Assignment 2: Mandelbrot in MPI

In this assignment you will learn how to use MPI. You will again parallelise the computation of the Mandelbrot image, but this time in a distributed-memory setting.

This assignment should be done by teams of 2-3 students. How you distribute the work within the team is up to you. However, you need to declare who did which part. This assignment is graded.

The repos have been updated with information about MPI, please do a git pull and read up. Based on FAQ they may be updated in the future, so it is helpful to pull regularly. Note that `src/seq/mandelbrot.c` now outputs GFLOPS/s instead of seconds.

Please include your code (does not count towards page limit) in the pdf as well, and upload the pdf non-zipped.

1 Assignment

Your tasks are to

Task 0: specs

As in the OpenMP assignment, clearly describe your experiments. We will be using the `csmpi_short` partition which has different hardware from the previous assignment.

- specify exactly what hardware is being used (at least CPU version, clock frequency, memory)
- specify exactly what software is being used (at least compiler version, compiler flags)
- specify exactly which parameters (max number of iterations, part of the image you compute, number of pixels) you are using.
- repeat each experiment at least 10 times and report average time as well as the variability (error bars). Specify exactly how you derived the numbers you obtained.
- If you use graphs, make sure that you annotate the axes appropriately, that you state explicitly what the graphs show, and make sure that you use useful scales. For example, a scale that has numbers ranging for `0,0000000001` to `0,00000001` is a **bad** choice!

Task 1: Programming

Your program should have the following functionality.

1. Compute the Mandelbrot set in parallel
2. Assemble the computed image to node 0, and output it to `stderr`.
3. Compute the percentage of the computed [x1, x2] x [y1, y2] subset that is in the Mandelbrot set. That is, the percentage of values in `picture` equal to `max_iter`. Output this once to `stderr`.

It is impossible to implement 2 in a scalable way, so this will only be used on small examples in order to inspect the generated image, giving you an idea whether you have implemented 1 correctly. For benchmarking, we are only interested in 3. You can have both programs in a single source with preprocessor directives. For example, take the following example

```
#include <stdio.h>

int main(int argc, char **argv)
{
#ifdef CHECK
    printf("Check version\n");
#else
    printf("Benchmark version\n");
#endif

    return 0;
}
```

If compiled with `gcc <name>.c -o name`, this will print `Benchmark version`, but if compiled with `gcc -DCHECK <name>.c -o name` it will print `Check version`.

Use this to make sure that your program does task 1 and 3 if compiled without defining `CHECK` and task 1 and 2 if compiled with macro `CHECK`.

Task 2: load balancing

The load balance is the most flops done by any processor, divided by the least flops done by any processor. You want this as close to 1 as possible. What does your implementation get? What does your implementation do to get good load balancing? Does your load-balancing strategy affect the memory scalability of your program?

Bonus: can you do dynamic scheduling like the OpenMP `#pragma omp parallel for schedule(dynamic)` in MPI?

Task 3: evaluation

Evaluate the performance of your MPI program. Do you have weak scaling? Strong scaling? Is your implementation memory-scalable? Support your claims with measurements. You should have a separate graphs for strong and weak scaling with the number of ranks on the x-axis and the number of GFLOPS/s on the y-axis.

Task 4: leaderboard

Submit your implementation to the leaderboard. See the `https://gitlab.science.ru.nl/parallel-computing2/leaderboard` repo.

Task 5: Team

Provide a short description on how you divided up the work, i.e., who did what? Attribute percentages to your overall contributions.

Page limit:

At most 4 pages excluding tables and figures.