Homework 8
ME 5773 High Performance Computing Due: 04/10/2024 11:59pm

This is a team assignment. The teams are assigned by the instructors. The team members are posted on Canvas. The deliverable for this assignment is an email to the instructors notifying the completion of the assignment. The activities of this assignment should be stored in the Teams github repository, within a folder named "assignment8". This assignment will be evaluated using the most updated version of the repository **by the due date**.

1. Line profiler and Numba - Euler's Method.

Consider an ordinary differential equation (ODE) from an initial value problem (IVP) of the form

$$\frac{dy}{dt} = f(y(t), t), \qquad y(t_0) = y_0$$

The goal is to approximate $y(t)$ at different times, $y(t_i) = y_i$. Implement a pure python code that uses Euler's method to solve the differential equation. See https://tutorial.math.lamar.edu/classes/de/eulersmethod.aspx.

The main idea of Euler's method is to define a very small time increment, Δt, such that the derivative is approximated as $dy/dt \approx \Delta y/\Delta t$. One can approximate the solution at a following time t_{i+1} as follows:

$$y_{i+1} = \Delta t\, f(y_i, t_i) + y_i$$

where $t_{i+1} = t_i + \Delta t$ and $y_{i+1} = y(t_{i+1})$. This is performed until the function is evaluated at the upper limit integration time t_{max}.

Implement Euler's method in a python file and follow the instructions to evaluate the performance, profile and improve performance using Numba.

- Copy the template file `/work/ME5773/hw8/euler/euler_ode.py` into your assignment8 folder. Complete the Euler's method implementation in the function `euler_integration`.
- Use the following ODE as the test case

$$f(y, t) = \sin(t)\,, \;\; y(0) = -1$$

This problem has analytical solution $y(t) = -\cos(t)$, which you can use to evaluate the accuracy of the method. Evaluate until $t_{max} = 10^{-6}s$.

- After implementing this pure-python version, run the file and measure the total execution time. Report the results of this pure python implementation. Report this on a document.
- Copy the file `euler_ode.py` with your implementation into a new file named `euler_ode_profile.py`.
    - Modify the file `euler_ode_profile.py` to use line_profiler to evaluate which lines in the function `euler_integration` are taking the most time during the evaluation. Write the results in the report, and add a conclusion of what are the lines that contribute the most to the execution time.
- Copy the file `euler_ode.py` with your implementation into a new file named `euler_ode_numba1.py`.
    - Add Numba `jit` decorators with the option `nopython=True` to the `int_funct` function.

- Add a dummy call to the function before you evaluate the full integration such that Numba compiles the function. You can use a $t_{max} = 4\Delta t$.
- Run the file, and compare the execution time to the pure python version. What are your conclusions?

- Copy the file `euler_ode.py` with your implementation into a new file named `euler_ode_numba2.py`.
  - Add Numba `jit` decorators with the option `nopython=True` to both `int_funct` and `euler_integration` functions.
  - Add a dummy call to the function before you evaluate the full integration such that Numba compiles the function. You can use a $t_{max} = 4\Delta t$.
  - Run the file, and compare the execution time to the pure python version as well as the `euler_ode_numba1` version. What are your conclusions?

2. Numba – Automatic parallelization.

Copy the numerical integration file in the folder:

`/work/ME5773/hw8/integration/numInt.py`

Evaluate the performance of this pure python file and then use Numba's `jit` decorator with the `nopython=True` and evaluate the speedup of the performance. Be aware of the need to add a dummy function call to let the just-in-time compiler generate a compiled version.

Add `parallel=True`, as well as Numba's `prange` function to implement the numerical integration in a parallelized manner.

Evaluate the speedup and efficiency of the parallelized implementation for a number of processors including 1, 2, 4, 8, 16 and 20 using the NUMBA_NUM_THREADS environment variable. Generate a plot with the speedup and efficiency results, and write the results to the report.

3. Bonus: Cython – Matrix-matrix multiplication.

Use Cython to implement a matrix-matrix multiplication (MATMUL) function that takes two numpy arrays (two dimensions, float64 arrays) and performs the matrix multiplication as an iterative algorithm:

$$\mathbf{C} = \mathbf{AB}, \qquad \mathbf{A} \in \mathbb{R}^{n\times m}, \mathbf{B} \in \mathbb{R}^{m\times p}, \mathbf{C} \in \mathbb{R}^{n\times p}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Compare it to Numpy's matrix-matrix multiplication, the dot function (i.e. `C=np.dot(A,B)`). Evaluate the performance of the implementation for multiple sizes of matrices, including: 3x3, 10x10, 100x100 and 1000x1000.

Note: Since small matrix multiplication may take a very small time to execute, it is recommended that you evaluate the performance on a loop counting for, say 100 matrix multiplications, and average the total time to execute that loop by the number of iterations.

Write the results to the report.