KTH Royal Institute of Technology

SF2568
Parallel computations for Large-Scale problems

Year 2017 - 2018

Homework 1

Group report

Picó Oristrell Sandra 940531 – 2308 sandrapo@kth.se
Corominas Larsson Miquel Sven 920614 – 5998 miquell@kth.se

Professors :
Hanke Michael

February 9, 2018

Question 1: Describe the difference between a *process* and a *processor*.

Although the names are quite similar, the meaning of these two words is strongly different. In computing, a process is considered *an instance of a computer program that is being executed*.[1] At the same time, depending of the programming code that it contains the process may be made up of multiple execution threads or only one. On the other hand, the word processor refers to *the electronic circuit that performs the code operations*.[2] Nowadays, some computer systems have more than one processor. To sum up, we can state that we execute the processes through the electronic processors.

Question 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?

The performance of the system in Gflops can be obtained with the following expression:

$$\frac{2t}{0.1t + \frac{0.9t}{100}} \sim 18.34\text{Gflops} \tag{1}$$

Where t is the sequential running time. Furthermore, following Amdahl's law, we can state that the code is able of achieving a maximum speedup factor of 10.

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

Given that we are in a parallel environment it is possible to have an efficiency higher than 100%. The reason behind that is the super-linear speed-up.Nevertheless, we are going to start talking about the efficiency concept.

The parallel efficiency is defined by how effectively you are using the multiple processors. In that case, if you have a 100% of efficiency, this would only imply that you are having a *p* speedup factor using p processors. This is because the efficiency and the speedup are defined as follows:

$$Speedup = \frac{T_s(n)}{T_p(n,p)} \tag{2}$$

$$Efficiency = \frac{Speedup}{p} = \frac{T_s(n)}{p * T_p(n,p)} \tag{3}$$

And using the previous formulas is when we can start talking about the super-linear speed-up, which is the reason why is possible to achieve more than 100% of efficiency.

We achieve super-linear speed-up when the amount of work done for a group of processors is strictly less than the work performed for a unique processor. An example where we can demonstrate that is the following one:[3]

Imagine that you have a machine with 8 processors. Each processor has 1MB of cache and your program uses 6MB of data. With one processor you need to do a lot of "movements" between the processor, the RAM memory and the Cache. However, with the processors you only need to move the data between CPU and the cache. Then, you have 8 caches but each of them only holding 6/8 MB of data.That implies that any of the processors need to re-fill or/and empty their caches during the computation.And, in that case, you are achieving the super-linear speed up.

Question 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.

In the lecture, a parallel approach of ordering a set of numbers was proposed, in which the computation of iterating through the whole set of numbers and assigning the corresponding rank to each number to order could be parallelized by giving this task to one processor for each number.

Now, we assume that there are ten times more elements to order than processors in the computer. This can be easily taken into account by simply splitting the list of numbers in equal parts and assigning each part to a different processor. Each processor would now realize the same work for a set of numbers, and since the ranking is obtained iterating each number through the whole complete set, the resulting rank for each number would be valid.

Question 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?

(a)

This fraction can be determined with:

$$64 * f + (1 - f)\frac{64}{8} = 22 \Rightarrow f = 0.25 \tag{4}$$

(b)

The parallel speedup is determined as:

$$S_P = \frac{T_s^*}{T_P} = \frac{64s}{22s} \sim 2.9 \tag{5}$$

(c)

The parallel running time can be determined as:

$$T_P = 0.2\frac{T_s}{5} + 0.2\frac{T_s}{10} + 0.6\frac{T_s}{15} \tag{6}$$

Where T_s is the serial running time. The speedup can then be obtained as:

$$S = \frac{T_s}{0.2\frac{T_s}{5} + 0.2\frac{T_s}{10} + 0.6\frac{T_s}{15}} = \frac{1}{\frac{0.2}{5} + \frac{0.2}{10} + \frac{0.6}{15}} = 10 \tag{7}$$

Question 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.

(a):

Before programming Mandelbrot in a parallel way, we decided to do it in a sequential form.

In order to implement it sequentially we have focused on two different parts: understanding the algorithm and plotting the generated image using Matlab. As indicated in the description of the exercise, it is much easier to plot through Matlab. Precisely for this motive we have generated an ascii file in the C code and read it in Matlab.These codes are attached in the appendix section.

The algorithm implemented, as said above, is the Mandelbrot algorithm. This algorithm is the implementation of the Mandelbrot set, *a set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from z=0.* [4]

The code that generates this set is quite easy. On the one hand, you have to program the function that will compute the main condition of the Mandelbrot set; $f_c(z) = z^2 + c$. In our case, this function is called *cal pixel*.Then, through loops and setting the correct parameters to be sure that the area of vision is a proper one, we generate a matrix where each cell has the color associated to each pixel of the image. That is the reason why the resolution of the image will change depending of the dimensions of the matrix. And then, after filling all the matrix, we generate the file and we plot it using Matlab. The pseudocode corresponding to this code could be the following one:

Algorithm 1 Mandelbrot algorithm

```
1:  procedure MANDELBROT()
2:      b = 2;                                  ▷ Define initial values of all the variables.
3:      N = 256;
4:      w = 2048;
5:      h = 2048.
6:      color[w][h] = 0;
7:      int x = 0;
8:      while x < w do                          ▷ All rows of the matrix.
9:          dreal = x*dx -b;
10:         int y = 0;
11:         while y < h do                      ▷ All columns of the matrix.
12:             dimag = y*dy-b;
13:             d = dreal + 1I*dimag;
14:             color[x][y] = calpixel(d,b,N);  ▷ Calculate the color of each pixel.
15:             y = y + 1;
16:         x = x + 1;
17:     CreateFile();                           ▷ Function that generates the file to plot in Matlab
```

Then, once the algorithm was working correctly in serial, we discuss how to do it in parallel. The first thing that you have to consider when you are doing parallel programming is the task you will assign to each worker. Since this exercise basically only computes a matrix, we decided to divide the matrix into different partitions so that each worker computed one slice.

The number of workers is a parameter that you receive at the time of execution because it is chosen by the user. For this reason, the matrix is divided into different sub-parts based on the number of workers.

For example, if the number of workers is 3, we will have the main process and 2 workers that will compute the matrix.Then, as the split is done in columns, the matrix will be divided as following:

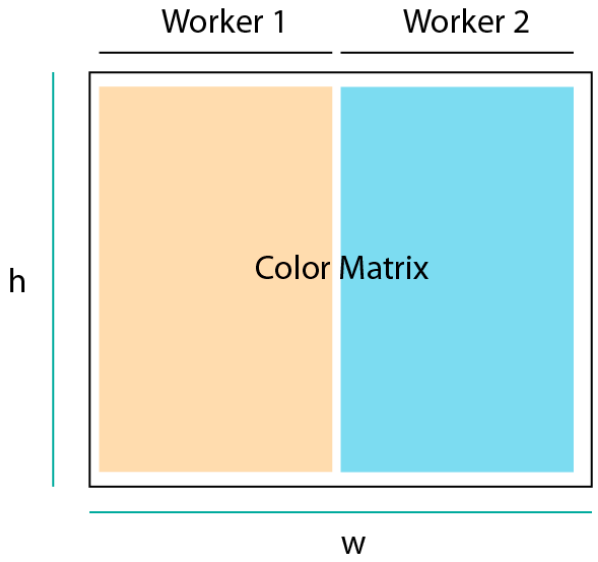

Figure 1: Example with P = 3

Once each worker is assigned a slice of work, we only need to implement it. As mentioned in the statement, we use the MPI library. However, since the code is quite simple, we only use the following MPI functions:

- **MPI-Init:** To initialize the MPI environment.
- **MPI-Comm-size:** To know how many workers we have.
- **MPI-Comm-rank:** To set the master process with rank $= 0$ and the other processes with rank > 0.
- **MPI-Send:** To send a message.
- **MPI-Recv:** To receive a message.

The pseudocode for the parallel implementation could be as follows:

Algorithm 2 Parallel Mandelbrot algorithm

```
1:  procedure ParallelMandelbrot()
2:      b = 2;                                   ▷ Define initial values of all the variables.
3:      N = 256;
4:      w = 2048;
5:      h = 2048.
6:      color[w][h] = 0;                         ▷ Defined as global variable
7:      int size,rank,rc = 0;                    ▷ Initialize MPI Environment.
8:      MPI-Init(NULL,NULL);
9:      MPI-Comm-size(MPI-COMM-WORLD, size);
10:     MPI-Comm-rank(MPI-COMM-WORLD, rank);
11:     slidewidth = width/(size-1);
12:     if rank == 0 then                        ▷ Master process
13:         int dest = 1;
14:         while dest <= (size − 1) do          ▷ Send the correspondent offset position to each worker.
15:             MPI-Send(offset)
16:             offset += slidewidth.
17:             dest+= 1;
18:         int dest = 1;
19:         while dest <= (size − 1) do
20:             MPI-Recv(color)                  ▷ Receive the updated matrix from each worker.
21:             dest+= 1;
22:         CreateFile();                        ▷ Generate the file.
23:     if rank > 0 then                         ▷ Workers
24:         MPI-Recv(offset);                    ▷ Receive the offset position to calculate the matrix
25:         CalculateMatrix(offset);             ▷ Compute the matrix
26:         MPI-Send(color);                     ▷ Send the computed matrix to the master.
```

Once the algorithm was finished, we made different tests to prove that it was really working. In order to compile it and run it locally, we have used the *open-mpi* library by running the following commands:

- *Compile:* mpicc -o <file> <file.c>
- *Execute:* mpirun <file>

(b):

The result is plotted using the same colormap() as shown in the lectures:

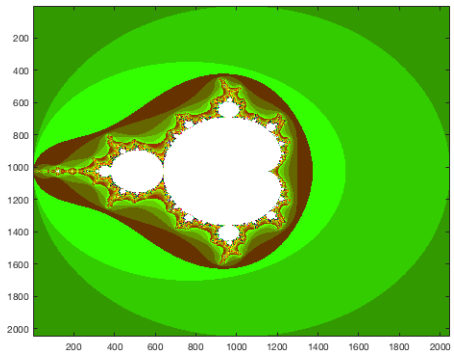

Figure 2: Parameters: w = 2048, h = 2048, N = 256, b = 2. Colormap(colorcube(255).

(c):

To magnify the interesting parts of the computed figures we decided to modify with b parameter. Concretely we needed to create a zoom window and for this reason we created four additional parameters to control: factor-width, factor-height,b-width and b-height. Using this parameters and updating them through different values you can achieve the desired zoom.

The equations that we change in our code are the ones attached below:

$$dx = 2 * \mathbf{WidthFactor}/(width - 1); \tag{8}$$

$$dy = 2 * \mathbf{HeightFactor}/(height - 1); \tag{9}$$

$$dreal = (x + offset) * \mathbf{dx} - \mathbf{BWidth}; \tag{10}$$

$$dimag = (y + yoffset) * \mathbf{dy} - \mathbf{BHeight}; \tag{11}$$

Here we attach some of the obtained results:

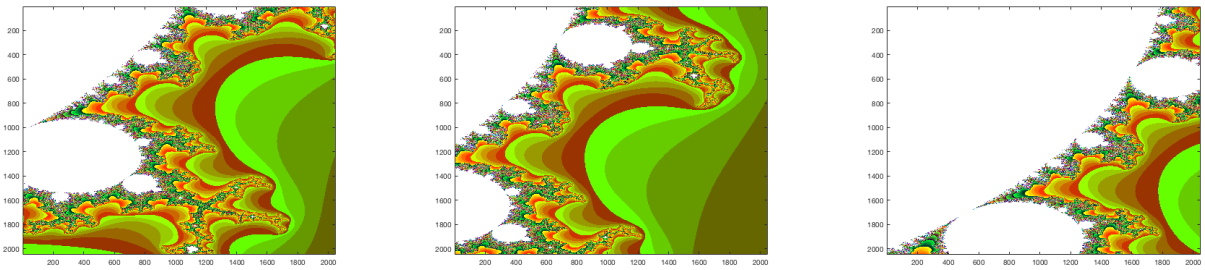

Figure 3: Zoom achieved with parameters: WidthFactor = [0.15], HeightFactor = [0.10], BWidth = [-0.20,-0.25,-0.15] and HWidth = [-0.25,-0.30,-0.25]

Appendices

A C code Question 6

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <mpi.h>

MPI_Status status;

// ---------------------------------
// MANDELBROT ALGORITHM PARAMETERS
//
// width  = Width of area to calculate (pixels)
// height = Height of area to calculate (pixels)
// N      = Number of different colors
// b      = Bound value for Mandelbrot set. Modyfying this value acts as a zoom.
//
#define width  2048
#define height 2048
double N = 256;
double b = 1;
// ---------------------------------

int color[height][width] = {0};
int final_color[height][width] = {0};
int num_workers = 0;
int slice_width = 0;
FILE *fp;

// Computations cal_pixel
int cal_pixel(double complex d, double b, double N){
    int count = 1;
    double complex z = 0;
    while ((cabs(z)<b) && (count < N)){
        z = (z*z)+ d;
        count= count + 1;
    }
    return count;
}

// Create color.txt for plotting of result
void CreateFile(){
    fp = fopen("color.txt", "w");
    for (int i = 0; i<height; i++){
        for (int j=0; j<width; j++){
            fprintf(fp,"%d", final_color[i][j]);
            fprintf(fp," ");
        }
        fprintf(fp,"\n");
    }
    fclose(fp);
}

// Add color array to final array
void ArrayAssignment(int color[height][width]){
    for (int i = 0; i < height;i++){
        for (int j = 0; j < width; j++){
            final_color[i][j] += color[i][j];
        }
```

```
58      }
59  }
60
61  // Mandelbrot calculations
62  void CalculateColumn(int offset){
63
64      double dx = 2*b/(width-1);
65      double dy = 2*b/(height-1);
66      double dreal = 0;
67      double dimag = 0;
68      double complex d = 0;
69      int yoff = 0;
70
71      for (int x = 0; x < slice_width; x++){
72          dreal = (x+offset)*dx - b;
73          for (int y = 0; y < height; y++){
74              dimag = (y+yoff)*dy-b;
75              d = dreal + 1I*dimag;
76              color[(x+offset)][y] = cal_pixel(d,b,N);
77          }
78      }
79  }
80
81  // Main function
82  int main (int argc, char **argv) {
83
84      int offset = 0;
85      int size = 0;
86      int rank = 0;
87      int rc = 0;
88      int id, tasks, tag, i;
89      int source;
90
91      // Initiate MPI environment, obtain number of workers and rank.
92      rc = MPI_Init(NULL,NULL);
93      rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
94      rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
95
96      // Number of workers is the total minus one (the master). Each worker will have a ...
            column to calculate.
97      num_workers = size-1;
98      slice_width = width/num_workers;
99
100     // Master
101     if (rank == 0){
102         // Send offset in y to each worker.
103         for(int dest = 1; dest ≤ num_workers; dest++){
104             MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
105             offset = offset + slice_width;
106         }
107
108         // Recieve all results from workers and join arrays for final result.
109         for(int i = 1; i ≤ num_workers; i++){
110             source = i;
111             MPI_Recv(&color,height*width,MPI_INT,source,1,MPI_COMM_WORLD,&status);
112             ArrayAssignment(color);
113         }
114
115         // Create final file with results.
116         CreateFile();
117
118     // Workers
119     }else{
120         source = 0;
```

```
121         // Recieve offset in y from which to start calculating, and send result array ...
                back.
122         MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
123         CalculateColumn(offset);
124         MPI_Send(&color, height*width, MPI_INT, 0, 1, MPI_COMM_WORLD);
125
126     }
127     MPI_Finalize();
128 }
```

B Matlab code Question 6

```
1 fid = fopen('color.txt','r');
2 data = textscan(fid,'%d');
3 fclose(fid);
4 data_matrix = cell2mat(data);
5 data_2 = reshape(data_matrix,6000,6000);
6 image(data_2);
7 colormap(colorcube(85))
8 %colormap(jet(40))
```

References

[1] "Wikipedia: Process in computing," https://en.wikipedia.org/wiki/Process_(computing), last accessed: 2018-02-8.

[2] "Wikipedia: Processor in computing," https://en.wikipedia.org/wiki/Processor_(computing), last accessed: 2018-02-8.

[3] "Super-linear speed-up example," https://stackoverflow.com, last accessed: 2018-02-8.

[4] "Wikipedia: Mandelbrot set," https://en.wikipedia.org/wiki/Mandelbrot_set, last accessed: 2018-02-8.