TU WIEN Informatics

Assoc.Prof. Dr. Sascha Hunold
TU Wien
Faculty of Informatics
Research Group for Parallel Computing

Basics of Parallel Computing
2021S
Assignment 1

---

issue date: 2021-04-21
due date: 2021-05-05
No extensions.

WHAT TO HAND IN VIA TUWEL

1. A PDF file containing your answers (plots and discussion thereof) to Sections 2.2–2.4.
2. The source code of your implementation. If you have only modified `julia_par.py`, submitting this file will suffice. If you have created additional files, upload an archive (zip, tar) containing all your source files.

   Do not hand in RAR, Word, raw data files (e.g., `output_exp.dat`), etc.

---

1 The Problem / Context

In this assignment, we will examine the computation and visualization of Julia sets. For this exercise, we slightly modified the Python code to generate an image of a Julia set, which is given at scipython.com. An example image of a Julia set is shown in Figure 1.

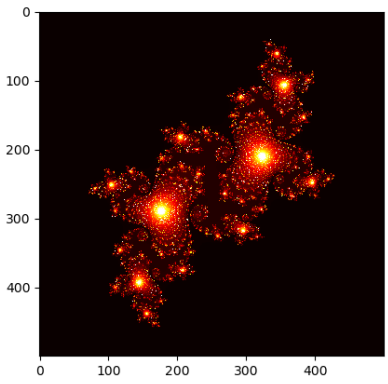

Figure 1: Example rendering of a Julia set.

In order to compute such an image, we first map each pixel of an image to a point in a complex plane and then apply an iterative function to each complex point. In a second step, the resulting complex points are color graded using some color map to produce the final image. Please see the description on the website for more details.

You are given a sequential Python program that computes and renders a Julia set. Your task is to parallelize the computation of the Julia set.
The provided program has the following parameters:

```
python julia_par.py -h
usage: julia_par.py [-h] [--size SIZE] [--xmin XMIN] [--xmax XMAX] [--ymin YMIN]
           [--ymax YMAX] [--patch PATCH] [--nprocs NPROCS] [-o O]

optional arguments:
  -h, --help       show this help message and exit
  --size SIZE      image size in pixels (square images)
  --xmin XMIN
  --xmax XMAX
  --ymin YMIN
  --ymax YMAX
  --patch PATCH    patch size in pixels (square images)
  --nprocs NPROCS  number of workers
  -o O             output file
```

- The parameter `size` denotes the size of the resulting image in pixels. We assume that the images are square, thus, the image has `size`$\times$`size` many pixels.
- In our implementation, we use `xmin` (`ymin`) and `xmax` (`ymax`) to denote the boundaries of our complex plane.
- The `patch` size specifies how large each patch (rectangular tile) is, which will be computed in the parallel computation (where `patch` $\leq$ `size`).
- The argument `nprocs` denotes the number of workers in the parallel execution.
- Last, we can save the image to a file which is passed using the `-o` flag.

Initially, the program only works sequentially and can be started like this

```
python julia_par.py -o test.png
500;20;1;9.512173211000118
```

2 The Tasks

2.1 Parallelize the Computation of the Julia Set (10 points)

Your first task is to parallelize the computation of the Julia set. We will use a task list and the Pool pattern to accomplish this task.

The provided, sequential Julia code does the following: For each pixel at position p_x, p_y of our image of `size`×`size` pixels, it computes the corresponding point in the complex plane, say z_p. This value z_p is then used to compute the final value iteratively. The original sequential code computes the result line by line and pixel by pixel. We are going to change that.

We select the following parallelization strategy: we decompose the problem of rendering the entire image into the problem of rendering subimages and putting these subimages back together to form the final resulting image. An example of this decomposition strategy is shown in Figure 2. Notice that patches are mostly square, except on the edges (right, bottom, bottom-right). Thus, the number of pixels in these corner patches may be smaller than `patch`×`patch`.

Your parallel implementation should follow the pseudo-code given in Listing 1. Your parallel implementation should work with any number of workers `--nprocs` ≥ 1.

Next, you will test the performance of your parallel Julia set implementation on one compute node of `hydra`. Make sure that you do not run the experiments on the front-end (login) node, which only has 16 cores. In contrast, the compute nodes of `hydra`, named `hydra01`–`hydra36`, comprise 32 cores. To run your code on one of the compute nodes, you need to execute either `sbatch` or `srun`.

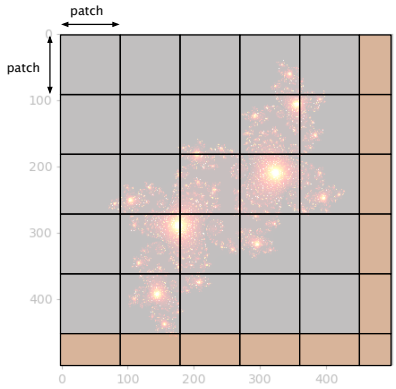

Figure 2: Decomposition of image into patches (tiles).

Listing 1: Parallel pseudo code

```
# assuming size (mod patch) = 0
for x in 0 to size (step patch):
  for y in 0 to size (step patch):
     task_list.append( (x, y, patch, meta_information) )
# meta information may contain offsets, original image size,
# original boundaries of complex plane, etc.

create Pool with nprocs workers

# make sure that each task in the task_list is handled alone
# in multiprocessing.Pool.map, we need to specify chunksize=1
completed_patches = Pool.map(compute_patch, task_list, 1)

for p in completed_patches:
   copy subimage of p to correct final position
```

2.2 Compute Speed-up and Parallel Efficiency for 2 Instance Sizes (11 points)

In the first set of experiments, we will analyze the scalability of your parallel implementation for two different problem sizes, i.e., `size` $\in \{200, 1000\}$. For each of the problem sizes, measure the running time for the following number of processors (cores): `nprocs` $\in \{1, 2, 4, 8, 16, 24, 32\}$. We keep the patch size fixed, i.e., `patch` $\in \{20\}$. Repeat each experiment three times and report the mean running time.

Now, compute the relative speed-up (with respect to the running time with 1 processor) and the parallel efficiency for each problem size.

Provide a table with the following data (note that the timings are artificial, speed-up and par. efficiencies have to be filled):

size	p	mean runtime (s)	speed-up	par. eff.
200	1	4.3	fill	fill
200	2	4.2	..	..
	⋮			
200	32	5.1		
1000	1	6.3		
1000	2	6.2		
	⋮			
1000	32	6.1		

Now, plot the data. You can choose the graph type that you think is best for plotting these data (e.g., bar charts, line charts, point charts, etc.).

Provide three different plots:

1. One plot compares the absolute running time for both problem sizes and varying numbers of cores.
   - x: number of cores
   - y: running time (s)
   - two groups (size=200, size=1000)
   - Discuss the findings (2–3 sentences)
2. One plot compares the relative speed-up for both problem sizes and varying numbers of cores.
   - x: number of cores
   - y: relative speed-up
   - two groups (size=200, size=1000)
   - Discuss the findings (2–3 sentences)
3. One plot compares the parallel efficiency for both problem sizes and varying numbers of cores.
   - x: number of cores
   - y: parallel efficiency
   - two groups (size=200, size=1000)
   - Discuss the findings (2–3 sentences)

2.3 Influence of Patch Size (4 points)

Next, we will analyze the influence of the patch size. To do so, we keep the number of cores (`nprocs` $\in \{32\}$) and the problem size (`size` $\in \{1000\}$) fixed. You then measure the running time of your parallel implementation for the following patch sizes: `patch` $\in \{1, 10, 30, 200, 500\}$. Again, measure the running time for each patch size three times and report the mean runtime in a table as follows:

size	p	patch	mean runtime (s)
1000	32	1	fill
1000	32	10	fill
	⋮		
1000	32	500	fill

Now, plot your data into one plot!

- x: patch size
- y: mean runtime (s)
- Discuss your findings (2–3 sentences)

2.4 Finding the Best Patch Size (4 points)

Last, we keep the problem size and the number of cores fixed, this time with `size` $\in \{800\}$ and `nprocs` $\in \{16\}$. Now, we incrementally increase the patch size and measure the mean running time of three runs each (`patch` $\in \{1, 2, \ldots 30\}$).

Provide your measurement results in a table as follows:

size	p	patch	mean runtime (s)
800	16	1	fill
800	16	2	fill
	⋮		
800	16	30	fill

Now, plot your experimental data!

- x: patch size
- y: mean runtime (s)
- Discuss your findings (2–3 sentences)

3 Appendix

SBATCH Files

We also provide a SBATCH file (`run_experiment.job`) that you can use to run your experiments. Please modify the variables in the script according to your needs and the actual experiment. You can then simply submit the sbatch file to SLURM:

```
sbatch run_experiment.job
```

This assumes that `julia_par.py` resides in the same directory as `run_experiment.job`. If not, you need to update the paths inside `run_experiment.job` accordingly.

Optional: IDEs / PyCharm

You may consider an IDE, such as PyCharm, to support you in completing this assignment. In this case, you will need to install an IDE (e.g., PyCharm) and a Python interpreter on your local machine.

Optional: Docker

You can also use Docker to build a container that provides a Python interpreter. A `Dockerfile` is part of the archive that should install all required packages. We provide this `Dockerfile` without any warranty that it works in your environment, but it may be helpful to some.

You can run the following command to build the container (assuming you are in the directory where the `Dockerfile` resides):

```
docker build  -t bopc2021 .
```

When the build has been completed, you may start the container

- in Jupyter mode

  ```
  docker run -it --rm -p 8888:8888 -v "$PWD":/home/jovyan/work bopc2021
  ```

- or in shell mode (good for Python scripts)

  ```
  docker run -it --rm -p 8888:8888 -v "$PWD":/home/jovyan/work bopc2021 bash
  ```

(`$PWD` is your current working directory. You may need to change this variable depending on your operating system.)