

Parallel programming

Python Numba - 2





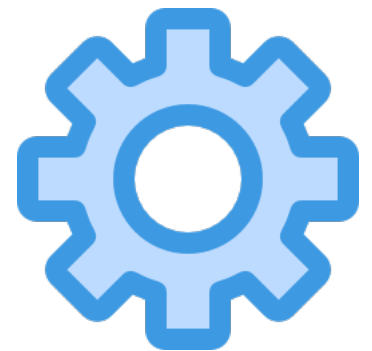
Automatic Parallelization in Numba

- ***Performance Boost:***
 - Harness the full potential of your CPU cores
 - Speed up computationally intensive tasks
- ***Simplicity and Readability:***
 - No need for complex parallel programming constructs
 - Write code in Python as usual and let @njit handle the parallel magic



How it works

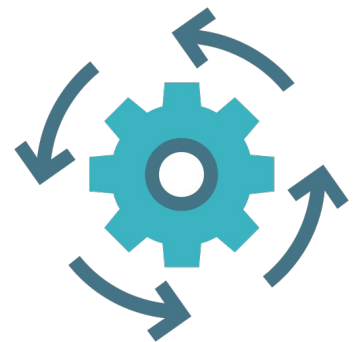
- **@jit's** parallel option automates parallelization and optimizations
- Identification of operations with parallel semantics
- Fusion of adjacent operations to form parallel kernels
- **Fully automated** process without user program modifications





Automatic Parallelization

- Setting the parallel option `@jit(parallel = True)` allows to ***automatically*** parallelize a function or its part and perform other optimizations
- Numba attempts to identify such operations in a user program, and fuse adjacent ones together, to form one or more kernels that are ***automatically run in parallel***





Supported operations

- All the operations which include **common arithmetic** functions between arrays and scalars:
 - Unary operations (+, -, ~)
 - Binary operations (+, -, *, /, %, >>, <<,)
 - Comparison operators (==, !=, <, >, <=, >=)
- Additionally Numba provides support for Numba ***ufunc*** (only in ***nopython*** mode) and user-defined ***DUFunc*** through ***vectorize()***



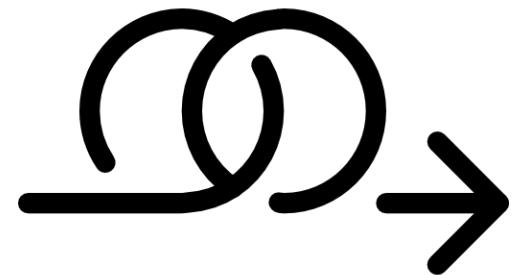
Supported **numpy** functions

- numpy **reduction** functions (*sum, prod, min, max, argmin, argmax*)
- numpy **math** functions (*mean, var, std*)
- numpy **array creation** functions (*zeros, ones, array, linspace*)
- numpy *dot()* function
- *Reduce* operator for 1D numpy arrays



Explicit Parallel Loops

- Another feature of the code is the support for ***explicit parallel loops*** (again, add “parallel=True” into `@jit`)
- One can use numba’s `prange()` instead of `range()` to specify that a loop can be parallelized
- **Warning:** the loop must not have cross iteration dependencies except for supported reductions





Example 1: Automatic Parallelization

- See the example of **automatic parallelization** in the provided .ipynb notebook with example codes

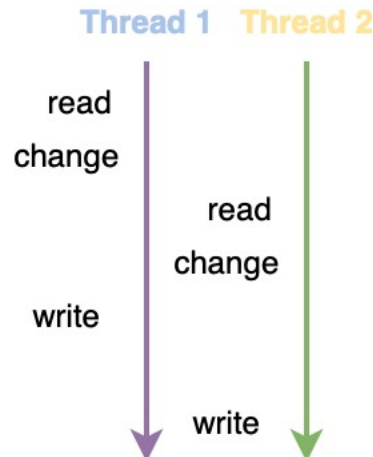




Beware race condition!

- Care should be taken, however, when reducing into **slices or elements** of an array
- If the specified elements are written to **simultaneously** by multiple parallel threads, a race condition would occur

Race condition





Example 2: Race Condition

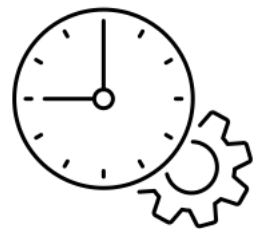
- See the example of **race condition** in the provided .ipynb notebook with example codes





Scheduling of parallel task

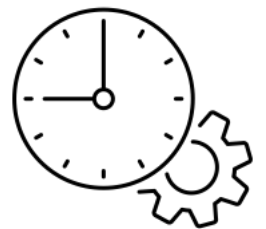
- By default, *Numba* divides the iterations of a parallel region into chunks
- Approximately ***equally sized chunk*** is given to each configured thread
- This scheduling approach is equivalent to static scheduling in OpenMP





Scheduling of parallel task

- Conversely, if the work per iteration varies significantly, static scheduling approach leads to load imbalances
- Numba provides a mechanism to control how many iterations of a parallel region (i.e., the chunk size) go into each chunk





Example 3: setting the chunk size

- See the example of **setting the chunk size** in the provided .ipynb notebook with example codes





Parallel diagnostics report

- The parallel option in `@njit` provides diagnostic information
- Two ways to access diagnostics:
 - Environment Variable:
 - Set `NUMBA_PARALLEL_DIAGNOSTICS` to enable diagnostics
 - Convenient for controlling diagnostics globally
 - Function Call:
 - Use `parallel_diagnostics()` to access the same information
 - Enables fine-grained control and flexibility





Parallel diagnostics report

- Level of Verbosity:
 - Set an integer argument (1 to 4) to control verbosity
 - 1: Least verbose, 4: Most verbose
- Leverage **@njit** diagnostics: empower your parallelized code with insights!





Example 4: diagnostic report

- See the example of **calling the diagnostic** in the provided .ipynb notebook with example codes





Coding exercise: π Calculation

- Implement the Monte-Carlo calculation of π using Numba automated parallelization:
 - access the provided skeletons
 - accelerate the process by automating the parallelization
 - accelerate the process by setting an explicit chunk size
 - call the diagnostic report

$$\pi = 4 * \frac{\text{no. of points generated inside the circle}}{\text{no. of points generated inside the square}}$$



References

- **Fundamental tutorial on numba:**

<https://numba.readthedocs.io/en/stable/cuda/index.html>

- **Selected pages:**

<https://numba.readthedocs.io/en/stable/user/parallel.html>

<https://numba.readthedocs.io/en/stable/user/performance-tips.html>