfter sequential processing:\n");  ShowImage(seq\_res\_img, height, width);  }   printf("\nSequential algorithm took: %f\n", seq\_end - seq\_begin);    par\_res\_img =  (unsigned char\*)malloc(height \* width \* sizeof(unsigned char));   } else {  kernel = (double\*)malloc(KERNEL\_SIZE \* KERNEL\_SIZE \* sizeof(double));  }   // Execute parallel algorithm  MPI\_Barrier(MPI\_COMM\_WORLD);  par\_begin = MPI\_Wtime();   MPI\_Bcast(&height, 1, MPI\_INT, root, MPI\_COMM\_WORLD);  MPI\_Bcast(&width, 1, MPI\_INT, root, MPI\_COMM\_WORLD);  MPI\_Bcast(kernel, KERNEL\_SIZE \* KERNEL\_SIZE, MPI\_DOUBLE,  root, MPI\_COMM\_WORLD);   if (proc\_num > height) {  printf("proc\_num should be <= height");  free(image);  free(seq\_res\_img);  free(par\_res\_img);  free(kernel);  MPI\_Finalize();  exit(EXIT\_FAILURE);  }   remaining\_rows = height % proc\_num;   height /= proc\_num;   counts = (int\*)malloc(proc\_num \* sizeof(int));  displs = (int\*)malloc(proc\_num \* sizeof(int));   sent\_rows\_num = 0;  for (int i = 0; i < proc\_num; ++i) {  counts[i] = height \* width;  if (remaining\_rows > 0) {  counts[i] += width;  remaining\_rows--;  }   displs[i] = sent\_rows\_num;  sent\_rows\_num += counts[i];  }   offset = width;  start\_row\_index = 1;  extended\_img\_part\_height = counts[proc\_rank] / width + 1;   if (proc\_rank == 0) {  offset = 0;  start\_row\_index = 0;  end\_row\_index = extended\_img\_part\_height - 1;  proc\_img\_part =  (unsigned char\*)malloc((counts[proc\_rank] + width) \*  sizeof(unsigned char));   } else if (proc\_rank == proc\_num - 1) {  end\_row\_index = extended\_img\_part\_height;  proc\_img\_part =  (unsigned char\*)malloc((counts[proc\_rank] + width) \*  sizeof(unsigned char));   } else {  extended\_img\_part\_height++;  end\_row\_index = extended\_img\_part\_height - 1;  proc\_img\_part =  (unsigned char\*)malloc((counts[proc\_rank] + 2 \* width)  \* sizeof(unsigned char));  }   MPI\_Scatterv(image, counts, displs, MPI\_UNSIGNED\_CHAR, proc\_img\_part + offset,  counts[proc\_rank], MPI\_UNSIGNED\_CHAR, root, MPI\_COMM\_WORLD);   if (proc\_rank != 0) {  MPI\_Sendrecv(proc\_img\_part + offset, width, MPI\_UNSIGNED\_CHAR,  proc\_rank - 1, 0, proc\_img\_part, width, MPI\_UNSIGNED\_CHAR,  proc\_rank - 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  }  if (proc\_rank != proc\_num - 1) {  MPI\_Sendrecv(proc\_img\_part + offset + counts[proc\_rank] - width, width,  MPI\_UNSIGNED\_CHAR, proc\_rank + 1, 0,  proc\_img\_part + offset + counts[proc\_rank], width, MPI\_UNSIGNED\_CHAR,  proc\_rank + 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  }   proc\_res\_img = ProcessImage(proc\_img\_part, kernel,  extended\_img\_part\_height, width, start\_row\_index, end\_row\_index);   MPI\_Gatherv(proc\_res\_img, counts[proc\_rank], MPI\_UNSIGNED\_CHAR,  par\_res\_img, counts, displs, MPI\_UNSIGNED\_CHAR, root, MPI\_COMM\_WORLD);   MPI\_Barrier(MPI\_COMM\_WORLD);  par\_end = MPI\_Wtime();   if (proc\_rank == root) {  if (original\_height <= VIEW\_THRESHOLD) {  printf("\nAfter parallel processing:\n");  ShowImage(par\_res\_img, original\_height, width);  }  printf("STATUS = %d, parallel algorithm took: %f\n",  CheckEquality(seq\_res\_img, par\_res\_img, original\_height, width),  par\_end - par\_begin);   free(image);  free(seq\_res\_img);  free(par\_res\_img);  }   free(counts);  free(displs);  free(proc\_img\_part);  free(proc\_res\_img);  free(kernel);   MPI\_Finalize();   return 0; } |