**Homework 8**

Juan Camilo Velasquez and Joaquin Rodriguez

**Section 1:** Line Profiler and Numba – Euler’s Method

*Pure Python - euler\_ode.py:* The traditional Python implementation was executed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Result** | **Analytical** | **Error** | **Execution Time** |
| Euler\_ode.py | 0.83907180 | 0.3907153 | 2.7264 | 11.7275 s |

*Line\_profiler file - euler\_ode\_profile.py:* The line-profiler was used on the pure python implementation to tests the lines that have the bigger impact over the total runtime of the code. The results are reported below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Timer unit:** 1e-06 s  **Total time:** 21.5381 s  **File:** euler\_ode\_profile.py  **Function:** euler\_integration at line 25 | | | | | |
| Line # | Hits | Time | Per Hit | % Time | Line Contents |
| 53 | 1 | 1.7 | 1.7 | 0.0 | nevals = int((tmax-t0)/dt) |
| 56 | 1 | 12.0 | 12.0 | 0.0 | y = np.zeros(nevals+1) |
| 57 | 1 | 4.9 | 4.9 | 0.0 | t = np.zeros(nevals+1) |
| 60 | 1 | 1.4 | 1.4 | 0.0 | y[0] = y0 |
| 61 | 1 | 0.3 | 0.3 | 0.0 | t[0] = t0 |
| 64 | 10000001 | 2231289.5 | 0.2 | 10.1 | for i in range(1, nevals + 1): |
| 65 | 10000000 | 3530092.1 | 0.4 | 16.1 | t[i] = t[i - 1] + dt |
| 66 | 10000000 | 16221835.6 | 1.6 | 73.8 | y[i] = y[i - 1]  + dt \* int\_funct(y[i - 1], t[i - 1]) |

**Conclusion:** In line 53, the code is computing the number of evaluations as an arithmetic operation and is non-significant to the execution time. From lines 55-61, these operations are also adding non-significant time. However, lines 64-66 are the main computation loops iterating the number of evaluations. Line 65 is updating the previous time array ‘t’ by the increment of ‘dt’. Finally, line 66 is the Euler’s method calculating the integration. These 3 lines are taking most of the CPU runtime.

*Numba Implementations - euler\_ode\_numba[1-2].py:* The decorators @jit were added to the int\_funct with the option nopython=True for the first version of the Numba file (\*numba1.py). Then, the decorator was added also to the euler\_integration function for the second version of the Numba file (\*numba2.py). The results of the execution are shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Result** | **Analytical** | **Error** | **Execution Time** |
| Euler\_ode\_numba1.py | 0.83907180 | 0.3907153 | 2.7264 | 8.4795 s |
| Euler\_ode\_numba2.py | 0.83907180 | 0.3907153 | 2.7264 | 0.379 s |

**Conclusions:**

* After adding Numba ‘@jit’ and ‘nopython=True’ to the int\_function, we improved the run time to 8.47 seconds compared to the pure python version of 11.72 seconds of the Euler’s integration. Both versions of y(t) values of provided different results, assuming a discrepancy due to the difference in implementation given by the Numba decorators. The dummy call verified the behavior of the code, providing insight for scenarios of improving this code. We can also understand better the overhead in the difference of time in the evaluation time.
* The second version of Numba decorators, we significantly improved the performance of the Euler’s method. The runtime was significantly reduced while maintaining accuracy. We added a dummy call to this version as well, however, the difference in time evaluation is not significant compared to the Numba1 version.

**Section 2:** Automatic Parallelization

*Pure Python - euler\_ode.py:* The traditional Python implementation was executed.

|  |  |  |
| --- | --- | --- |
| **File** | **Result** | **Execution Time** |
| **numInt.py** | 1.804776 | 83.705071s |
| **numInt.py** with @jit, nopython=True | 1.804776 | 1.703458s |
| **numInt.py** with @jit, nopython=True and prange | 1.804776 | 1.71281s |

**Conclusion:** The pure python version compared to the first optimization speedup increase of 49.13% applying the decorators @jit and nopython=True for compilation. It was noted that this version did not include parallelism. Additionally, we executed the code using decorators of @jit, parallel=True, and prange, which allowed parallel execution of the code and increased speedup of 65.32% when compared to the pure version runtime.

NUMBA\_NUM\_THREADS Environment Variable

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of Processes** | **CPU time[s]** | **Speedup** | **Efficiency** |
| **1** | 1.71281 | 1 | 1 |
| **2** | 1.137560 | 1.5056 | 0.7528 |
| **4** | 0.898708 | 1.9058 | 0.4764 |
| **8** | 0.633814 | 2.7023 | 0.3377 |
| **16** | 0.304047 | 5.6333 | 0.3520 |
| **20** | 0.264419 | 6.4776 | 0.3238 |

**Conclusion:** The parallelized implementation using different number of processes with the variable NUMBA\_NUM\_THREADS displayed interesting results. In Speedup, there is a significant increase using 2 threads, indicating effective parallelization. However, after 2 threads, the speedup gains become less significant, and the efficiency drops, leading the new overhead for finding resources. From previous course material, we have understood that efficiency starts to decrease as the number of threads increases.

While parallelization provided significant performance improvements, we recommend to balance the number of threads being used to avoid overhead and inefficiency in compilation.

**Section 3:** Cython – Matrix-Matrix Multiplication

|  |  |  |
| --- | --- | --- |
| **Matrix Sizes** | **Cython (s)** | **Numpy (s)** |
| **3 x 3** | 0.0000 | 0.0177 |
| **10 x 10** | 0.0000 | 0.0194 |
| **100 x 100** | 0.0030 | 0.0335 |
| **1000 x 1000** | 2.3346 | 0.1162 |
| **Average** | 0.5844 | 0.0467 |

Conclusion: From these results we can see that the implementation of Cython outperformed the Numpy in the smaller matrix sizes, however, Numpy had better performance in larger matrices than Cython.