Parallel Computing - MPI

Last edit: May 16, 2017

Introduction

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

There are two 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 | Volume Rendering

This assignment is about rendering 3D objects. Rather, 2D projections of 3D objects. A couple of datasets are given, each of which contains the data obtained by a CT-scanner. The type of renderer that is implemented is a so-called *orthogonal maximum intensity projection*.

Assignment: A sequential program that solves the problem at hand is given and listed in appendix A.1 on page 4. You are asked to parallelise the program in any way you can think of.

Constraints: Modifications of the program should take place in `render.c`; the other files are not to be modified. Make sure only the process with $rank = 0$ accesses the file system.

Minimum 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 both datasets (`footsmall` and `foot`).

2 | Wave

In a previous session, we solved the wave equation using OpenMP. We will now solve the wave equation using MPI. For convenience, the introductory text on the wave equation is given again below.

The wave equation is an important second-order linear partial differential equation that describes the propagation of a variety of waves, such as sound waves and water waves. It arises in fields such as acoustics and fluid dynamics.

In its simplest form, the 2D wave equation refers to a scalar function $u(x, y, t)$ that satisfies the property given in equation 1, in which c is a fixed constant equal to the propagation speed of the wave. Furthermore, the first and second-order partial derivatives with respect to x are approximated in equations 2 and 3 respectively.

$$c^2 \left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \right) = \frac{\partial^2 u(x,y,t)}{\partial t^2} \tag{1}$$

$$\frac{\partial u(x,y,t)}{\partial x} \approx \frac{u(x + \frac{\Delta x}{2}, y, t) - u(x - \frac{\Delta x}{2}, y, t)}{\Delta x}. \tag{2}$$

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} = \frac{\partial}{\partial x} \left(\frac{\partial u(x,y,t)}{\partial x} \right) \approx \frac{u(x + \Delta x, y, t) - 2u(x,y,t) + u(x - \Delta x, y, t)}{(\Delta x)^2}. \tag{3}$$

Introducing the discretisation $u[i, j, k] = u(x_0 + i\Delta x, y_0 + j\Delta y, t_0 + k\Delta t)$ yields the result in equation 4.

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} \approx \frac{1}{(\Delta x)^2}(u[i+1, j, k] - 2u[i, j, k] + u[i-1, j, k]). \tag{4}$$

For the other second-order partial derivatives we do the same. We sample the grid with equal spatial resolution for x and y by introducing $\Delta x = \Delta y = \delta$. After substitution in equation 1 and some calculus we find the recurrence equation listed in equation 5, in which $\lambda = \frac{c\Delta t}{\delta}$.

$$u[i, j, k+1] = \lambda^2 \left(u[i-1, j, k] + u[i+1, j, k] + u[i, j-1, k] + u[i, j+1, k] - 4u[i, j, k] \right) + 2u[i, j, k] - u[i, j, k-1] \tag{5}$$

In order to achieve a stable approximation, it is necessary that $|\lambda| \leq \frac{1}{2}\sqrt{2}$. Also, the recurrence requires the initial values to be supplied: $u[i, j, 0]$ and $u[i, j, 1]$ for all i, j.

Assignment: A sequential program that solves the problem at hand is given and listed in appendix A.2 on page 8. You are asked to parallelise the program. Decide on how to distribute the data.

Minimum 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 several different input values (see the `README` file on how to do so).

A | Appendix

A.1 | Volume Rendering

Compile the program using the `-O3` flag for strict optimisation options. Compilation can be done with the GNU compiler collection. Note that the program consists of three source files and two header files. Also note that the `image.c` and `image.h` files are interchangeable with those of the contrast stretching assignment, but not with those of the mandelbrot assignment. Running the program requires one argument:

```
<executable> <input_volume>
```

Once the program has returned a set of images, it suffices to run the `peregrine_makemovie` script to create a simple gif out of the separate frames.

render.c

```c
// File: render.c
// Written by Arnold Meijster and Rob de Bruin.
// Restructured by Yannick Stoffers.
//
// A simple orthogonal maximum intensity projection volume render.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "image.h"
#include "volume.h"

#define NFRAMES 360

void rotateVolume (double rotx, double roty, double rotz, Volume volume, Volume rotvolume)
{
    // Rotate the volume around the x-axis with angle rotx, followed by a
    // rotation around the y-axis with angle roty, and finally around the
    // z-axis with angle rotz. The rotated volume is returned in rotvolume.
    int i, j, k, xi, yi, zi;
    int width = volume->width;
    int height = volume->height;
    int depth = volume->depth;
    byte ***vol = volume->voldata;
    byte ***rot = rotvolume->voldata;
    double x, y, z;
    double sinx, siny, sinz;
    double cosx, cosy, cosz;

    for (i = 0; i < depth; i++)
        for (j = 0; j < height; j++)
            for (k = 0; k < width; k++)
                rot[i][j][k] = 0;

    sinx = sin (rotx); siny = sin (roty); sinz = sin (rotz);
    cosx = cos (rotx); cosy = cos (roty); cosz = cos (rotz);
    for (i = 0; i < depth; i++)
    {
        for (j = 0; j < height; j++)
        {
            for (k = 0; k < width; k++)
            {
                xi = j - height / 2;
                yi = k - width / 2;
                zi = i - depth / 2;

                // Rotation around x-axis.
                x = (double)xi;
                y = (double)(yi * cosx + zi * sinx);
```

```
                z = (double)(zi * cosx - yi * sinx);
                xi = (int)x;
                yi = (int)y;
                zi = (int)z;
                // Rotation around y-axis.
                x = (double)(xi * cosy + zi * siny);
                y = (double)(yi);
                z = (double)(zi * cosy - xi * siny);
                xi = (int)x;
                yi = (int)y;
                zi = (int)z;

                // Rotation around z-axis.
                x = (double)(xi * cosz + yi * sinz);
                y = (double)(yi * cosz - xi * sinz);
                z = (double)zi;

                xi = (int)(x + height / 2);
                yi = (int)(y + width / 2);
                zi = (int)(z + depth / 2);
                if ((xi >= 0) && (xi < height) && (yi >= 0) && (yi < width) && (zi >= 0) && (zi <
                    depth))
                    rot[zi][xi][yi] = vol[i][j][k];
            }
        }
    }
}

void contrastStretch (int low, int high, Image image)
{
    // Stretch the dynamic range of the image to the range [low..high].
    int row, col, min, max;
    int width = image->width, height = image->height, **im = image->imdata;
    double 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 = (double)(high - low) / (max - min);

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

void orthoGraphicRenderer (Volume volume, Image image)
{
    // Render image from volume (othographic maximum intensity projection).
    int i, j, k;
    int width = volume->width;
    int height = volume->height;
    int depth = volume->depth;
    int **im = image->imdata;
    byte ***vol = volume->voldata;

    for (i = 0; i < height; i++)
```

```
        for (j = 0; j < width; j++)
            im[i][j] = 0;

    for (i=0; i<depth; i++)
        for (j=0; j<height; j++)
            for (k=0; k<width; k++)
                im[j][k] += vol[i][j][k];

    contrastStretch (0, 255, image);
}

void smoothImage (Image image, Image smooth)
{
    int width = image->width, height = image->height;
    int **im = image->imdata, **sm = smooth->imdata;
    int i, j, ii, jj, sum, cnt;

    for (i = 0; i < height; i++)
    {
        for (j = 0; j < width; j++)
        {
            cnt = 0;
            sum = 0;
            for (ii = i-1; ii <= i+1; ii++)
            {
                if ((ii >= 0) && (ii < height))
                {
                    for (jj = j-1; jj <= j+1; jj++)
                    {
                        if ((jj >= 0) && (jj < width) && (im[ii][jj] != 0))
                        {
                            cnt++;
                            sum += im[ii][jj];
                        }
                    }
                }
            }
            sm[i][j] = cnt == 0 ? 0 : sum / cnt;
        }
    }
}

void computeFrame(int frame, double rotx, double roty, double rotz, Volume vol, Volume rot, Image
     image, Image smooth)
{
    char fnm[256];
    rotateVolume (rotx, roty, rotz, vol, rot);
    orthoGraphicRenderer (rot, image);
    smoothImage (image, smooth);
    sprintf (fnm, "frame%04d.pgm", frame);
    writeImage (smooth, fnm);
}

int main (int argc, char **argv)
{
    int width, height, depth;
    Volume vol, rot;
    Image im, smooth;
    int frame;
    double rotx, roty, rotz;

    if (argc != 2)
    {
        fprintf (stderr, "Usage: %s <volume.vox>\n", argv[0]);
        exit (EXIT_FAILURE);
    }
```

```
    vol = readVolume (argv[1]);
    width = vol->width;
    height = vol->height;
    depth = vol->depth;
    rot = makeVolume (width, height, depth);

    im  = makeImage (width, height);
    smooth = makeImage (width, height);

    // Compute frames.
    for (frame = 0; frame < NFRAMES; frame++)
    {
        rotx = roty = rotz = 0;
        switch (3 * frame / NFRAMES)
        {
            case 0:
                rotx = 6 * 3.1415927 * frame / NFRAMES;
                break;
            case 1:
                roty = 6 * 3.1415927 * frame / NFRAMES;
                break;
            case 2:
                rotz = 6 * 3.1415927 * frame / NFRAMES;
                break;
        }
        computeFrame (frame, rotx, roty, rotz, vol, rot, im, smooth);
    }

    freeVolume (rot);
    freeVolume (vol);
    freeImage (im);
    freeImage (smooth);

    return EXIT_SUCCESS;
}
```

A.2 | Wave

Compile the program using the -O3 flag for strict optimisation options. Compilation can be done using the GNU compiler collection. See the README file for execution instructions.

Once the program has returned a set of images as result, it suffices to run the peregrine_makemovie script to create a simple movie out of the separate frames.

wave.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include "waveio.h"

typedef float real;
typedef unsigned char byte;

#define ABS(a) ((a)<0 ? (-(a)) : (a))

static real ***initialize (int N, int NFRAMES, real dx, real *dt, real v, int nsrc, int **src, int **ampl);
static void boundary (real ***u, int N, int iter, int nsrc, int *src, int *ampl, real timespacing);
static void solveWave (real ***u, int N, int NFRAMES, int nsrc, int *src, int *ampl, real gridspacing, real timespacing, real speed);

static real ***initialize (int N, int NFRAMES, real dx, real *dt, real v, int nsrc, int **src, int **ampl)
{
    real ***u;
    int i, j, k;
    real lambda;

    // Determine lambda, adjust timespacing if needed.
    lambda = v * *dt / dx;
    if (lambda > 0.5 * sqrt(2))
    {
        fprintf (stdout, "Error: Convergence criterion is violated.\n");
        fprintf (stdout, "speed * timespacing / gridspacing = ");
        fprintf (stdout, "%lf * %lf / %lf = %f > 0.5 * sqrt(2)\n", v, *dt, dx,lambda);
        *dt = (real)(dx * sqrt (2) / (2 * v));
        fprintf (stdout, "Timestep changed into: timespacing=%lf\n", *dt);
        fflush (stdout);
    }

    // Draw n random locations of wave sources.
    srand (time (NULL)); // Initialize random generator with time.
    *src = malloc (2 * nsrc * sizeof (int));
    *ampl = malloc (nsrc * sizeof (int));
    for (i = 0; i < nsrc; i++)
    {
        (*src)[2 * i] = random () % N;
        (*src)[2 * i + 1] = random () % N;
        (*ampl)[i] = 1; // Change this to modify the amplitude of the waves.
    }

    // Allocate memory for u.
    u = malloc (NFRAMES * sizeof (real **));
    for (k = 0; k < NFRAMES; k++)
    {
        u[k] = malloc (N * sizeof (real *));
        u[k][0] = malloc (N * N * sizeof (real));
        for (i = 1; i < N; i++)
            u[k][i] = &(u[k][0][i * N]);
    }

```

```c
    // Initialize first two time steps.
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            u[0][i][j] = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            u[1][i][j] = 0;

    return u;
}

static void boundary (real ***u, int N, int iter, int nsrc, int *src, int *ampl, real timespacing)
{
    // Initialise new frame by inserting boundary values and wave sources.
    int i, j;
    real t = iter * timespacing;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            u[iter][i][j] = 0;

    for (i = 0; i < nsrc; i++)
        u[iter][src[2 * i]][src[2 * i + 1]] = (real)(ampl[i] * sin (t));
}

static void solveWave (real ***u, int N, int NFRAMES, int nsrc, int *src, int *ampl, real
     gridspacing, real timespacing, real speed)
{
    // Computes all NFRAMES consecutive frames.
    real sqlambda;
    int i, j, iter;
    sqlambda = speed * timespacing / gridspacing;
    sqlambda = sqlambda * sqlambda;

    for (iter = 2; iter < NFRAMES; iter++)
    {
        boundary (u, N, iter, nsrc, src, ampl, timespacing);
        for (i = 1; i < N - 1; i++)
            for (j = 1; j < N - 1; j++)
                u[iter][i][j] += sqlambda * (u[iter - 1][i + 1][j] + u[iter - 1][i - 1][j] + u[iter -
                    1][i][j + 1] + u[iter - 1][i][j - 1]) + (2 - 4 * sqlambda) * u[iter - 1][i][j] -
                    u[iter - 2][i][j];
    }
}

int main (int argc, char **argv)
{
    real ***u;        // Holds the animation data.
    int N, NFRAMES;   // Dimensions and amount of frames.
    int *src = NULL;  // Source coordinates.
    int *ampl = NULL; // Amplitudes of sources.
    int n;            // Number of sources.
    int bw;           // Colour(=0) or greyscale(=1) frames.
    real dt;          // Time spacing (delta time).
    real dx;          // Grid spacing (distance between grid cells.
    real v;           // Velocity of waves.

    struct timeval start, end;
    double fstart, fend;

    // Default values.
    n = 10;       // Number of sources.
    bw = 0;       // Black and white disabled.
    NFRAMES = 100; // Number of images/frames.
    N  = 300;     // Grid cells (width,height).
```

```
    dt = 0.1;      // Timespacing.
    dx = 0.1;      // Gridspacing.
    v  = 0.5;      // Wave velocity.

    // Parse command line options.
    parseIntOpt (argc, argv, "-f", &NFRAMES);
    parseIntOpt (argc, argv, "-src", &n);
    parseIntOpt (argc, argv, "-bw", &bw);
    parseIntOpt (argc, argv, "-n", &N);
    parseRealOpt (argc, argv, "-t", &dt);
    parseRealOpt (argc, argv, "-g", &dx);
    parseRealOpt (argc, argv, "-s", &v);

    // Init.
    u = initialize (N, NFRAMES, dx, &dt, v, n, &src, &ampl);

    // Start timer.
    fprintf (stdout, "Computing waves\n");
    gettimeofday (&start, NULL);

    // Render all frames.
    solveWave (u, N, NFRAMES, n, src, ampl, dx, dt, v);

    // Stop timer; compute flop/s.
    gettimeofday (&end, NULL);
    fstart = (start.tv_sec * 1000000.0 + start.tv_usec) / 1000000.0;
    fend = (end.tv_sec * 1000000.0 + end.tv_usec) / 1000000.0;
    fprintf (stdout, "wallclock: %lf seconds (ca. %5.2lf Gflop/s)\n", fend - fstart, (9.0 * N * N *
        NFRAMES / (fend - fstart)) / (1024 * 1024 * 1024));

    // Save separate images.
    fprintf (stdout, "Saving frames\n");
    stretchContrast (u, N, NFRAMES);
    saveFrames (u, N, NFRAMES, bw);

    fprintf (stdout, "Done\n");
    return EXIT_SUCCESS;
}
```