

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 3×3 .»

Выполнил:

студент группы 1606-1

Тужилкина А.А.

Проверил:

Кустикова В.Д.

Нижний Новгород

2018

Содержание

Постановка задачи.....	3
Метод решения.....	4
Схема распараллеливания	5
Описание программной реализации.....	6
Подтверждение корректности.....	7
Результаты экспериментов	8
Заключение	10
Приложение	11

Постановка задачи

Одна из самых важных задач при работе с изображениями связана с их предварительной обработкой - выделением и фильтрацией шума. При этой обработке необходимо обеспечить максимальное сохранение деталей изображения. Фильтрация искаженных пикселей относится к группе низкоуровневых операций обработки изображения. При последовательной обработке каждого пикселя время обработки изображений достаточно велико. Это неприемлемо в реальном времени для решения различных прикладных задач. Поэтому эту проблему можно решить, если использовать высокопроизводительные параллельные вычислительные машины.

В данной лабораторной работе ставится задача реализовать программу, использующую средства параллельного программирования MPI, которая фильтрует изображение, используя ядро Гаусса 3×3 .

Метод решения

Главная часть матричного фильтра – ядро – это матрица коэффициентов, которая покомпонентно умножается на значение пикселей изображения для получения требуемого результата (не то же самое, что матричное умножение, коэффициенты матрицы являются весовыми коэффициентами для выбранного подмассива изображения).

Для размытия изображений используют фильтр Гаусса, коэффициенты для которого рассчитываются по формуле Гаусса:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

По условию задачи необходимо выполнить горизонтальное разбиение на блоки. Данные в массиве хранятся по строкам, поэтому это самое естественное разбиение. Серые зоны - информация о границах. Соседние процессы обмениваются этой информацией. Стрелки указывают направление обмена данными.

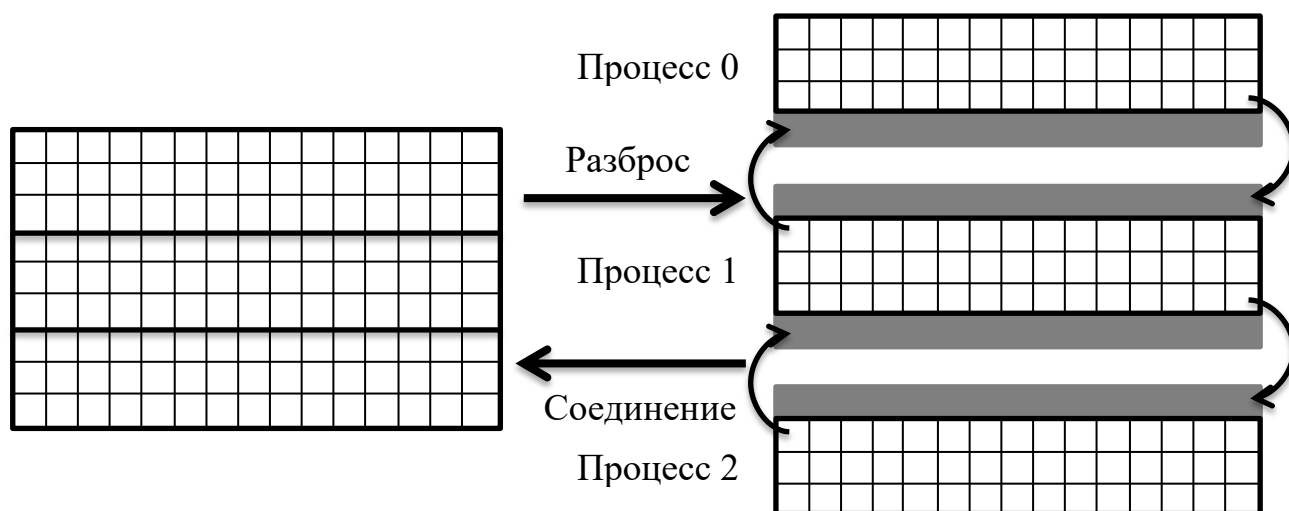


Рисунок 1. Обмен «граничными» строками между процессами.

Схема распараллеливания

Вся поверхность изображения в процессе обработки делится на горизонтальные блоки. Параллельно в рамках отдельного процесса выполняется обработка пикселей каждого блока. После этого необходимо произвести обмен «граничной» информацией между соседними процессами. Это нежелательный побочный эффект распараллеливания, требующий связи между параллельно работающими процессами через связывающий элемент (для отправки и получения информации между процессами, участвующими в параллельной обработке). Рассмотрим алгоритм:

- 1) Деление изображения на равные горизонтальные блоки
- 2) Обмен «граничной» информацией между соседними блоками
- 3) Обработка каждого из блоков
- 4) Объединение обработанных блоков в изображение

Описание программной реализации

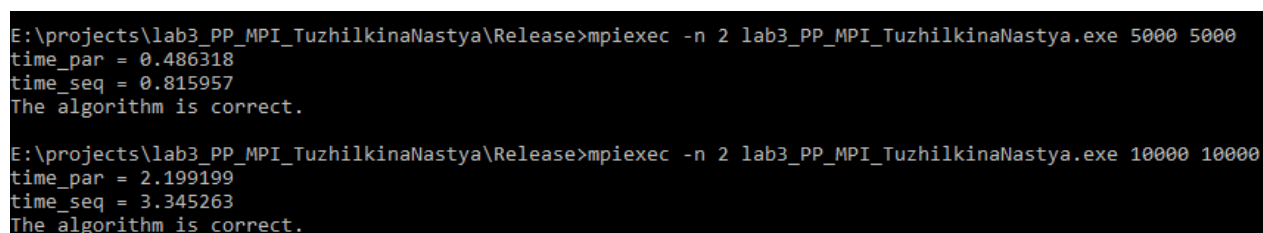
Руководство пользователя

Исполняемый файл с реализацией работы lab3_PP_MPI_TuzhilkinaNastya.exe является MPI приложением, поэтому необходимо осуществлять запуск для правильной работы в параллельном режиме в соответствующей среде выполнения MPI.

Для запуска программы необходимо перейти в директорию с файлом lab3_PP_MPI_TuzhilkinaNastya.exe и ввести следующую команду:

```
mpiexec -n <proc> lab3_PP_MPI_TuzhilkinaNastya.exe <w> <h>
```

- proc – число процессов;
- w – ширина изображения;
- h – высота изображения.



```
E:\projects\lab3_PP_MPI_TuzhilkinaNastya\Release>mpiexec -n 2 lab3_PP_MPI_TuzhilkinaNastya.exe 5000 5000
time_par = 0.486318
time_seq = 0.815957
The algorithm is correct.

E:\projects\lab3_PP_MPI_TuzhilkinaNastya\Release>mpiexec -n 2 lab3_PP_MPI_TuzhilkinaNastya.exe 10000 10000
time_par = 2.199199
time_seq = 3.345263
The algorithm is correct.
```

Рисунок 2. Пример запуска программы.

Выводятся параллельное время и последовательное время. Если подтверждается корректность программы, то выводится сообщение «The algorithm is correct.».

Руководство программиста

Программа реализующая поставленную задачу состоит из следующих функций:

- *generateKernel* – функция, которая создает матрицу свертки для фильтра, основываясь на распределении Гаусса;
- *countElemProc* – функция, готовящая данные для передачи в функцию MPI_Scatterv (число элементов передаваемых каждому из процессов, массив сдвигов);
- *imageProcess* – функция, в которой с помощью созданной матрицы свёртки организована обработка изображения;
- *checkEquality* – функция, которая сравнивает результат работы двух алгоритмов: последовательного и параллельного;
- *main()* – основная функция программы, где происходит создание изображения и его дальнейшая обработка.

Код программы можно просмотреть в разделе «Приложение».

Подтверждение корректности

Для подтверждения корректности в программе реализована функция автоматической проверки *checkEquality()*, которая принимает на вход результат работы двух алгоритмов: последовательного и параллельного. Функция возвращает 0, если расходятся значения каких-то элементов. Функция возвращает 1, если результаты работы двух алгоритмов совпадают. Если подтверждается корректность программы, то выводится сообщение «The algorithm is correct.».

Результаты экспериментов

Эксперименты проводились на ПК со следующими характеристиками:

- Процессор: Pentium(R) Dual-Core CPU T4500 @ 2.30GHz 2.30GHz.
- Оперативная память: 2,00ГБ.
- ОС: Windows 10 Корпоративная 2016 с долгосрочным обслуживанием.

Рассмотрим приведённую ниже таблицу 1 для оценки результатов:

Таблица 1.

	5000×5000	10000×10000	20000×20000
Последовательный алгоритм	0,814445	3,322277	13,641659
Параллельный алгоритм 2 процесса			
Время	0,472106	1,921428	11,032743
Ускорение	1,725132	1,729066	1,236470
Параллельный алгоритм 4 процесса			
Время	0,498356	1,977131	12,333076
Ускорение	1,634263	1,680352	1,106103
Параллельный алгоритм 6 процессов			
Время	0,485667	1,875000	11,791835
Ускорение	1,676962	1,771881	1,156873
Параллельный алгоритм 8 процессов			
Время	0,489760	1,930301	10,154982
Ускорение	1,662947	1,721119	1,343346
Параллельный алгоритм 10 процессов			
Время	0,656430	1,969818	8,212991
Ускорение	1,240718	1,686590	1,660986

Исходя из полученных данных можно оценить среднее ускорение:

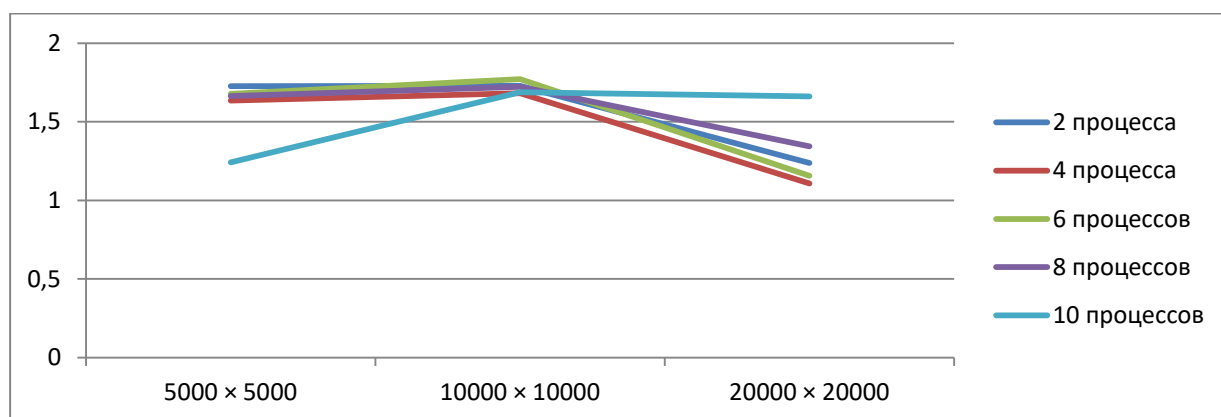


Рисунок 3. Результаты экспериментов.

Заключение

В данной лабораторной работе был разработан параллельный алгоритм фильтрации изображений. По представленным выше результатам экспериментов видно, что параллельная программа работает быстрее последовательной. В случае горизонтального разбиения отправка/получение частей изображения между основным и подчиненными процессами была упрощена функциями `MPI_Scatterv` и `MPI_Gather` из библиотеки функций `MPI`. При обработке изображений большого объема в реальном времени можно эффективно использовать предлагаемый алгоритм.

Приложение

```
#define _USE_MATH_DEFINES
#define STB_IMAGE_IMPLEMENTATION
#define SIGMA 2

#include <stdint.h>
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <stdlib.h>

typedef unsigned char uint8_t;

void generateKernel(double* kernel, int sigma)
{
    for (int i = -1; i <= 1; i++)
    {
        for (int j = -1; j <= 1; j++)
        {
            kernel[i + j + 2 * (i + 2)] = (double)(exp(-(i*i + j*j) /
(sigma*sigma))) / (sigma*sqrt(2 * M_PI));
        }
    }
}

void countElemProc(int* elem_proc, int* displs, int height, int width, int psize)
{
    int port = height / psize;
    port = (port < 1) ? 1 : port;
    int rem = height % psize;
    int elem = port*width;

    int sent_rows_num = 0;
    for (int i = 0; i < psize; ++i)
    {
        elem_proc[i] = elem;
        if (rem > 0)
        {
            elem_proc[i] += width;
            rem--;
        }

        displs[i] = sent_rows_num;
        sent_rows_num += elem_proc[i];
    }
}

uint8_t* imageProcess(uint8_t* recvbuf, double* kernel, int width, int elem_proc)
{
    uint8_t* res = new uint8_t[elem_proc];
    for (int i = width; i < elem_proc - width; i++)
    {
        if (((i % width) != (width - 1)) && ((i % width) != 0))
        {
            int k = 0;
            for (int j = -width; j <= width; j += width)
            {
                res[i] += static_cast<uint8_t>(recvbuf[(i - 1) + j] * kernel[k] +
recvbuf[i + j] * kernel[k + 1] + recvbuf[(i + 1) + j] * kernel[k + 2]);
                k += 3;
            }
        }
    }
}
```

```

        else
        {
            res[i] = recvbuf[i];
        }
    }
    return res;
}

int checkEquality(uint8_t* seq_image, uint8_t* par_image, int height, int width)
{
    for (int i = 0; i < height * width; ++i)
    {
        if (seq_image[i] != par_image[i])
        {
            printf("%d\n", i);
            return 0;
        }
    }
    return 1;
}

void main(int argc, char* argv[])
{
    int ProcRank;
    int psize = 0;
    int width = 0, height = 0;
    int port = 0;
    int offset = 0;
    double rem = 0;
    double* kernel = new double[9];
    double start_par_time = 0, end_par_time = 0;
    double start_seq_time = 0, end_seq_time = 0;
    uint8_t* par_image = nullptr;
    uint8_t* seq_image = nullptr;

    if (argc < 3)
    {
        printf("Image size not set\n");
        return;
    }
    else
    {
        if (atoi(argv[1]) < 100 && atoi(argv[2]) < 100)
        {
            printf("Invalid arguments\n");
            return;
        }
    }

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &psize);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    double* buf = new double[11];
    if (ProcRank == 0)
    {
        width = atoi(argv[1]);
        height = atoi(argv[2]);

        generateKernel(kernel, SIGMA);
        par_image = new uint8_t[width*height];
        for (int i = 0; i < width*height; i++)
        {
            par_image[i] = rand() % 255;
        }
    }

```

```

        buf[0] = width;
        buf[1] = height;
        for (int i = 2; i < 11; i++)
        {
            buf[i] = kernel[i - 2];
        }
    }

    //Sequential algorithm

    start_seq_time = MPI_Wtime();
    if (ProcRank == 0)
        seq_image = imageProcess(par_image, kernel, width, height*width);
    end_seq_time = MPI_Wtime();

    //Prepare additional data

    start_par_time = MPI_Wtime();

    MPI_Bcast(buf, 11, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if (ProcRank != 0)
    {
        width = buf[0];
        height = buf[1];
        for (int i = 2; i < 11; i++)
        {
            kernel[i - 2] = buf[i];
        }
        offset = width;
    }

    port = height / psize;
    port = (port < 1) ? 1 : port;
    rem = height - port * psize;    //remainder

    int* elem_proc = new int[psize];    // count elements for one proc
    int* displs = new int[psize];    // displ for each portion
    int* sizerec = new int[psize];    // size recv buf

    countElemProc(elem_proc, displs, height, width, psize);

    //Image division

    uint8_t* recvbuf = nullptr;
    if ((ProcRank == 0) || (ProcRank == psize - 1))
    {
        sizerec[ProcRank] = elem_proc[ProcRank] + width;
        recvbuf = new uint8_t[sizerec[ProcRank]];
    }
    else
    {
        sizerec[ProcRank] = elem_proc[ProcRank] + 2 * width;
        recvbuf = new uint8_t[sizerec[ProcRank]];
    }
    for (int i = 0; i < elem_proc[ProcRank]; i++)
    {
        recvbuf[i] = 0;
    }

```

```

    MPI_Scatterv(par_image, elem_proc, displs, MPI_UINT8_T, recvbuf + offset,
elem_proc[ProcRank], MPI_UINT8_T, 0, MPI_COMM_WORLD);

    //Block edges transfer

    MPI_Status status;

    if (ProcRank != psize - 1)
    {
        MPI_Recv(recvbuf + elem_proc[ProcRank] + offset, width, MPI_UINT8_T, ProcRank
+ 1, 0, MPI_COMM_WORLD, &status);
    }
    if (ProcRank != 0)
    {
        MPI_Send(recvbuf + offset, width, MPI_UINT8_T, ProcRank - 1, 0,
MPI_COMM_WORLD);
    }

    if (ProcRank != 0)
    {
        MPI_Recv(recvbuf, width, MPI_UINT8_T, ProcRank - 1, 0, MPI_COMM_WORLD,
&status);
    }
    if (ProcRank != psize - 1)
    {
        MPI_Send(recvbuf + elem_proc[ProcRank] - width + offset, width, MPI_UINT8_T,
ProcRank + 1, 0, MPI_COMM_WORLD);
    }

    //Image processing and build new image

    recvbuf = imageProcess(recvbuf, kernel, width, sizerec[ProcRank]);

    MPI_Gatherv((void*)(recvbuf + offset), elem_proc[ProcRank], MPI_UINT8_T, par_image,
elem_proc, displs, MPI_UINT8_T, 0, MPI_COMM_WORLD);

    end_par_time = MPI_Wtime();

    //Correctness

    int res = 0;
    if (ProcRank == 0)
    {
        res = checkEquality(seq_image, par_image, height, width);
    }

    if (ProcRank == 0)
    {
        delete[] par_image;
        delete[] seq_image;
        printf("time_par = %f\n", end_par_time - start_par_time);
        printf("time_seq = %f\n", end_seq_time - start_seq_time);
        if (res)
            printf("The algorithm is correct.\n");
    }
    delete[] kernel;
    delete[] buf;

    MPI_Finalize();
}

```