Homework 1

SF2568 - Parallel Computation for Large Scale Problems

Lukas Bjarre - lbjarre@kth.se

Q1: Difference between a *process* and a *processor*

A *process* is defined by the instructions that are needed to compute a problem. A *processor* refers to the physical hardware which executes a process.

Q2: Performance calculation

10% of the time the computer is operating at single processor performance, i.e. 2 Gflops. The remaining 90% of the time all 100 processors are computing in parallel at a performance of 200 Gflops. The total performance will therefore be

$$0.1 \cdot 2 + 0.9 \cdot 200 = 180.2 \text{ Gflops} \tag{1}$$

Q3: Is a system efficiency larger than 100% possible?

The system efficiency η_P for P processors is given by

$$\eta_P = \frac{S_P}{P} = \frac{T_s^*}{PT_P} \tag{2}$$

where T_s^* is the fastest serial runtime and T_P is the parallel runtime for P processes. Assuming (unrealistically) no communication in the parallel implementation, the optimal parallel runtime will be $T_P^* = T_s^* P^{-1}$. This implementation inserted into (2) would result in a system efficiency of 100%. However, this is a lower bound for the parallel runtime, hence $\eta_P \leq 1$.

Q4: Modified Parallel Rank Sort

Since we have 10 times as many elements as processors 10 elements can be allocated to each processor. Each processor has to then loop through each of the allocated elements and execute the original Parallel Rank Sort for each of them. The resulting rank arrays then needs to be shared to the rest of the processors, which can be done for example in a recursive doubling procedure.

Q5: Parallel speedup

(a) *What fraction of the serial runtime is spent in serial procedures?*

We have $T_s^* = 64$ s and $T_P = 22$ s for $P = 8$. The serial fraction of the program depends on which law you apply to the numbers. The serial fraction as given by Amdahl's law is

$$\begin{aligned} f &= \frac{PT_P - T_s^*}{T_s^* (P-1)} \\ &= \frac{8 \cdot 22 - 64}{64 \cdot 7} \\ &= \frac{1}{4} = 0.25 \end{aligned} \tag{3}$$

The serial fraction as given by Gustafson's law is

$$\begin{aligned} f' &= \frac{PT_P - T_s^*}{T_P (P-1)} \\ &= \frac{8 \cdot 22 - 64}{22 \cdot 7} \\ &= \frac{8}{11} \approx 0.72 \end{aligned} \tag{4}$$

(b) ***Determine the parallel speedup***

The speedup once again depends on which law you apply. The speedup as given by Amdahl is

$$\begin{aligned} S_P &= \frac{1}{(1-f)\,P^{-1} + f} \\ &= \frac{1}{0.75P^{-1} + 0.25} \end{aligned} \tag{5}$$

For $P = 8$ we have a speedup of $S_8 \approx 2.9$. We also see that even if we let $P \to \infty$, the speedup would only converge to $S_\infty = 4$.

The scaled speedup as given by Gustafson is

$$\begin{aligned} S'_P &= f' + (1-f')P \\ &= \frac{8 + 3P}{11} \end{aligned} \tag{6}$$

For $P = 8$ we have a scaled speedup of $S'_8 \approx 2.9$, i.e. the same as predicted by Amdahl. However, unlike S_P, S'_P grows unbounded as $P \to \infty$.

(c) ***Parallel speedup on three phased program***

We notate the serial runtimes for each program phase as T_{pi}, where $i = 1, 2, 3$ describes the different phases. We also know the relation between each phase's serial runtime and the optimal serial runtime as $T_{p1} = 0.2T^*_s$, $T_{p2} = 0.2T^*_s$, and $T_{p3} = 0.6T^*_s$. In the parallel version of the program each program phase will be sped up by a optimal factor of k^*_{pi} for each program phase. We will assume perfect speedup for each phase given the given optimal process count, i.e. $k^*_{pi} = P^{*-1}_{pi}$ for optimal process count P^{*-1}_{pi}. The total speedup will be

$$\begin{aligned} S_P &= \frac{T^*_s}{T^*_s\left(0.2k^*_{p1} + 0.2k^*_{p2} + 0.6k^*_{p3}\right)} \\ &= \frac{1}{\frac{1}{25} + \frac{1}{50} + \frac{3}{75}} \\ &= \frac{750}{75} \\ &= 10 \end{aligned} \tag{7}$$

Q6: Parallel Mandelbrot implementation

Implementation

My parallel implementation of the mandelbrot set calculation is similair to the one presented in the lecture slides, except with some small alterations to increase usability and performance. The code can be found in the appendix.

The first change is how the data is distributed between the processors. The version presented in the lecture slides separated the complex plane grid into vertical slices, i.e. common x-coordinates and iterating over the y-coordinates. However, when the data is saved we want to save rows and not columns, i.e. common y-coordinates and iterating over the x-coordinates. I therefore changed the data distribution to horizontal strips of the complex plane grid instead to better accomodate the saved data structure.

The second change has also to do with the saved data structure. I wanted to make the saved file as similair to the complex plane as possible. Since the complex plane usually is plotted with positive complex values on the top and the negatives on the bottom, the first row of the saved file should contain the highest complex values and the last row the file the lowest. Therefore, the sign of the starting y-position and the direction in which the complex values was iterated in was flipped.

Finally, the viewport in which the complex plane is evaluated was generalized. The method used in the lecture slide assumed a viewport of $[-b,\, b] \times [-b,\, b]$, i.e. a square with sides $2b$ placed with its center at the origin. This was changed to a viewport of $[x_s,\, x_s + x_w] \times [y_s,\, y_s - y_h]$, i.e. a more general rectangle placed with its upper left corner at $(x_s,\, y_s)$ and with width x_w and height y_h.

Output Images

Below are some images produced by the code.

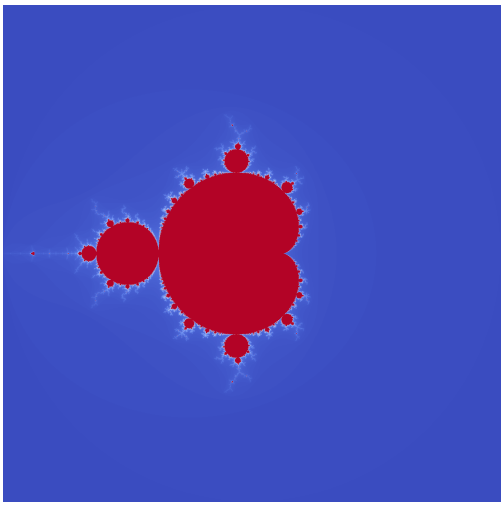
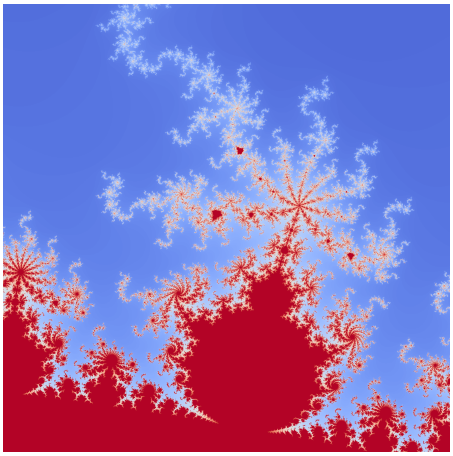
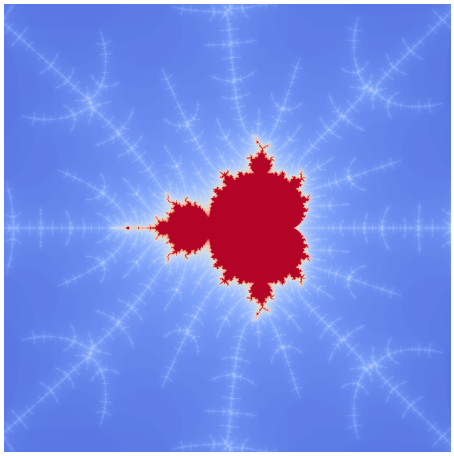


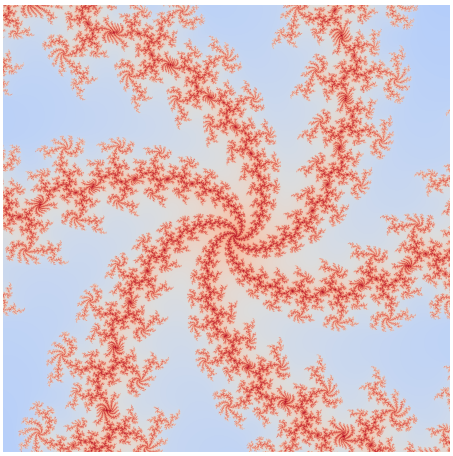
Figure 1: Plot of the Mandelbrot set within $[-2, 2] \times i[-2, 2]$.



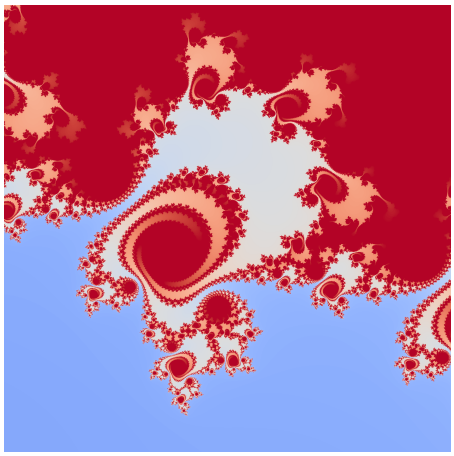
(a) $[-0.970, -0.995] \times i[0.245, 0.270]$



(b) $[-1.50, -1.55] \times i[-0.025, 0.025]$



(c) $[0.3981, 0.3982] \times i[0.23312, 0.23322]$



(d) $[-0.1555, -0.1535] \times i[0.6522, 0.6542]$

Figure 2: Various zoomed in plots on the Mandelbrot set.

Appendix A: Parallel Mandelbrot Sourcecode

Below is the source code for the program which implements the parallel version of the Mandelbrot set calculation. The code is quite lengthy, so it is broken down into smaller snippets containing separate functions together with a small annotation describing its function.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  #define NUMB_COLORS 255
6  #define ESC_DIST_SQ 4.0
7  #define PIXEL_W   2048
8  #define PIXEL_H   2048
```

Library imports and our constants. The constants define how the color picking is set by the maximum iteration count (the number of possible colors) and the squared escape distance that for the semi-stable convergence definition. Constants defining the pixel size of the viewport are also set.

```
10 int mandelbrot(double c_real, double c_imag)
11 {
12     int i = 0;
13     double z_real = c_real;
14     double z_real_sq = c_real * c_real;
15     double z_imag = c_imag;
16     double z_imag_sq = c_imag * c_imag;
17
18     while (z_real_sq + z_imag_sq < ESC_DIST_SQ && i < NUMB_COLORS) {
19         z_imag = 2 * z_real * z_imag + c_imag;
20         z_real = z_real_sq - z_imag_sq + c_real;
21         z_real_sq = z_real * z_real;
22         z_imag_sq = z_imag * z_imag;
23         ++i;
24     }
25
26     return i;
27 }
```

The function which assigns colors to each pixel. This implementation uses slightly more memory than the straight-forward approach, but saves a couple of operations for each loop.

```
29 void mandelbrot_grid(int *M, int y_offset, int process_height,
30                       double x_start, double y_start, double h_w)
31 {
32     int i, j;
33     double x, y;
34
35     double dx = h_w/(PIXEL_W - 1);
36     double dy = -h_w/(PIXEL_H - 1);
37
38     for (j = 0; j < process_height; ++j) {
39         y = (j + y_offset) * dy + y_start;
40         for (i = 0; i < PIXEL_W; ++i) {
41             x = i * dx + x_start;
42             *M++ = mandelbrot(x, y);
43         }
44     }
45 }
```

Next we have the main calculation loop for each process. It iterates over the pixels assigned to the processor, converts each pixel to a complex number, and saves the calculated color value in the array M.

```
47 void write_to_file(FILE *fp, int *M, int process_height)
48 {
49     int i, j;
50     for (j = 0; j < process_height; ++j) {
51         for (i = 0; i < PIXEL_W; ++i) {
52             fprintf(fp, "%hhu ", *M++);
53         }
54         fprintf(fp, "\n");
55     }
56 }
```

The function for writing the results to a file is fairly basic.

```
58 int init_mpi(int *size, int *rank, int argc, char **argv)
59 {
60     int flag = 0;
61
62     MPI_Init(&argc, &argv);
63     MPI_Comm_size(MPI_COMM_WORLD, size);
64     MPI_Comm_rank(MPI_COMM_WORLD, rank);
65
66     if (PIXEL_H % *size) {
67         flag = -1;
68     }
69
70     return flag;
71 }
```

A small function for setting up all the MPI stuff outside of the main procedure. This function also checks to make sure that the number of processes evenly divide the pixel grid height. If not, it returns a negative flag to main.

```
73  int set_viewport(double *x_start, double *y_start, double *h_w,
74                   int argc, char **argv)
75  {
76      int flag = 0;
77      *x_start = -2.0;
78      *y_start = 2.0;
79      *h_w = 4.0;
80      if (argc == 4) {
81          sscanf(argv[1], "%lf", x_start);
82          sscanf(argv[2], "%lf", y_start);
83          sscanf(argv[3], "%lf", h_w);
84      }
85      else if (argc != 1) {
86          flag = -1;
87      }
88      return flag;
89  }
```

This function sets the variables defining the viewport, either through argv or as defaults if no arguments are passed. Returns a negative flag if argv is of incorrect length.

```
91  int main(int argc, char **argv)
92  {
93      int rank, size, y_off, process_h, vec_len, i, flag_mpi, flag_arg;
94      double x_start, y_start, h_w;
95      MPI_Status st;
96
97      flag_arg = set_viewport(&x_start, &y_start, &h_w, argc, argv);
98      flag_mpi = init_mpi(&size, &rank, argc, argv);
99      if (flag_arg != 0 || flag_mpi != 0) {
100         exit(1);
101     }
102
103     process_h = PIXEL_H / size;
104     vec_len = PIXEL_W * process_h;
105     y_off = rank * process_h;
106     int M[vec_len];
107     mandelbrot_grid(M, y_off, process_h, x_start, y_start, h_w);
108
109     if (rank != 0) {
110         MPI_Send(M, vec_len, MPI_INT, 0, 0, MPI_COMM_WORLD);
111     }
112     else {
113         FILE *fp = fopen("color.txt", "w");
114         write_to_file(fp, M, process_h);
115         for (i = 1; i < size; ++i) {
116             MPI_Recv(M, vec_len, MPI_INT, i, 0, MPI_COMM_WORLD, &st);
117             write_to_file(fp, M, process_h);
118         }
119         fclose(fp);
120     }
121
122     MPI_Finalize();
123     return 0;
124 }
```

The main procedure calls the initializing functions, sets up and calculates the mandelbrot for the process height, and sends everything to the 0 rank process. The 0 rank process saves some memory by overwriting the already allocated M-vector after writing its contents to the output file

Appendix B: Plotting Sourcecode

Below is the sourcecode for the Python script used to read the data produced by the C code and produce images.

```
#!/usr/bin/env python3

import sys
import numpy as np
import matplotlib.cm as cm
from PIL import Image

COLORMAP = 'coolwarm'
DEFAULT_PATH = 'color.txt'
SAVED_IMAGE_EXT = '.bmp'

def load_image(filepath):

    read_vals = []

    with open(filepath, 'r') as f:
        for row in f.readlines():
            read_vals.append(row.rstrip().split(' '))

    return np.array(read_vals, dtype=np.uint8)

if __name__ == '__main__':
    try:
        filepath = sys.argv[1]
    except IndexError:
        filepath = DEFAULT_PATH

    image_vals = load_image(filepath)
    cmap = cm.get_cmap(COLORMAP)
    image_vals = np.uint8(255 * cmap(image_vals))
    image = Image.fromarray(image_vals)

    savepath = filepath.split('.')[0] + SAVED_IMAGE_EXT
    image.save(savepath)
```