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

Basics of Parallel Computing
2023S
Assignment 1

---

issue date: 2023-04-20
group registration: 2023-04-27
due date: 2023-05-04
No extensions.

Please do not forget to register with a group on TUWEL, also if you want to work alone! Otherwise, you will not be able to submit your answers.

What to Hand in via TUWEL?

1. A PDF file containing your answers (plots and discussion thereof) to Sections 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, etc.

Rules

- Ensure that the following information is provided in the PDF document:
  - names of authors
  - date
  - course and assignment name
  - TUWEL group size and number
    - These are the values for `GROUP_SIZE` and `GROUP_NUMBER` in the Python script.
    - Example: If you are registered for “2 Person Group 12” on TUWEL, then your group size is 2 and your group number is 12. If you are working alone in group “1 Person Group 20”, your group size is 1 and your group number is 20.
- Use a spell checker. Check your grammar.
- Label your figures and tables appropriately, e.g., “Table 1: Summary” is not an appropriate description.
- Reference each figure and table in the text. Ensure that each figure/table is indeed described in the text. Only stating that “Table 1 contains a summary.” is not considered appropriate. Please explain what exactly is summarized in a table.
- Provide your raw benchmark data in the appendix of your submission report. When using LaTeX, you may want to have a look at the csvsimple package to generate tables from your csv files.

---

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.

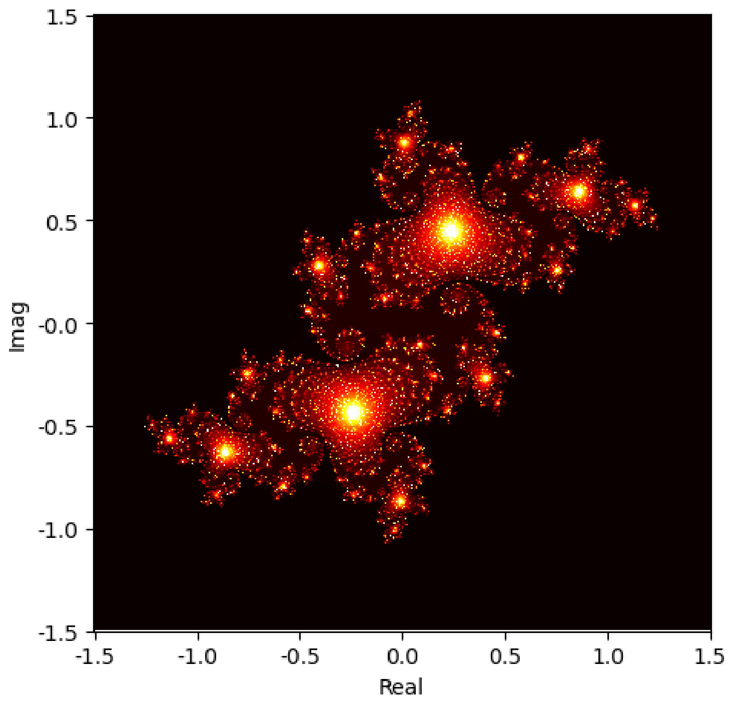

Figure 1: Example rendering of a Julia set ($c = -0.1 + 0.65i$, $I_{\text{max}} = 2000$).

In order to compute such an image, we first map each pixel of an image to a point z_p in the complex plane. We then apply the non-linear difference equation $z_{n+1} = (z_n)^2 + c$ with $z_0 = z_p$ for some $c \in \mathbb{C}$ and count the number of iterations I until either $|z_I| > \theta$ (for a given threshold $\theta \in \mathbb{R}$), or $I = I_{\max}$ ($I_{\max} \in \mathbb{N}$). In a second step, the pixel corresponding to z_p is color graded using some color map according to the value of I 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:

```
python3 julia_par.py -h
usage: julia_par.py [-h] [--size SIZE] [--xmin XMIN] [--xmax XMAX] [--ymin YMIN] [--ymax YMAX]
                    [--patch PATCH] [--nprocs NPROCS] [--draw-axes] [-o O] [--benchmark]


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
  --draw-axes      Whether to draw axes
  -o O             output file
  --benchmark      Whether to execute the script with the benchmark Julia set
```

- The parameter `size` denotes the size of the resulting image in pixels. We assume that the images are square, thus, the image has `size`×`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` ≤ `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.

Make sure to set the variables `GROUP_SIZE` and `GROUP_NUMBER` in the `julia_par.py` script to your values from TUWEL. Otherwise you will not be able to execute the script. Please also provide the values of `GROUP_SIZE` and `GROUP_NUMBER` you used in your submission report.

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

```
python3 julia_par.py -o test.png
```

This then produces output in the following form, as well as an image file:

```
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.

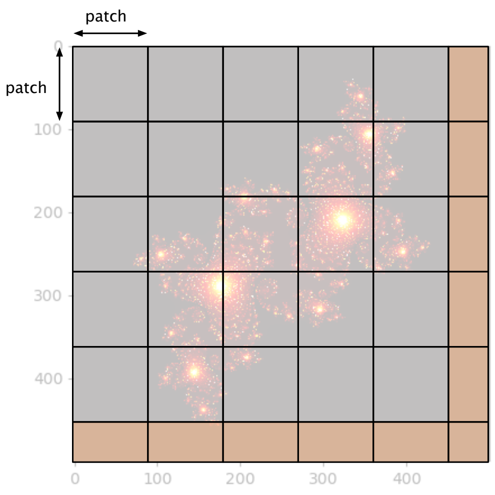

Figure 2: Decomposition of image into patches (tiles).

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` $\geq$ 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`.[1]

[1]See Appendix A as well as `http://doku.par.tuwien.ac.at/docs/slurm` (accessible within the TU VPN)

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 (14 points)

You are provided two parameters c_s and c_b used in the iterative Julia set computation which result in two different Julia sets (we will call them *student* set and *benchmark* set, respectively). Make sure to set the variables `GROUP_SIZE` and `GROUP_NUMBER` in the `julia_par.py` script according to the size and number of the group you registered for in TUWEL for retrieving your respective c_s. The parameter c_b is used when the `--benchmark` flag is provided. Otherwise, c_s is used.

In the first set of experiments, we will analyze the scalability of your parallel implementation for two different problem sizes, i.e., `size` $\in \{160, 1050\}$. 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` $\in \{25\}$. Repeat each experiment three times using c_b 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 parallel efficiencies have to be filled):

size	p	mean runtime (s)	speed-up	par. eff.
160	1	4.3	fill	fill
	⋮		..	..
160	32	5.1		
1050	1	6.3		
	⋮			
1050	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=160, size=1050)
   - 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=160, size=1050)
   - 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=160, size=1050)
   - Discuss the findings (2–3 sentences)

Finally, *repeat the whole experiment* and *provide the same table and plots* for your respective c_s and compare and contrast your individual results to the results obtained using c_b. Try to familiarize yourself with the problem and explain the differences between the two workloads and how these influence the benchmarks of your parallel computation.

Thus, your submission for this exercise should consist of the following:

1. A table with mean runtime, speed-up and parallel efficiencies for $c = c_b$.
2. Three plots for $c = c_b$ with two groups each (size=160, size=1050) and short discussion:
   a) Absolute running time
   b) Relative speed-up
   c) Parallel Efficiency
3. A table with mean runtime, speed-up and parallel efficiencies for $c = c_s$.
4. Three plots for $c = c_s$ with two groups each (size=160, size=1050) and short discussion:
   a) Absolute running time
   b) Relative speed-up
   c) Parallel Efficiency
5. Discussion comparing the results obtained with c_b and c_s.

Also, make sure to include the raw benchmark data for c_b *and* c_s in your submission report as described in the Rules.

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` $\in \{750\}$) fixed. You then measure the running time of your parallel implementation for the following patch sizes: `patch` $\in \{1, 5, 10, 20, 55, 150, 400\}$. In this and the following exercise, only use the c_s parameter, i.e., execute the script *without* the `--benchmark` argument. Measure the running time for each patch size three times, provide your raw results in the appendix of your submission and report the mean runtime in a table as follows:

size	p	patch	mean runtime (s)
750	32	1	fill
750	32	5	fill
	⋮		
750	32	400	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` $\in \{700\}$ and `nprocs` $\in \{16\}$. Using c_s, 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)
700	16	1	fill
700	16	2	fill
	⋮		
700	16	30	fill

Now, plot your experimental data!

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

Again, also make sure to include your raw benchmark data in the appendix of your submission report.

3. Speed-up Analysis (3 Points)

Note: For all calculations in $\mathcal{O}$-notation in Exercises 3 and 4, you can assume that constants have been normalized to one (e.g., $\mathcal{O}(n^3) = n^3$) and that log is the *natural logarithm* (i.e., $\log e = 1$).

Consider two parallel sorting algorithms:

- Algorithm $\mathcal{A}_1$, running in $T_{par}^{\mathcal{A}_1}(n,p) = \mathcal{O}(\frac{n \log n}{p} + \log n)$ time steps
- Algorithm $\mathcal{A}_2$, running in $T_{par}^{\mathcal{A}_2}(n,p) = \mathcal{O}(\frac{n \log n}{p} + n)$ time steps.

The best sequential algorithm for this problem is known to run in $T_{seq}^{*}(n) = \mathcal{O}(n \log n)$ time steps.

1. For both algorithms, calculate the *absolute speed-up* $S_a^{\mathcal{A}_i}(n,p)$ for $p \in \{4, 16, 64\}$, $n = 1000$.
2. For both algorithms, calculate the *parallel efficiency* $E^{\mathcal{A}_i}(n,p)$ for $p \in \{4, 16, 64\}$, $n = 1000$.
3. Calculate the potential speed-up $S^{\mathcal{A}_i}(n)$ for both algorithms in terms of n.

4. Weak Scaling Analysis (2 Points)

Consider a *work-optimal* parallel algorithm that runs in $T_{par}(n,p) = \mathcal{O}(n^4/p)$ time steps. You are asked to perform a weak-scaling analysis of this algorithm. In this analysis, the average work per processor must stay fixed at w operations, such that a given, fixed-budget running time $\mathcal{O}(w)$ is guaranteed.

1. How must n increase as a function of w and p in order to keep the average work at $\mathcal{O}(w)$?
2. We now fix the input size $n = 100$ when running the parallel algorithm with $p = 1$ processor. Compute the required input sizes n for $p \in \{2, 4, 8, 16, 128\}$.

A. 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.

Testing Your Code

You can easily check whether your code works on your own personal machine. You could simply run in a terminal

```
$ python3 julia_par.py --size 800 --nprocs 4
800;20;4;2.9226629859767854
```

However, if you want to test whether your code works with a certain number of cores, e.g., 24 cores, you can use `hydra` and run

```
$ hunold@hydra:~/julia_set$ srun -p q_student -t 1 -N 1 -c 32 python3 julia_par.
    py --size 600 --nprocs 24
600;20;24;0.46532210102304816
```

Notice that the entire call is in one line (it was just too long for printing it on this page). Running the Python program with `srun -p q_student -t 1 -N 1 -c 32` ensures that you

- run through the students' job queue (`-p q_student`),
- request to run for a maximum duration of 1 minute (`-t 1`),
- request 1 compute node (`-N 1`), and
- request to reserve all 32 cores on this compute node (`-c 32`), although you may actively use less than 32 cores. Keep this value at `-c 32`!

Thus, if you want to test whether the code scales on `hydra`, we recommend to prepend

```
srun -p q_student -t 1 -N 1 -c 32
```

always. The only thing that you may adapt is the `-t` parameter, which can be used to increase the requested time up to `5` minutes, as five minutes is the maximum time that each job is allowed to run on the system before being killed.

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 will install all required packages. We provide this `Dockerfile` without any warranty that it works in your environment, but it may be helpful to some.

Please see the file `README.docker` for more details.