HPC – HOMEWORK 8

Dai Nam Nguyen and Sara Restrepo Velasquez

For the first part of the homework, the **line_profiler** was applied to the pure Python implementation of the **euler_ode.py** script. The results of this evaluation are shown in Figure 1. The lines that contribute the most to the runtime, are those contained within the for loop. Line 73, which corresponds to calling the function **int_funct**, and evaluating it, is the one that takes up most of the time, with a percentage of 47.3% of the total time. Lines 74 and 75 take similar portions of the total time, with 14.1% and 11% respectively. These lines correspond to calculating the current values of **Y** and **t**.

```
72  10000001    1977016.3      0.2      8.2      for i in range(1,nevals+1):
73  10000000   11350040.1      1.1     47.3          m = int_funct(y0,t0)
74  10000000    3369290.0      0.3     14.1          y[i] = y0 + dt * m
75  10000000    2644538.9      0.3     11.0          t[i] = t0 + dt
76  10000000    2347669.0      0.2      9.8          y0 = y[i]
77  10000000    2290336.6      0.2      9.6          t0 = t[i]
78                                                # end for.
```

Figure 1. Profiling results.

Once the profiling was performed, two variations were performed to the script, with the aim of reducing the runtime, by targeting the lines identified with **line_profiler**. The first implementation, Numba1, uses the **@jit** decorator, with the option **nopython=True** with the function **int_funct**. The second implementation, Numba2, uses the **@jit** decorator, with the option **nopython=True** with the function **int_funct**, as well as with the function **euler_integration**.

A comparison of the runtimes for the 3 script implementations is shown in Table 1.

Table 1. Runtime comparisons between script implementations of euler_ode.py .

Version	Runtime [s]
Pure Python	10.640769720077515
Numba1	7.007452726364136
Numba2	0.37809133529663086

It may be observed that the usage of the **@jit** decorators significantly reduces the script's runtime. With Numba1, the decorator is solely used with the **int_funct**, which is the biggest contributor to the runtime of the script. However, by using the decorator with both **int_funct**, as well as with **euler_integration**, the other lines which are contributing to the runtime are also addressed, thus, reducing the runtime considerably.

For the second part of the homework, **the num_int.py** script was executed and its time was measured. Then, Numba's **@jit** decorators were added to the functions **my_funct** and **integral_riemman**, using the **nopython=True** parameters. The results of these two implementations and the speed up factor are shown in Table 2.

Table 2. Runtime comparisons and Speed Up factor for num_int.py .

Version	Runtime [s]	Speed Up
Pure Python	87.618596	1
JIT decorators	1.707653	51.30936789

The next step was to automatically parallelize the implementation of the **num_int.py** script, using Numba. The only changes with respect to the previous Numba usage, were the addition of the **parallel=True** parameter to the **@jit** decorator in the **integral_riemman** function. Additionally, Numba's **prange** was used in the **For** loop within this function. The number of threads were varied between 1, 2, 4, 8, 16 and 20, using **numba.set_num_threads(n)**. The results of applying these changes are shown in Table 3 and are graphically displayed in Figure 2.

Table 3. Results of Numba's automatic parallelization in num_int.py .

Num Processors	Time [s]	Speed Up	Efficiency
1	1.71406	1	1
2	1.102486	1.55472269	0.77736135
4	0.784751	2.18420875	0.54605219
8	0.475271	3.60648977	0.45081122
16	0.3018	5.67945659	0.35496604
20	0.263549	6.50376211	0.32518811

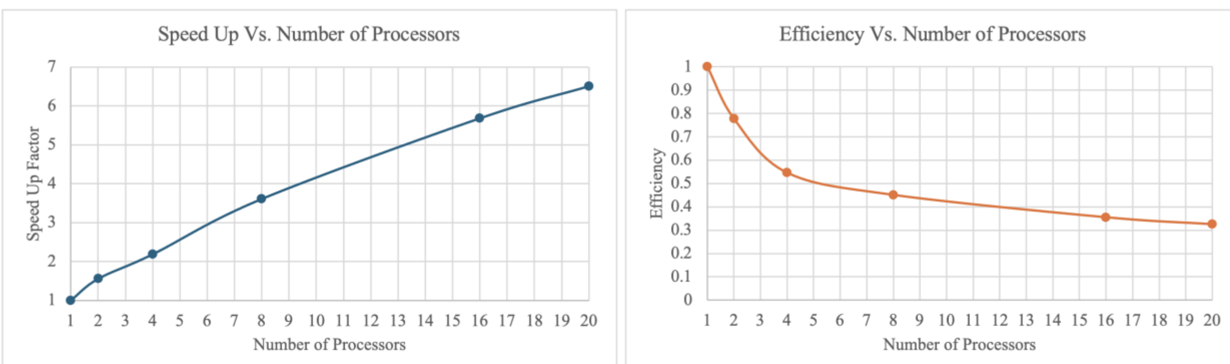

Figure 2. Speed Up Factor and Efficiency Vs. Number of Processors.

The results of the last problem are shown in Table 4. For this problem, an iterative function was created using Cython to execute the multiplication of matrices. The performance of this function was compared against Numpy's built-in function **np.dot** . Matrices of sizes 3x3, 10x10, 100x100 and 1000x1000 were used to compare the two approaches.

Table 4. Timing comparison of Cython Vs. Numpy Dot.

Matrix Size	**Time Per Multiplication [s]**	
	Cython	**Numpy Dot**
3x3	0.000001	0.000201
10x10	0.000006	0.000004
100x100	0.001964	0.000287
1000x1000	1.2035	0.003465