Parallel Computing - MPI

Last edit: May 9, 2017

Introduction

Documentation on MPI can be found be found **here**[1].

There are three assignments available for this week. For each assignment a sequential program is available and listed in the appendices. These will form the basis for your solutions. Some assignments will impose some constraints you must comply to, these will mostly forbid modifications in (parts of) some files. These constraints are introduced as a form of a guideline on how to parallelise the programs. For every assignment a test is available on Nestor.

Before you can compile your code on the Peregrine cluster, you will first have to load one of the available modules of the cluster in order to use the MPI libraries and compilers. This is done using `module load foss/2016a`.

For these assignments it is recommended that you take a look at the PDF 'MPI-lab-MultidimensionalArrays'.

Contents

[1]`http://www.mpi-forum.org/`

1 | Load Balancing - Primes

In this exercise you are asked to compute the number of prime numbers in a given interval $[a, b]$. Let a and b be natural numbers, such that $1 \leq a \leq b$.

Assignment: A sequential program that performs the task at hand is listed in appendix A.1 on page 5.

- **a**. Parallelise the given program using MPI, by dividing the interval into equal subintervals. Each thread should operate on a single subinterval. Measure the time each thread requires to complete its computations.
- **b**. Devise and apply a different load balancing strategy. Make sure that all threads perform approximately equal amounts of computations.

Constraints: Use the given `isPrime` function to determine whether a number is prime, and do not modify said function.

Requirements: In order to be able to the questions on Nestor, you are to time your parallelised program for 1, 2, 4, 8, 12, 16 and 24 threads. Also time each process individually and determine the total computation time. Additionally, determine the real execution time. Run the program on the interval $[1, 5 \times 10^7]$. Make sure that the measured times are part of your output.

2 | Data Distribution - Contrast

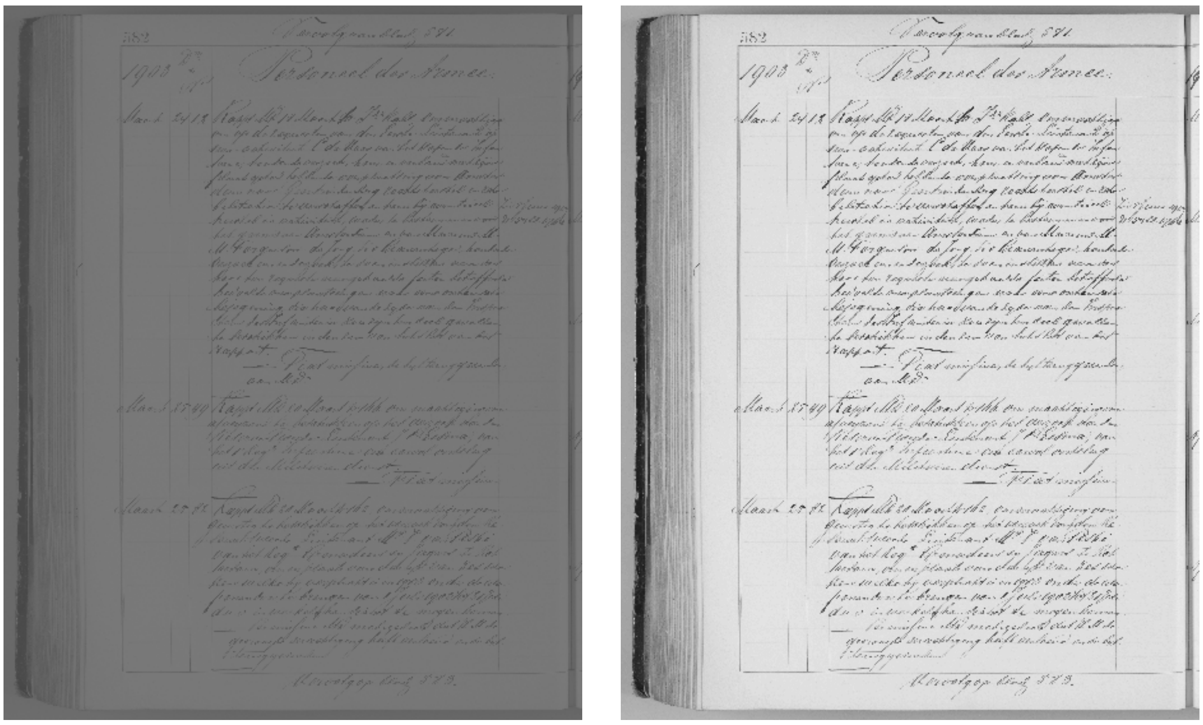

Figure 1: (Left) Image before contrast stretching. (Right) Image after contrast stretching.

A very simple digital greyscale imaging operation is the enhancement of the contrast of an image. In this exercise you will develop a parallel algorithm for this operation. Figure 1 displays an example of contrast stretching. A sequential program that solves the problem at hand is listed in appendix A.2 on page 6.

Assignment: The given program only initialises the MPI environment. Modify the functions `readImage` and `writeImage`, such that the master process ($rank = 0$) reads and writes images from/to storage. Said process should also distribute the data to the other processes using the `Scatterv` and `Gatherv` operations. Also change the `contrastStretch` function, such that contrast stretching is performed in parallel.

Constraints: Modifications of the program should take place in `contrast.c`; the other files are not to be modified.

Requirements: Time your parallelised program for 1, 2, 4, 8, 12, 16 and 24 threads. Also time each process individually and determine the total computation time. Additionally, determine the real execution time. Run the program on the images *kdk*, *netherlands_small*, and *netherlands_big*. Make sure that the measured times are part of your output.

3 | Master/Slave Load Balancing - Mandelbrot

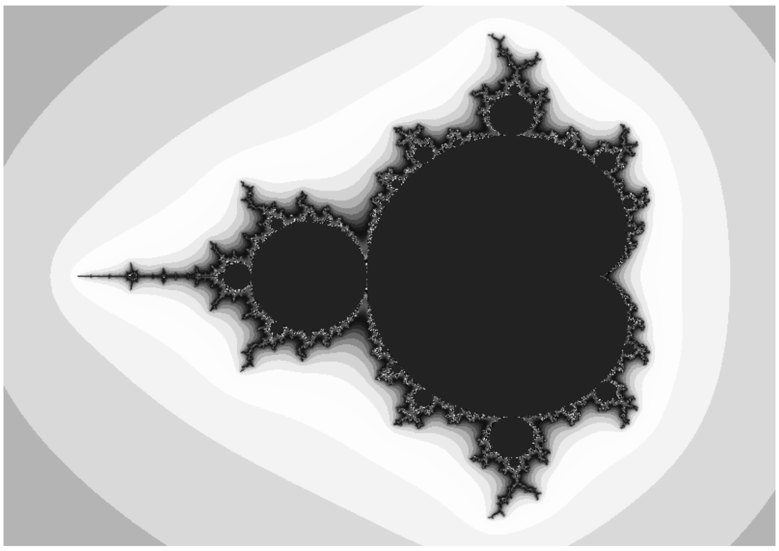

Figure 2: Mandelbrot image generated with ac=-0.65, bc=0.00, hght=2.5

In this exercise we will implement a parallel version of a program that generates nice images of the mandelbrot set. In figure 2 a greyscale example is given. If you are interested in the mathematical background of fractals, you can find that **here**[2].

Assignment: A sequential program that solves the problem at hand is listed in appendix A.3 on page 8.

a. Parallelise the outer loop of the `mandelbrotSet` function (i.e. use static domain decomposition).

b. Make a new version of the program in which process 0 becomes a *master* process that sends tasks on request to *slave* processes.

Constraints: Modifications of the program should take place in `mandelbrot.c`; the other files are not to be modified.

Requirements: Time your parallelised program for 2, 4, 8, 12, 16 and 24 threads. Also time each process individually and determine the total computation time. Additionally, determine the real execution time. Run the program on several image sizes. Make sure that the measured times are part of your output.

[2]`http://en.wikipedia.org/wiki/Mandelbrot_set`

A | Appendix

A.1 | Primes

Compile the program using the -O3 flag for strict optimisation options. Compilation can be done with the GNU Compiler Collection.

prime.c

```c
// File: prime.c
//
// A simple program for computing the amount of prime numbers within the
// interval [a,b].

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

#define FALSE 0
#define TRUE 1

static int isPrime (unsigned int p)
{
    int i, root;
    if (p == 1)
        return FALSE;
    if (p == 2)
        return TRUE;
    if (p % 2 == 0)
        return FALSE;

    root = (int)(1 + sqrt (p));
    for (i = 3; (i < root) && (p % i != 0); i += 2);
    return i < root ? FALSE : TRUE;
}

int main (int argc, char **argv)
{
    unsigned int i, a, b, cnt = 0;

    // Prints current date and user. DO NOT MODIFY
    system("date"); system("echo $USER");

    fprintf (stdout, "Enter two integer number a, b such that 1<=a<=b: ");
    fflush (stdout);
    scanf ("%u %u", &a, &b);

    if (a <= 2)
    {
        cnt = 1;
        a = 3;
    }
    if (a % 2 == 0)
        a++;

    for (i = a; i <= b; i += 2)
        if (isPrime (i)) {
            cnt++;
        }

    fprintf (stdout, "\n#primes=%u\n", cnt);
    fflush (stdout);
    return EXIT_SUCCESS;
}
```

A.2 | Contrast Stretching

Compile the program using the `-O3` flag for strict optimisation options. Compilation should be done using a MPI compiler. Note that the program consists of two source files and a single header file, but only one source file is listed below, as it contains the relevant functions. Running the program should also be done within the MPI runtime environment, and requires two arguments:

```
mpirun -np <num_threads> <executable> <input_image> <output_image>
```

contrast.c

```c
// File: contrast.c
// Written by Arnold Meijster and Rob de Bruin.
// Restructured by Yannick Stoffers.
//
// A simple program for contrast stretching PPM images.

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "image.h"

// Stretches the contrast of an image.
void contrastStretch (int low, int high, Image image)
{
    int row, col, min, max;
    int width = image->width, height = image->height, **im = image->imdata;
    float scale;

    // Determine minimum and maximum.
    min = max = im[0][0];
    for (row = 0; row < height; row++)
    {
        for (col = 0; col < width; col++)
        {
            min = im[row][col] < min ? im[row][col] : min;
            max = im[row][col] > max ? im[row][col] : max;
        }
    }

    // Compute scale factor.
    scale = (float)(high-low) / (max-min);

    // Stretch image.
    for (row = 0; row < height; row++)
    {
        for (col = 0; col < width; col++)
        {
            im[row][col] = scale * (im[row][col] - min);
        }
    }
}

int main (int argc, char **argv)
{
    Image image;

    // Prints current date and user. DO NOT MODIFY
    system("date"); system("echo $USER");

    // Initialise MPI environment.
    MPI_Init (&argc, &argv);

    if (argc != 3)
    {
        printf ("Usage: %s input.pgm output.pgm\n", argv[0]);
        exit (EXIT_FAILURE);
    }

    image = readImage (argv[1]);
    contrastStretch (0, 255, image);
    writeImage (image, argv[2]);

    freeImage (image);

    // Finalise MPI environment.
    MPI_Finalize ();
    return EXIT_SUCCESS;
}
```

A.3 Mandelbrot

Compile the program using the `-O3` flag for strict optimisation options. Compilation should be done using a MPI compiler. Note that the program consists of two source files and a single header file. Also note that `image.c` and `image.h` are not interchangeable with those of the other assignments.

mandelbrot.c

```c
// File: mandelbrot.c
// Written by Arnold Meijster and Rob de Bruin
// Restructured by Yannick Stoffers
//
// A simple program for computing images of the Mandelbrot set.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include "fractalimage.h"

#define WIDTH 4096
#define HEIGHT 3072
#define MAXITER 3000

int rank, size;

// Function for computing mandelbrot fractals.
void mandelbrotSet (double centerX, double centerY, double scale, Image image)
{
    int w = image->width, h = image->height, **im = image->imdata;
    double a, b;
    double x, y, z;
    int i, j, k;

    for (i = 0; i < h; i++)
    {
        b = centerY + i * scale - ((h / 2) * scale);
        for (j = 0; j < w; j++)
        {
            a = centerX + j * scale - ((w / 2) * scale);
            x = a;
            y = b;
            k = 0;
            while ((x * x + y * y <= 100) && (k < MAXITER))
            {
                z = x;
                x = x * x - y * y + a;
                y = 2 * z * y + b;
                k++;
            }
            im[i][j] = k;
        }
    }
}

int main (int argc, char **argv)
{
    Image mandelbrot;

    // Prints current date and user. DO NOT MODIFY
    system("date"); system("echo $USER");

    // Initialise MPI environment.
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    mandelbrot = makeImage (WIDTH, HEIGHT);
    mandelbrotSet (-0.65, 0, 2.5 / HEIGHT, mandelbrot);

    writeImage (mandelbrot, "mandelbrot.ppm", MAXITER);

    freeImage (mandelbrot);

    // Finalise MPI environment.
    MPI_Finalize ();
    return EXIT_SUCCESS;
}
```