Assignment for Scientific Computing Using Python - High Performance Computing In Python

Weifan Zhang

1 Implementation

I implemented 5 versions:

- naive.
- vectorize.
- guvectorize.
- multi-process.
- multi-thread.

These implementations are validated for correctness by the unit tests introduced in Section 4.

2 Output

- File “Mandelbrot_Set_Colormap.pdf” is the colormap of Mandelbrot Set shown in Figure 1
- File “real.csv” is the real part of the complex numbers in the simulation. (all 5 versions are the same)
- File “images.csv” is the image part of the complex numbers in the simulation. (all 5 versions are the same)
- File “results.csv” is the result data of the Mandelbrot Set. (all 5 versions are the same)
- File “benchmark_laptop.csv” is the result data of the benchmark on my 4-core-8-thread laptop.
- File “benchmark_laptop.pdf” is the bar chart of the benchmark results on my 4-core-8-thread laptop, which can show the execution time for different versions and different numbers of parallel workers.
- File “benchmark_vm.csv” is the result data of the benchmark on an 8-core-8-thread Virtual Machine (VM).
- File “benchmark_vm.pdf” is the bar chart of the benchmark results on an 8-core-8-thread VM, which can show the execution time for different versions and different numbers of parallel workers.

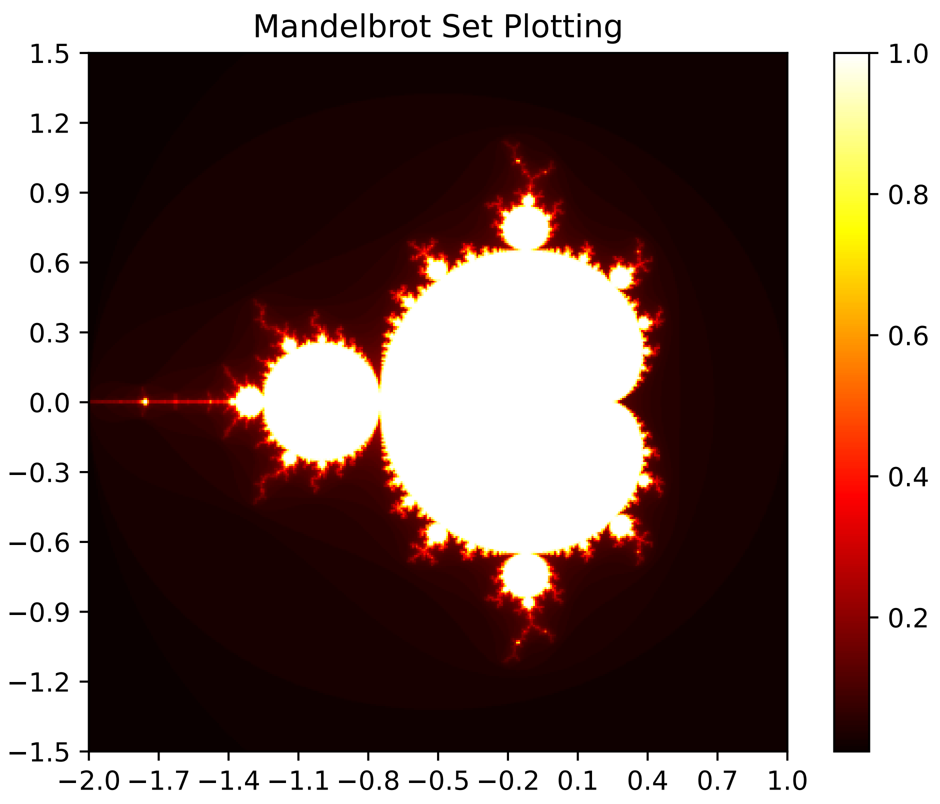

Figure 1: Mandelbrot Set colormap.

3 Software Design

The folder “mandelbrot-set-py” is the code of this project. I set the Mandelbrot Set parameters according to the project description, and according to the computational resources I have available, this project has 1000 different real parts and 1000 different imaginary parts, which form 1000×1000 different complex numbers “c”.

3.1 naive

This version is in the file “naive.py”. We can run the function “calcResults()” to get the result and this function does not have input parameters.

It uses a common algorithm for Mandelbrot Set which we can easily find on the Internet. We can write the code by combining the algorithm with the project description. It is not related to the skills or knowledge in this Ph.D. course, so I do not put it here.

3.2 vectorize

This version is in the file “use_vectorize.py”. We can run the function “calcResults()” to get the result and this function does not have input parameters.

I make this version by changing the naive version. I use "@vectorize" to decorate the function "M()" to let Python do the same operation to all elements in the lists in parallel. To avoid the warning, I also use "@vectorize" to decorate the functions called in "M()". "@vectorize" does not support 2 return values, so I divide a function into 2 functions "mandelbrotRealFunc" and "mandelbrotImageFunc".

3.3 guvectorize

This version is in the file "use_guvectorize.py". We can run the function "calcResults()" to get the result and this function does not have input parameters.

I make this version by changing the naive version. I add a function "doMForAll" decorated by "@guvectorize" to apply "M()" to all elements in the lists in parallel. To avoid the warning, I use "@njit" to decorate the functions called in "doMForAll".

3.4 multi-process and multi-thread

These 2 versions are in the file "use_multiprocessing.py". We can run the function "calcResults()" to get the result. This function has 3 input parameters. The parameter "maxWorkers" is the maximum number of processes or threads. The boolean parameter "multiThreads" means whether we use the multi-process version or multi-thread version. The parameter "chunkPerWorker", together with "maxWorkers", controls the number of chunks that we split the tasks into.

I make these 2 versions by changing the naive version. I split the tasks into "$maxWorkers \times chunkPerWorker$" chunks and do them in parallel. This is because the program can only do up to "maxWorkers" works in parallel, so splitting the tasks into too many chunks is not helpful but will increase the consumption of CPU/task/process switching as well as the memory usage. However, if we split the tasks into too few chunks, there are 2 conditions:

- If the number of chunks is smaller than "maxWorkers", the multi-process ability will not be fully used, which is a waste.
- If the number of chunks is equal or only a little larger than "maxWorkers", there will also be a waste of multi-process ability. As the number of loops in the Mandelbrot Set algorithm will be different for different input complex numbers "c", the workloads of all tasks are not the same, so some processes will finish their tasks earlier than others and then they will not do anything but just wait for other processes finishing working.

Therefore, we should not set the chunk size too big or too small. In this project, I set "chunkPerWorker" by (1). Although I provide the parameter "chunkPerWorker", I fix it as 3 in my tests and benchmarking, because in my tests, this value best takes advantage of the multi-process ability, does not cause big CPU/task/process switching consumption, and does not cause Out of Memory (OOM) errors.

$$chunkPerWorker = \begin{cases} chunkPerWorkerGiverByUser, & \text{if } chunkPerWorkerGiverByUser \geq 1 \\ 1, & \text{if } chunkPerWorkerGiverByUser < 1 \end{cases} \tag{1}$$

4 Test Plan

The unit test in the file "unit_tests.py" can validate the correctness of this project. It validates that:

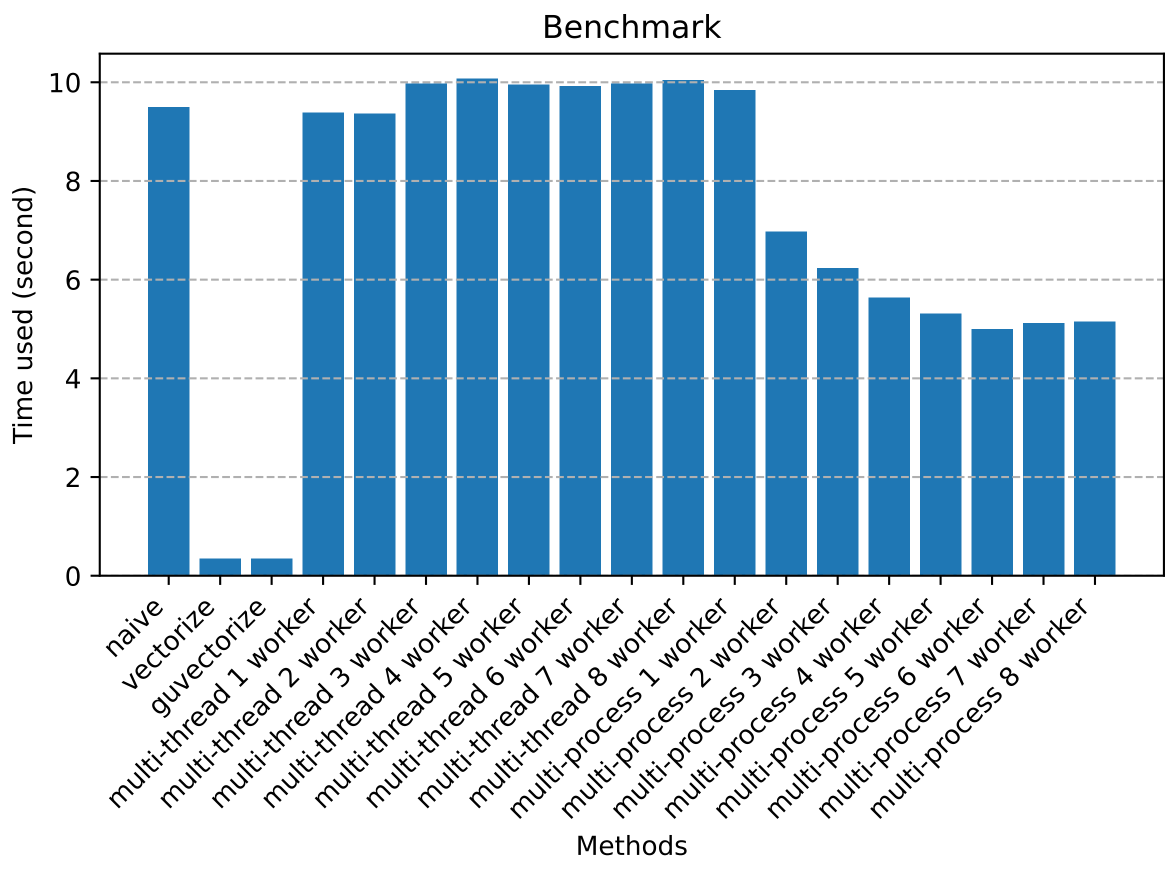

Figure 2: Benchmark results on my 4-core-8-thread laptop.

- The “naive” version is correct.
- All versions have the same “reals”, “images”, and “results”.

Because of the above 2 points, all versions are correct.

5 Profiling and Benchmarking

I benchmark the 5 versions and different numbers of parallel workers.

5.1 benchmarking on my laptop

Firstly, I run the benchmark on the WSL (Windows Subsystem for Linux) of my laptop. My laptop has the CPU “11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz”, which has 4 cores and 2 threads per core, and the CPU clock speed is 2800MHz. The memory is 24GiB and 7GiB of swap.

As shown in Figure 2, we can find that “vectorize” and “guvectorize” have similar performance and perform much better than others, which means that Numba is very good. “Multi-thread” does not reduce the execution time when compared with “naive”, no matter how many workers it has, so I think we can consider that “multi-thread” cannot run multiple tasks in parallel.

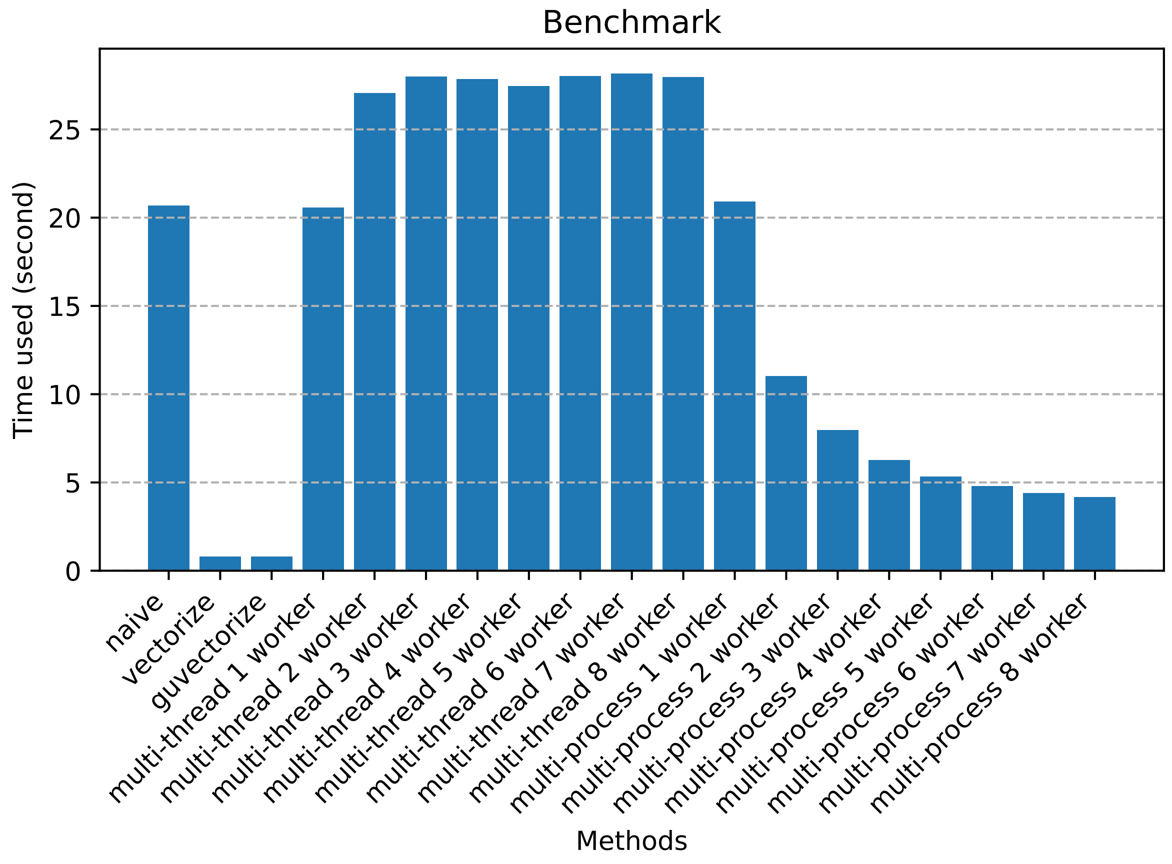

Figure 3: Benchmark results on an 8-core-8-thread VM.

In "multi-process", when workers are less than 6, the execution time decreases when workers become more, but the performance is kind of worse with 7 and 8 workers. This is different from what I thought because my laptop only has 4 cores, which should only support up to 4 workers working in parallel, which means more than 4 workers should not have effects, but actually 6 workers work better than 4. Maybe this can show that the workers more than CPU cores are also useful. However, for 7 and 8 workers, the chunks of tasks will be too much, but the extra workers may not be very useful, so more cost of CPU/task/process switching and memory usage will increase the execution time.

5.2 benchmarking on a VM

To explore more, I run the benchmark on a VM in a physical server. The physical server has the CPU "Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz", which has 28 cores and 2 threads per core, and the CPU clock speed is 1200-2400MHz. The VM has 8 cores and 1 thread per core. The VM has 32GiB memory with no swap.

The results are shown in Figure 3. The first thing that we notice is that the "naive", "vectorize", and "guvectorize" versions on the VM use about the double of the time on my laptop. I guess that the reason may be that each core of the VM may be actually half a core of the physical server, because the server has 28 cores but the server can have VMs with 56 cores in total, which may mean that a VM core only has the half of the CPU clock speed of a server core.

We can also find that in "multi-thread" items, 1 worker uses less time than more workers, which is not shown on my laptop. I think this is because VMs have some cost of switching between threads while

Physical Machine (PMs) do not. Another reason may be the CPU clock speed of my laptop is fixed to 2800MHz, but the VM has the CPU clock speed 1200-2400MHz and this range causes the difference in execution times.

In “multi-process” items, we can observe that, from 1 to 8 workers, every newly added worker can bring a reduction to the execution time, which means that the 8 CPU cores can support 8 running processes in parallel, even though they are in a VM rather than a PM. We also find that, the more workers the less time decrease, which I think is because more workers have a higher cost of CPU/task/process switching and memory usage.