CA3. High Performance Python

Christian Javier Can Montero
Data Engineering 7°B
Universidad Politécnica de Yucatán
Ucú, Yucatán. México
2009017@upy.edu.mx

I. Introduction

THE purpose of this project is to bridge the knowledge gap between high-performance computing (HPC) theory and real-world applications in a variety of contexts. Through participation in simulated data science projects or process improvement challenges, participants can gain real-world experience and develop their understanding of the practical applications of HPC.

Through a variety of HPC-related exercises, learners will be able to observe several techniques in action, including distributed systems, GPU acceleration, and parallel computing. Through the application of HPC methodologies to concrete and recognisable situations, people can gain a deeper understanding of HPC's capacity to solve complex problems and improve computing speed and efficiency.

The incorporation of real-world scenarios into these assignments enhances and increases the relevance of the learning process by assisting students in realising the application of HPC ideas in real-world circumstances. Through hands-on experience, participants will gain the knowledge and self-assurance needed to handle HPC-related tasks in their future careers.

II. Benchmarking and Profiling

Each point in a certain area of the complex plane is subjected to an iterative application of a complex quadratic polynomial, and when the sequence diverges, the point is coloured according to the number of iterations. This process creates the fractal.

Key components and functionality of the code include:

- Parameter Definition:

  The complex constants constant_real and constant_imag, as well as an area in the complex plane (x_min, x_max, y_min, y_max) that will be utilised to create the Julia set, are defined by the code.

- Julia Set Generation (generate_julia_set):

  The function generates a complex number grid that encompasses the designated area and repeatedly applies the Julia set formula ($z = z\hat{2} + c$) to every point. Before the size of the result above a threshold (signalling an escape to infinity), the number of iterations is recorded.

- Image Visualization (display_image):

  The script first computes the iteration counts, normalises these values, and then utilises them to create an image representation of the Julia set in grayscale (which is then copied into RGB layers). The colour of points that escape to infinity is different from that of those that stay limited.

- Execution and Benchmarking (create_fractal and %timeit):

  The function create_fractal coordinates the operations of production and display, providing information on execution metrics such as processing time and dimensions of the produced image. This function can be benchmarked by using the %timeit magic command, which can be used in an IPython or Jupyter environment. It runs the function many times and outputs the average execution time.

This code's visualisation of the particular Julia set is defined by the complex constant -0.62772 - 0.42193i. One of the hallmarks of the sensitivity and complexity inherent in fractals is the huge differences in fractals produced by different constants.

The visualisation output makes it possible to observe the complex boundary patterns typical of Julia sets, in which the boundary separating the bounded points from those that escape to infinity displays infinitely repeated, self-similar patterns.

```
Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    3.013    3.013    3.706    3.706 <ipython-input-15-39f8968e0069>:12(generate_julia_set)
     1    0.534    0.534    0.534    0.534 <ipython-input-15-39f8968e0069>:16(<listcomp>)
     1    0.146    0.146    0.146    0.146 {built-in method numpy.array}
     1    0.011    0.011    0.011    0.011 numeric.py:274(full)
     1    0.009    0.009    0.009    0.009 shape_base.py:372(stack)
    16    0.006    0.000    0.006    0.000 socket.py:543(send)
     1    0.003    0.003    3.726    3.726 <ipython-input-21-440d50228030>:1(<cell line: 8>)
     1    0.001    0.001    0.001    0.001 {built-in method numpy.zeros}
     1    0.001    0.001    0.001    0.001 {method 'reduce' of 'numpy.ufunc' objects}
     2    0.000    0.000    0.000    0.000 function_base.py:24(linspace)
     2    0.000    0.000    0.000    0.000 {built-in method builtins.compile}
    14    0.000    0.000    0.006    0.000 iostream.py:384(write)
    16    0.000    0.000    0.006    0.000 iostream.py:195(schedule)
     1    0.000    0.000    3.723    3.723 <ipython-input-15-39f8968e0069>:27(create_fractal)
    16    0.000    0.000    0.000    0.000 threading.py:1169(is_alive)
     5    0.000    0.000    0.007    0.001 {built-in method builtins.print}
     2    0.000    0.000    3.726    1.863 interactiveshell.py:3512(run_code)
     1    0.000    0.000    0.000    0.000 {built-in method numpy.empty}
    14    0.000    0.000    0.000    0.000 iostream.py:308(_is_master_process)
    16    0.000    0.000    0.000    0.000 threading.py:1102(_wait_for_tstate_lock)
    16    0.000    0.000    0.000    0.000 iostream.py:91(_event_pipe)
    14    0.000    0.000    0.000    0.000 iostream.py:321(_schedule_flush)
     2    0.000    0.000    0.000    0.000 interactiveshell.py:3337(_update_code_co_name)
     7    0.000    0.000    0.000    0.000 {built-in method numpy.asanyarray}
     2    0.000    0.000    0.000    0.000 {built-in method numpy.arange}
    16    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
    14    0.000    0.000    0.000    0.000 {built-in method posix.getpid}
     2    0.000    0.000    0.000    0.000 codeop.py:117(__call__)
     1    0.000    0.000    0.000    0.000 {built-in method numpy.asarray}
```

Fig. 1. Profiling output.

III. List and Tuples

The code of this exercise tests the performance of Python's list and deque data structures across a range of sizes by benchmarking important operations:

- List Operations:

  The benchmarks demonstrate that while adding to a list is quick (constant time), operations with linear time complexity, such as pop(0) and insert(0, 1), become slower as list size increases.

- List Search:

  Larger lists require more time to search within using list.index, demonstrating the linear time complexity of the method.

- Bisect Operations:

  On sorted lists, binary search using bisect_left is effective and shows a small increase in time with bigger list sizes.

- Deque Operations:

  Despite the size, deque operations like pop, popleft, append, and appendleft work consistently, demonstrating their effectiveness—especially for actions at the ends.

```
Size: 10000, Time: 0.00 (ms)
Size: 20000, Time: 0.00 (ms)
Size: 30000, Time: 0.00 (ms)
Size: 10000, Time: 0.00 (ms)
Size: 20000, Time: 0.00 (ms)
Size: 30000, Time: 0.00 (ms)
Size: 10000, Time: 0.00 (ms)
Size: 20000, Time: 0.00 (ms)
Size: 30000, Time: 0.00 (ms)
Size: 10000, Time: 0.00 (ms)
Size: 20000, Time: 0.00 (ms)
Size: 30000, Time: 0.00 (ms)
Size: 10000, Time: 0.00 (ms)
Size: 20000, Time: 0.00 (ms)
Size: 30000, Time: 0.00 (ms)
Size: 10000, Time: 0.00 (ms)
Size: 20000, Time: 0.00 (ms)
Size: 30000, Time: 0.00 (ms)
Size: 10000, Time: 0.00 (ms)
Size: 20000, Time: 0.00 (ms)
Size: 30000, Time: 0.00 (ms)
```

Fig. 2. Deque implementation output.

These findings, which highlight the cost of some list operations and the effectiveness of deque for end operations, illustrate the significance of selecting the right data structure based on the particular activities and performance requirements.

IV. Dictionaries and Sets

This activity compares the administrative zones of a municipality in Mérida in both 2010 and 2020 in terms of space. It indicates the new zones, the ones that vanished, and the ones that stayed the same.

It takes data for the municipality code '050', compares zone codes between the two years using GeoPandas for spatial data processing, and shows the results on a map with unaltered zones in green, new in blue, and vanished in red.

A map illustrating these modifications is the resultant image. The findings indicate changes in the region's administrative structure and urban development as some zones have stayed stable over the past 10 years, while others have vanished.

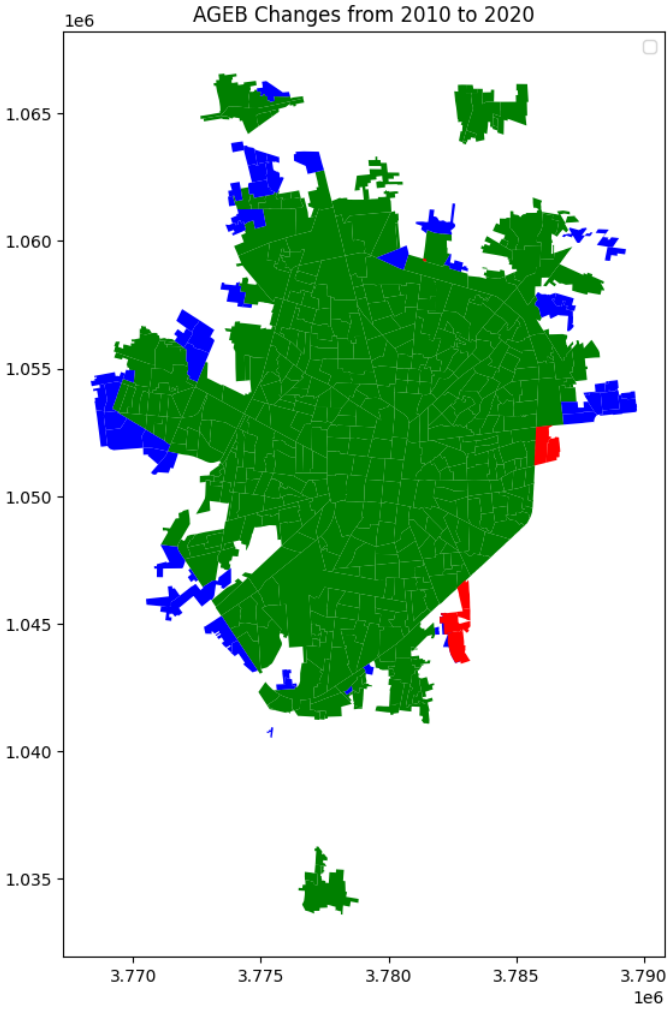

Fig. 3. Merida changes from 2010 to 2020.

V. Matrix and Vector Computations

It was tested with the goal of increasing performance by comparing the response times of standard Python with the optimization-focused module numexpr.

It was found that numexpr responds more quickly than standard Python, highlighting the optimisation capabilities of numexpr that are intended for larger tasks than are generally possible with simple Python scripts.

If the original code is modified to take advantage of numexpr's features, the processing becomes much more efficient and can be achieved by switching to its array-based techniques.

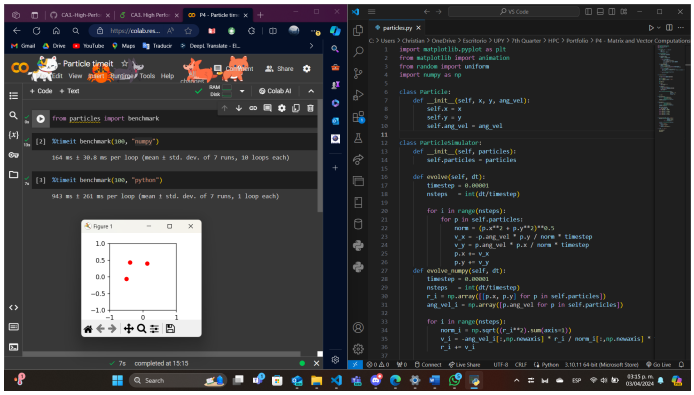

Fig. 4. Performance comparison using %timeit.

VI. Compiling to C

A new view on performance improvements is provided by exploring the incorporation of Cython into our computational framework. Through the use of Cython's features, we aimed to carefully evaluate its effect on faster execution as compared to standard Python scripts.

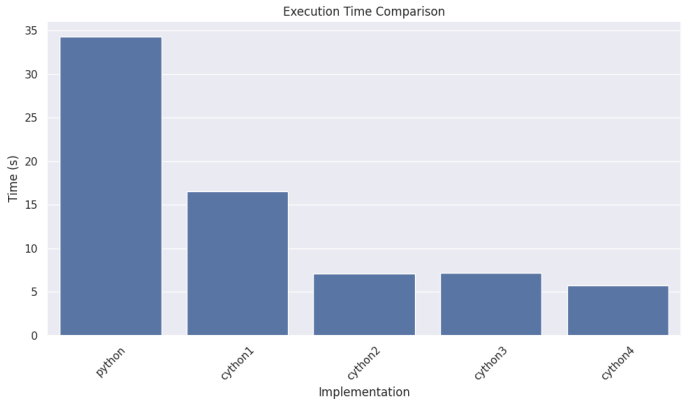

Fig. 5. Python and Cython performance comparison.

As shown in Fig. 5, each successive ”cython” implementation appears to be more optimized than the last, resulting in lower execution times for the task at hand. The ”python” implementation is the least optimized in terms of execution time.

- ”python” has the highest execution time, which is slightly above 30 seconds.
- ”cython1” shows a significant reduction in execution time compared to ”python”, dropping to slightly above 20 seconds.
- ”cython2” further reduces the execution time to around 15 seconds.
- ”cython3” has a similar execution time to ”cython2”, with a marginal decrease.
- ”cython4” shows the lowest execution time of all, just below 10 seconds.

This experimentation exercise shows that using Cython techniques greatly speeds up processing, highlighting its value in maximising the handling of large amounts of data.

APPENDIX A
GITHUB REPOSITORY

CA3. High Performance Python Repository