Prof. Dr. A-S. Smith
am Lehrstuhl für Theoretische Physik I
Department für Physik
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Advanced Python for Research Projects

Exercise sheet 3: Parallelization and Scientific computation

Problem 3.1 *Parallel operators*

In this exercise, we want to work towards optimizing code through parallelization. As a first application, we will take our map/reduce functions from the previous exercise and calculate them in parallel. Especially the map function lends itself to parallelzation, because it works on distinct input data. Parallelization through data separation is possible. Because parallelization is prone to breaking stuff, we will also implement tests for our code to make sure it at least works to some degree. If you want to use the non-parallel version for reference in your tests, rename the functions you implement here to `reduce_parallel` and `map_parallel` to avoid name clashes.

a) Before we start coding: we want to optimize our map and reduce operations previously implemented in exercise 2 by parallelizing them with multiprocessing or multi-threading. Discuss, which option is more fitting for this problem, what the up or down sides are and what future changes to the GIL may change about the answer.

b) Implement the concurrent version of the `map` function, where you split up the data set and assign them to a number of processes. For this purpose, add an optional parameter `min_executor_data_count` with which you can configure the minimum number of data points to be processed by a single thread or process. Then use the provided iterators to collect all of the input data eagerly. To deal with multiple iterators being passed, you can use the `zip` function to get an iterator returning tuples of entries instead of dealing with the iterators independently. For each of the executors (threads or processes), provide them with a slice of your collected data array that contains at least `min_executor_data_count` entries. Also make sure, that you do not create more workers than there are threads available on the CPU. You should distribute the data as equally as possible among the workers. Because you provide multiple arguments to the executor, you cannot just pass the function `f` that is supposed to be called on the data directly to the executor. Instead, create a local function that gets passed the function `f` and the list of data for the function `f` and then iteratrively calls f on each of the entries. (Use the `*` prefix to unwrap arguments for a function from a tuple). Eventually add documentation to your `map` function to allow for other people using it. Explain all parameters, its parallel implementation and the requirements that this imposes on the function`f`. Also declare if you are using processes that the function cannot access shared data.

c) Add a Test class for the parallel `map` function using python's `unittest` framework (see lecture slides) in which you implement a few unit tests to check the functionality of your `map` function. Some scenarios you should check:

1. Make sure that your `map` function works correctly with one input iterator/list and a function taking one input (e.g. a function calculating the square of its argument). You can check the results by iteratively calculating the intended result in a for loop and then comparing with the result of the `map` invocation.
2. Make sure it works with a function taking multiple arguments and multiple iterators (e.g. a function calculating the sum of two lists).
3. Make sure it does not break if you provide an empty list.

4. Make sure the `map` invocation fails if the number of iterators does not satisfy the number of arguments required by the function. (It should fail automatically, but add a test that ensures that it does fail in this scenario)

d) We want to implement a parallel `reduce` function, where we change the semantics a bit: We now want to apply the divide-and-conquer technique, in that our parallel version splits the dataset into subsets that are reduced in parallel with the classical reduce function. Afterwards we will have to combine the individual reduce results back into one overall result. For this, add an additional argument `combine_function`, which will take the result of a `reduce` call and returns the combined result. Also add an optional parameter `min_executor_data_count` with similar semantics as in the `map` function written before. First exhaust the input iterator to retrieve all data that needs to be distributed across workers. Then slice the data as above and run the classical reduce function on the slices. You may even use the `map`-function to run the classical reduce-function on the slices of data or you can use a different approach. (Note: the input for the map function then needs to be an array holding the slices of data that the reduce function should be called upon as well as the initial argument to your reduce function.) After the parallel reduce calls have finished, use the `combine_function` to combine the results into one final result. For this, you can use a classical `reduce` call with the `combine_function` and the results of the parallel execution. Add appropriate documentation to your parallel `reduce` function.

e) Add a Test class for the parallel `reduce` function using the pytest framework in which you implement a few unit tests to check the functionality of your `reduce` function. Some scenarios you should check:

1. Make sure that your `reduce` function works correctly with an empty input list and an initial argument (it should return the initial argument).
2. Make sure it runs correctly if there is only one executor, i.e. not enough data entries for more than one worker. (this should yield the same result as the non-parallel reduce function)
3. Make sure that the combine function is applied correctly by running with more data than is required for 2 workers.
4. Make sure that the return value of your reduce function is reasonable or that it throws a (documented) exception in case it is called with an empty iterator and no initial state.

f) We want to create a so-called Map-Reduce functionality, where first the data is mapped in parallel with a given function applied to each data point and then the reduce function is applied to the results of the map operation. Create this new function called `map_reduce`, taking a mapping function `func_map`, a reduce function `reduce_func` with a potential initial value `reduce_initial`, (a combine function `combine_func` if you want to use a parallel `reduce` function internally), and an arbitrary number of iterators as an argument `*iterators`. Use the parallel `map` implementation to first apply `func_map` to the input iterators and then apply the reduction with `reduce_func` to the results (with `combine_func`) using the initial value parameter. Add documentation and tests to ensure its correct performance by calculating the sum of squares, cubes and seventh powers of an input list. Consider, which other kinds of scenarios you should be testing here.

Problem 3.2 *Using numpy and scipy for scientific calculation*

In this exercise, we want to get familiar with some helpful scientific packages. Create a new file to implement the following functionality:

a) Write a function `get_random_array` that takes the number of desired entries `n` and generates an array of `n` normal-distributed floating point numbers. When you are done, make the mean and variance of the normal distribution configurable via optional parameters of the function. If

you want, extend the parameter `n` to allow for a shape tuple to be passed. The function should then generate an array of that shape with the desired distribution.

b) Practise how to choose every number at an odd index, every number at an even index, every number in the first half of an array, and every number in the second half of an array using slices. Once you are done, combine your insights to pick only entries at odd positions in every line in the last third of a 2d-array and set them to zero.

c) Look at the documentation of `numpy.loadtxt` to load a file into memory as a 2d-array. Figure out how to ignore the third column in a file in case it contains text data, whereas every other column is a floating point number.

d) Use the result of the `get_random_array` as an input for `np.histogram`, to generate a histogram of the randomly distributed values your function generates. Use your function to generate a 2d-array with (x,y)-entries and use that in combination with `numpy.histogram2d` to calculate a histogram of 2-dimensional data.

e) Use matplotlib to visualize your 1d- and 2d-histograms

f) Use `scipy.optimize.curve_fit` to try and fit a normal distribution (also available in scipy) to your histogram to compare the fit parameters to your actually chosen parameters.

Problem 3.3 *Rebuilding numpy and scipy functionality ourselves*

As a programmming exercise, we want to implement a few simple numerical functions on our own:

a) Implement a function `get_center_derivative` that takes two numpy arrays `x` and `y`, representing coordinates of points on a function graph. You may assume that the `x` are evenly spaced, but they may be unsorted. Use `numpy.argsort` to sort both `x` and `y` by their `x`-positions. Then calculate the central derivatives $(y_{i+1} - y_{i-1})/(x_{i+1} - x_{i-1})$ by employing clever slices instead of a loop or other forms of iterations. Return the resulting array of numerical central derivatives and add basic documentation. The resulting array should have 2 fewer entries than the input.

b) With similar inputs as in the previous problem in a function `get_trapezoid_integral` , implement numerical integration of the resulting discrete function graph (after sorting by x), using the trapezoid rule: $(y_{i+1} + y_i)(x_{i+1} - x_i)/2.0$. This can again be achieved via efficient slicing and using the `numpy.sum` function as a reduce functionality.

c) Implement a root finding algorithm via bisection in a function `get_biscection_root`. The function should take an argument `x_min` and an `x_max` setting the boundary for the search for a root of a function `f` at a maximum of `max_steps` iterations, where the default value for the latter should be 20. First check, whether `f` has different signs on the boundary of the chosen interval. If not, log an error and throw an exception. (Set up a logger for this file based on what you did in the first exercise). Then iterate as follows a maximum of `max_steps` times: In each iteration, calculate the middle point `x_mid` of the current interval [`x_min`, `x_max`]. Evaluate `f` on `x_mid`. Then replace the boundary of the interval, where `f` has the same sign as in the middle with `x_mid`. At the end, after the iterations have finished, return the middel of the final interval as a result for the best guess for the root of the function.

d) Add Documentation for the above functions and at least one unit test each.

Problem 3.4 *Parallel numpy/scipy replacement*

a) To try and mimic numpy's internal parallellization, use your previously implemented parallel map and reduce functions, to write parallel versions of the integration and derivative functions written in the prior exercise. Use the fact that the individual terms of the central derivative as well as the individual pieces of area under the function graph in the integration can be calculated in parallel. Potential combinations into the final result can be implemented using `reduce`.

b) Add tests to make sure, the operation is identical to the result obtained by numpy/scipy by generating random y-coordinates and equally spaced x-coordinates (see: `numpy.linspace`) and putting them into your function and the corresponding numpy (gradient)/scipy (trapezoid) function.

c) Write tests to make sure that your custom-built functions perform the way they are expected by adding a few hand-picked cases. E.g. integrate or calculate the derivative for a linear or constant function.

d) Put the tests for the numpy-imitations in a different file than the original function implementation.

e) Group the tests appropriately so that differentiation, integration and root finding have their own test group and can be run independently, but so that all tests belonging to each of the individual functions are run together.

Problem 3.5 *Measuring performance*

We want to use the test functions as a template to write a function to compare the speed of numpy and scipy functions to our own implementation

a) Look up how to measure time in python at the microsecond scale and output the time that it takes your parallel function and the numpy function to calculate the result. To get meaningful results, generate arrays with at least a few thousand or a million entries.

b) What do we learn from the comparison in performance?

c) Plot how the execution time of the different functions (derivative, integration) behaves as a function of the input size using matplotlib and sufficiently many datapoints.

d) Why does one solution perform so much better?