# Parallel programming

# Python Numba. Part 2

# 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.
- At the moment, this feature only works on CPUs.

# Supported operations

These are operations on *Numba* arrays which include common arithmetic functions between numpy arrays, between arrays and scalars, as well as numpy ufuncs:

- Unary operations (+, -, ~)
- Binary operations (+, -, *, /, %, >>, <<, ….)
- Comparison operators (==, !=, <, >, <=, >=)
- Numba *ufunc* (only in *nopython* mode)
- User-defined *DUFunc* through *vectorize()*

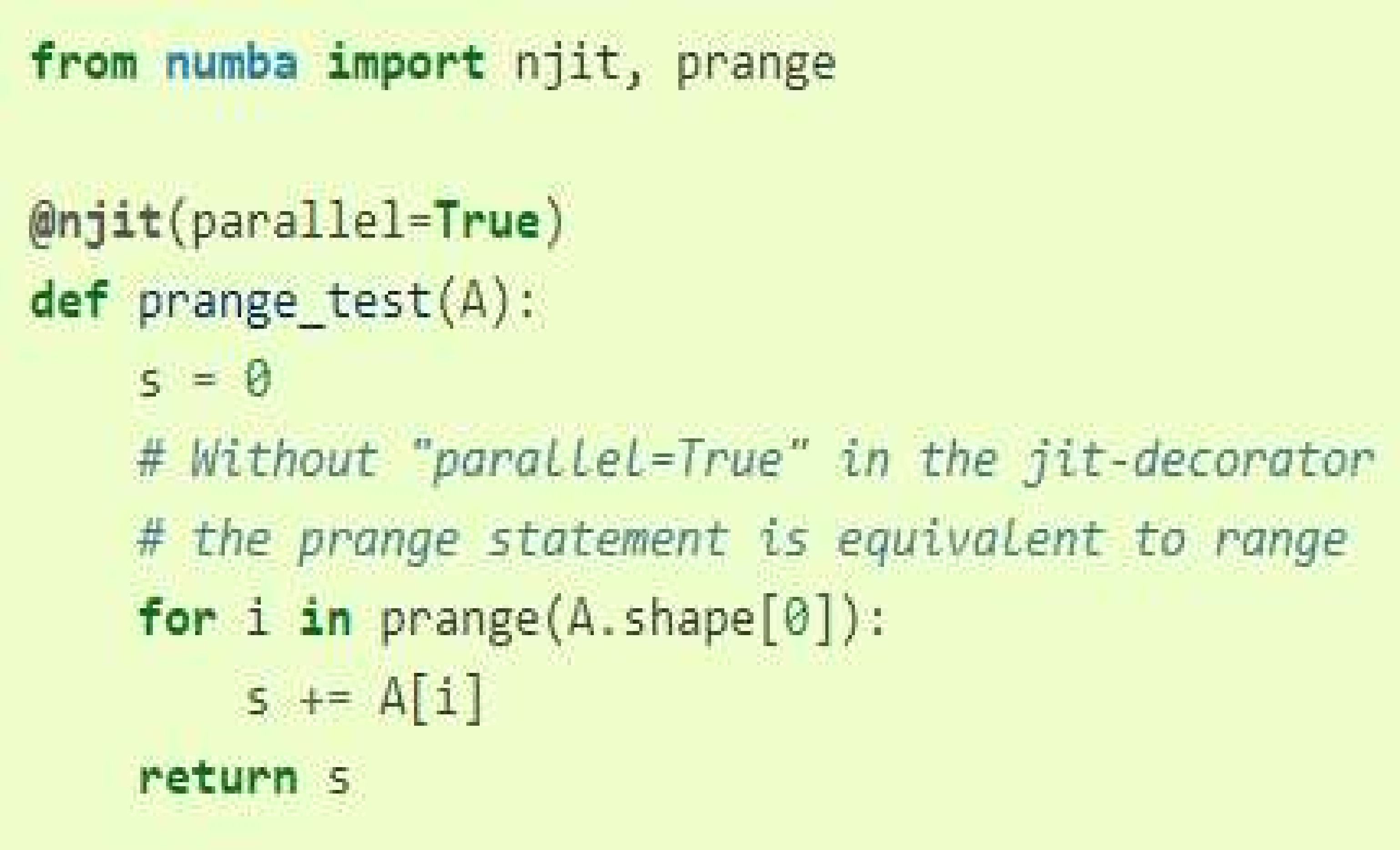
# Supported functions

- 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

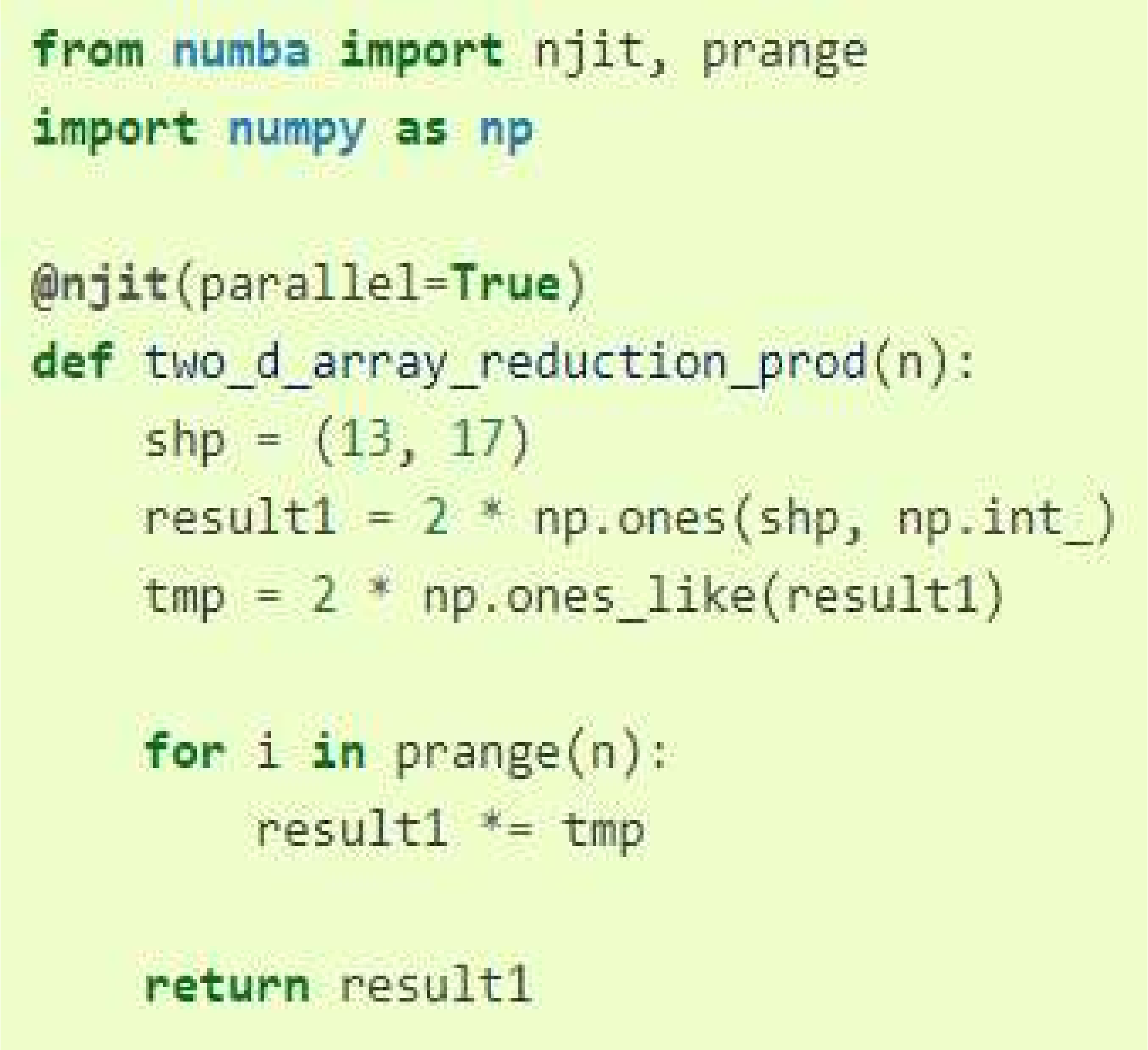
- Another feature of the code is the support for explicit parallel loops (again, add “parallel=True” into @*jit*) .
- One can use numba’s *prange()* instead of *range()* to specify that a loop can be parallelized.
- Warning: the loop must not have cross iteration dependencies except for supported reductions

# Example 1: A parallel loop with reduction



```python
from numba import njit, prange

@njit(parallel=True)
def prange_test(A):
    s = 0
    # Without "parallel=True" in the jit-decorator
    # the prange statement is equivalent to range
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

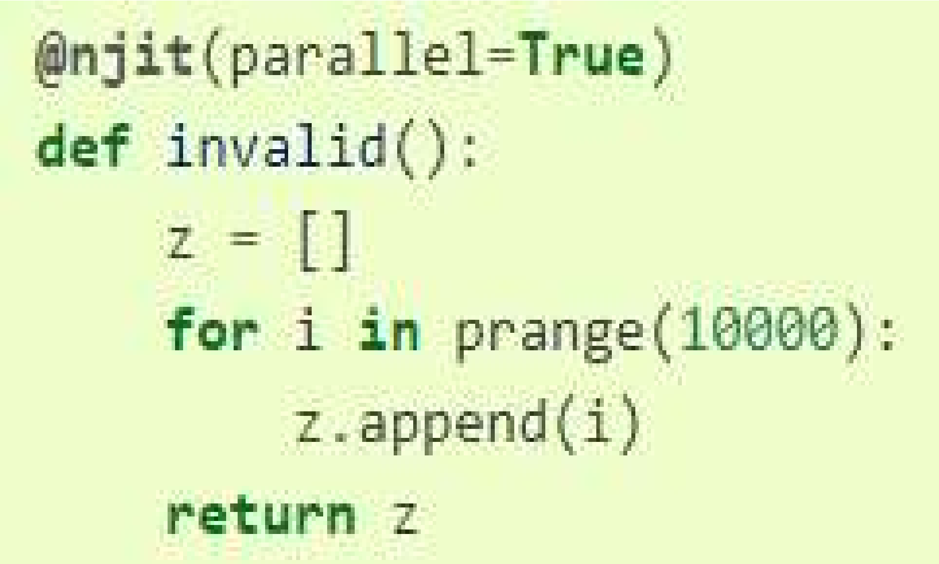
# Example 2: reduction on 2D array



```python
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def two_d_array_reduction_prod(n):
    shp = (13, 17)
    result1 = 2 * np.ones(shp, np.int_)
    tmp = 2 * np.ones_like(result1)

    for i in prange(n):
        result1 *= tmp

    return result1
```

# Unsupported operations

Concurrent *write* operations on container types (i.e., lists, sets and dictionaries) in a *prange* parallel region are not threadsafe:



```
@njit(parallel=True)
def invalid():
    z = []
    for i in prange(10000):
        z.append(i)
    return z
```

# 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

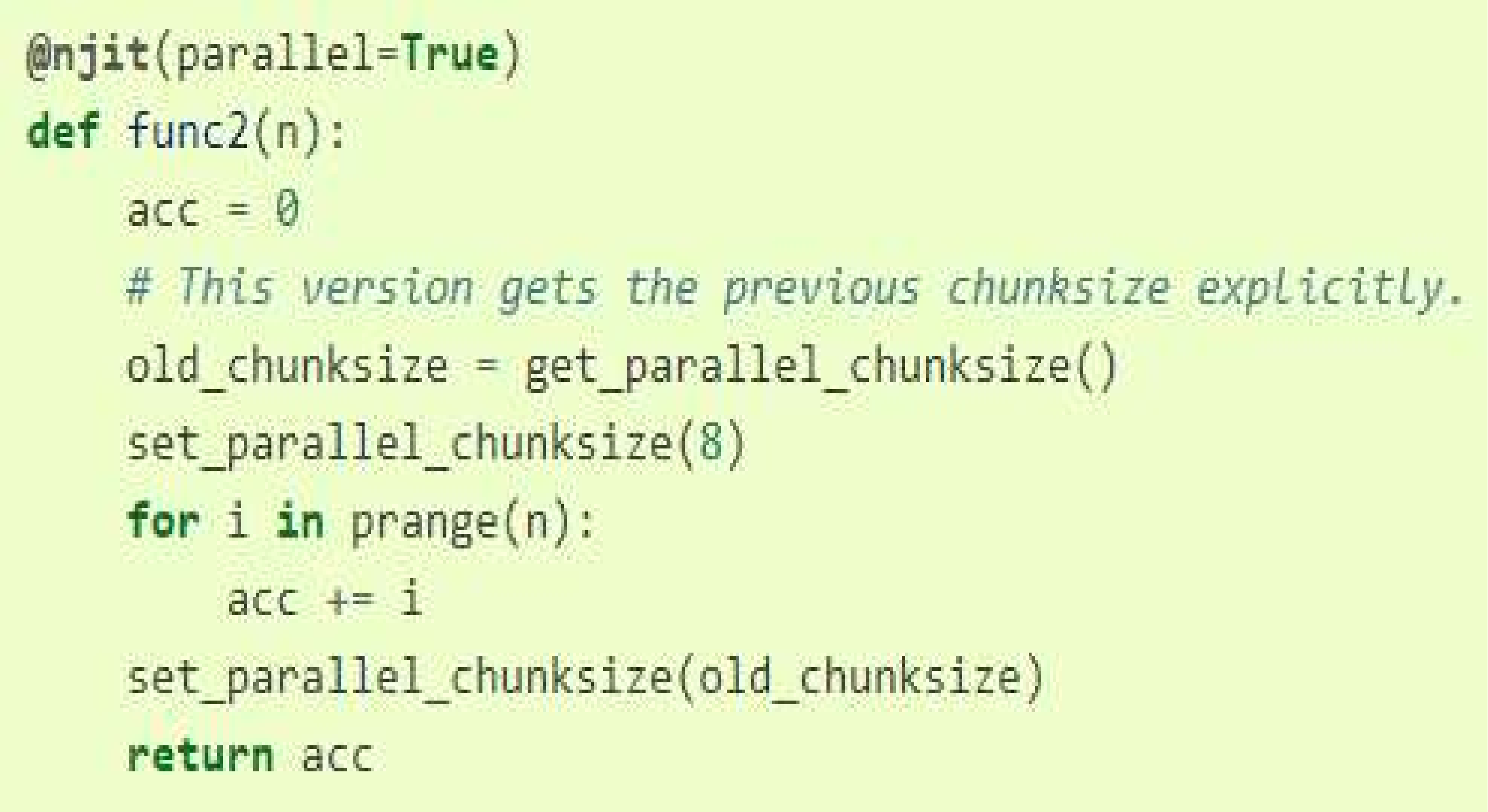
# Scheduling of parallel task

- 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.

- This approach is similar to OpenMP’s dynamic scheduling option with the specified chunk size.

# Example: setting the chunk size

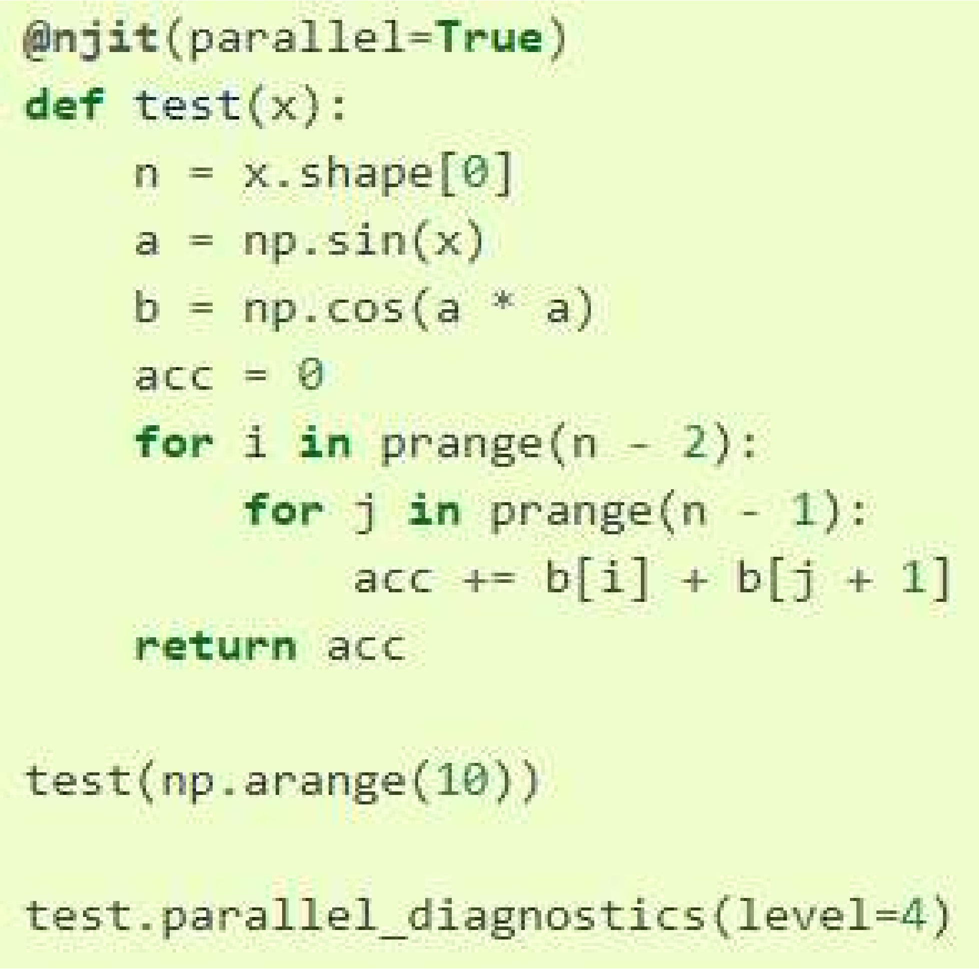


```python
@njit(parallel=True)
def func2(n):
    acc = 0
    # This version gets the previous chunksize explicitly.
    old_chunksize = get_parallel_chunksize()
    set_parallel_chunksize(8)
    for i in prange(n):
        acc += i
    set_parallel_chunksize(old_chunksize)
    return acc
```

# Parallel diagnostics report

- The parallel option for *@jit* can produce diagnostic information about the automatic parallelizing of the code
- The first way to access it is by setting the environment variable NUMBA_PARALLEL_DIAGNOSTICS.
- The second way is by calling parallel_diagnostics(), both methods give the same information and print to STDOUT.

# Simple diagnostics example



```
@njit(parallel=True)
def test(x):
    n = x.shape[0]
    a = np.sin(x)
    b = np.cos(a * a)
    acc = 0
    for i in prange(n - 2):
        for j in prange(n - 1):
            acc += b[i] + b[j + 1]
    return acc

test(np.arange(10))

test.parallel_diagnostics(level=4)
```

# Overview of other performance tips

- ***Nopython mode:*** getting functions to compile under it can be the key to good performance.

- Numba supports most of *numpy.linalg* in nopython mode.

- ***Fastmath:*** it is possible to relax some numerical rigour gaining additional performance of the *fastmath* keyword argument: *@njit(fastmath = True)*

# 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