Homework 1 - Parallel Programming for Large Scale Problems - SF2568

Gabriel Carrizo

December 2017

1. Describe the difference between a process and a processor

> **Solution:** From Lecture 1 slides: A processor is the physical hardware that performs computations and has access to memory. Process is the software, or computational activity assigned to the processor.

2. A multiprocessor consists of 100 processors, each capable of a peak execution rate of 2 Gflops (i.e. 2×10^9 floating operations per second). What is the performance of the system as measured in Gflops when 10% of the code is sequential and 90% is parallelizable?

> **Solution:** Gustafsson's Law:
>
> $$T_1 = f'T_P + (1 - f')PT_P \tag{1}$$
>
> Where: T_p is the parallel execution time, P the number of processors and f' is the fraction of the code that is sequential. This results in the following:
>
> $$T_1 = 0.1 \times 2 \times 10^9 + 100(1 - 0.1) \times 2 \times 10^9 = 180.2\text{Gflops} \tag{2}$$

3. Is it possible for a system to have a system efficiency (η_P) of greater than 100%? Discuss.

> **Solution:** Best case scenario all our code is in parallel and trivial to parallel with little to no communication between the nodes. Given Gustafsson's law the scaled speedup becomes:
>
> $$S'_P = f' + (1 - f')P = P \tag{3}$$
>
> With this, the *parallel efficiency* η_P becomes the following:
>
> $$\eta_P = \frac{S_P}{P} = \frac{P}{P} = 1 = 100\% \tag{4}$$
>
> This means that our upper bound becomes 100%, and we can not get results that are better than this.

4. In the Parallel Rank Sort method presented in the lecture, the number of processors must be the same as the number of elements in the list. Assume that the number of elements in the list is actually ten times larger than the number of processors in the computer. Describe a modification to handle this situation.

Solution: One modification tot he code that would solve this problem is dividing the algorithms to work on n equal size chunks of the data.

5. 5. You are given some time on a new parallel computer. You run a program which parallelizes perfectly during certain phases, but which must run serially during others.

   (a) If the best serial algorithm runs in 64s on one processor, and your parallel algorithm runs in 22s on 8 processors, what fraction of the serial running time is spent in the serial section?

   (b) Determine the parallel speedup.

   (c) You have another program that has three distinct phases. The first phase runs optimally on 5 processors; this is, performance remains constant on 6 or more processors. The second phase runs optimally on 10 processors, and the third on 15. The phases consume 20%, 20%, and 60% of the serial running time, respectively. What is the speedup on 15 processors?

Solution: a) With Amdahl's Law:

$$S_P = \frac{P}{1 + (P-1)f} = \frac{64}{22} \Rightarrow f = 0.25 = 25\% \tag{5}$$

Gustafsson's Law:

$$S'_P f' + (1 + f')P = \frac{64}{22} \Rightarrow f' = 0.73 = 73\% \tag{6}$$

b) Parallel speedup:

$$S_P = \frac{T_S}{T_P} = \frac{64}{22} = 2.91 \approx 3 \tag{7}$$

c) Amdahl's Equation gives us the following:

$$T_{5,10,15} = \frac{f_5}{5}T_1 + \frac{f_{10}}{10}T_1 + \frac{f_{15}}{15}T_1 \tag{8}$$

$$T_{5,10,15} = \frac{0.2}{5}T_1 + \frac{0.2}{10}T_1 + \frac{0.6}{15}T_1 = 0.1T_1 \tag{9}$$

$$T_S = fT_1 = (f_5 + f_{10} + f_{15})T_1 = T_1 \tag{10}$$

$$S_P = \frac{T_S}{T_P} = \frac{T_1}{0.1T_1} = 10 \tag{11}$$

6. For the following problem, you need to have access to MPI on one of the platforms (preferably tegner). Implement the Mandelbrot algorithm in a parallel environment!

...

(a) Implement the Mandelbrot program using MPI. You can assume that the number of processors divides the number of columns evenly. (Test this in your program!!)

(b) Reproduce the figure from the lecture notes. (You may play around with different palettes in matlab.)

(c) Magnify some interesting parts of the figure. Do this by computing only parts of the complete picture using higher resolutions.

Solution: a) See appendix for implementation

b) See appendix Figure 1. Note that my images were reconstructed in Python (not matlab) with output from code in c.

c) See appendix, figs. 2, 3a and 3b

Appendix A1

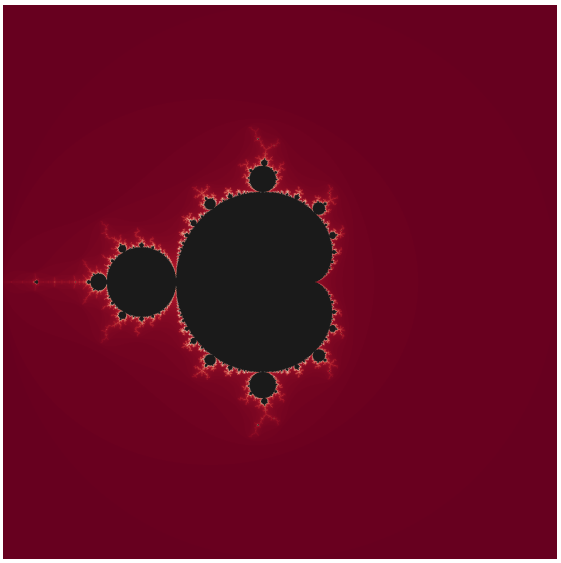


Figure 1: Basic mandelbrot with no zoom

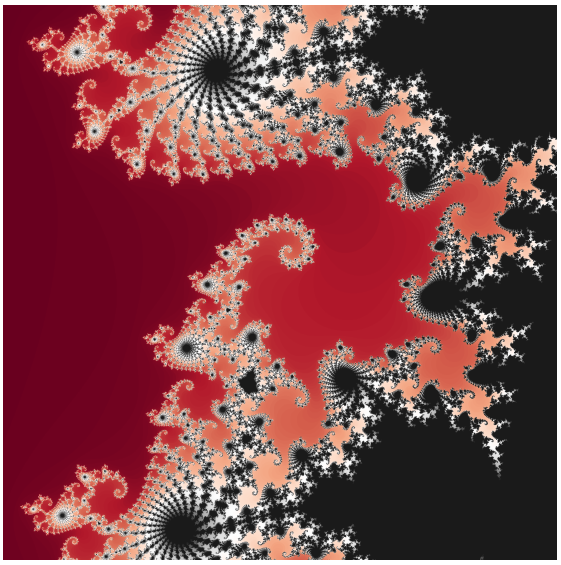
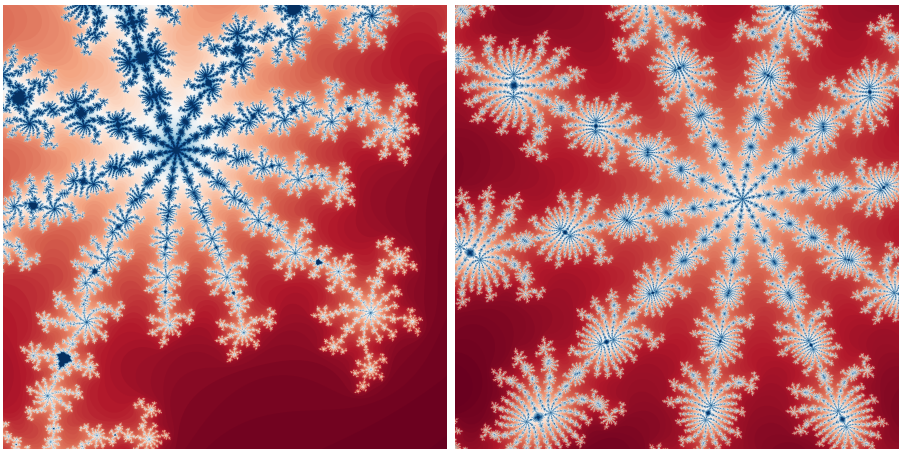


Figure 2: Zoom in on the ridge between the two main "blobs".



(a) N=127 (b) N=255

Figure 3: These two images are of the same region but the right most is zoomed in on the center of the left figure and number of iterations is changed from 127 to 255.

Appendix A.2

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#include <math.h>
#include <complex.h> //must be compiled with -lm flag
#include <unistd.h>
#include <stdlib.h>

unsigned char pixel_value(double complex d, double b, unsigned char N){
  /*
  forumla from lecture notes: z = 0 initially,
  z = z*z + d, where d is the complex coordinates of the point (re,im)
  N is the maximum number of allowed max iterations
  b = maximum radius of z.

  out: the number of iterations required for z > N
  */

  complex double z = 0;
  unsigned char count = 0;

  while (cabs(z)<b && count < N) {
    z = z*z + d;
    count++;
  }
  return count;
}

int main(int argc, char **argv) {
  int rank, size, tag, rc, i, j, x, y;
  double b;
  MPI_Status status;
  char message[20];
  tag = 100;

  rc = MPI_Init(&argc, &argv);
```

```
37    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
38    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
39
40    //windowing parameters:
41    long int h;
42    long int w;
43    unsigned char N = 255;
44
45    if (argc < 2){
46      h = 512;
47      w = 512;
48    }else{
49      h = atoi(argv[1]); //pixel height
50      w = atoi(argv[1]); //pixel width
51    }
52    int partition_width = h/size;
53
54    unsigned char partition[h*partition_width];
55
56    b = 2;
57
58    //exit if the number of processors do not divides the number of columns evenly
59    int mod = h%size;
60    if (mod != 0) {
61      printf("Uneven distribution of columns per process\n");
62      exit(1);
63    }
64
65    printf("%d\n", mod);
66    double zoom = 0.0078125;
67    //scaling and centering.
68    double dx = zoom*b/(w-1); //step size
69    double dy = zoom*b/(h-1);
70
71    double x_offset = 1.25;//2.057; // offsets
72    double y_offset = 1.85;//2.656;
73
74    if (rank == 0) {
75      printf("Number of nodes available: %d\n", size);
76    }
77    printf("Node with rank %d checking in.\n", rank);
78
79    double complex d;
80    double im;
81    double re;
82    int n = 0;
83    for (x = rank*partition_width; x <partition_width*(rank+1); x++) {
84      re = x*dx-b+ x_offset;
85      for (y = 0; y < h; y++) {
86        im = y*dy-b + y_offset;
87        d = re + I*im;
88        partition[n] = pixel_value(d, b, N);
89        ++n;
90      }
91    }
92
93    printf("Node %d finalized, gathering data. \n",rank);
94
95    unsigned char map[h*w];
96    rc = MPI_Gather(partition, partition_width*h, MPI_UNSIGNED_CHAR, &map,
       partition_width*h, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
97
98  if (rank == 0) {
99    FILE *fp;
```

```
100     int count = 0;
101     fp = fopen(argv[2],"w");
102     for (int x = 0; x < w; x++) {
103       for (int y = 0; y < h; y++) {
104         fprintf(fp, "%hhu ", map[x*w+y]);
105       }
106       fprintf(fp, "\n");
107       //printf("LINE BREAK: %d\n", count);
108       count++;
109     }
110     fclose(fp);
111  }
112     rc = MPI_Finalize();
113     return 0;
114 }
```