# 1. Parallel threads of execution

One way to improve the performance of the execution of a program is to use **explicit parallel loops**, so that several iterations of the loop are distributed among different threads of execution, and can be executed concurrently (simultaneously), using the multiple processing cores available in a current processor. The user is required to make sure that the loop does not have **cross-iteration dependencies** except for supported **reductions**.

A **cross-iteration dependence** occurs when the input data for the $i^{th}$ loop iteration is generated by one of the previous loop iterations (frequently the $(i-1)^{th}$ loop iteration). Cross-iteration dependences force the execution of iterations to be sequential. If the $i^{th}$ loop iteration uses the value of a certain variable *v*, and is executed at the same time as the $(i-1)^{th}$ loop iteration that writes a value on this same variable *v*, then the execution may use the old value of the variable instead of the updated value. If a program is parallelized, then the result of the execution may be different on different executions, sometimes the result could be correct and sometimes could be incorrect. This program's behavior is called **non-determinism**, and is generally not desired.

A **reduction** is a special cross-iteration dependence that can be transformed for correct parallelization. The simplest reduction is the accumulation of the values generated by each of the iterations, for example the addition of all the elements in a list. Since the addition operation is **associative**, instead of computing the addition of **N** values as a **sequence** of additions, like ((((a[0]+a[1])+a[2])+a[3])+ ...), the operations can be rearranged to exhibit parallelism, like (((a[0]+a[1])+(a[2]+a[3]))+ ... ). Other associative operations, like multiplication, minimum, maximum, average ... can also determine reductions, and are suitable for parallelization. The compiler may infer a reduction automatically if a variable is updated by a binary function/operator using its previous value in the loop body. The initial value of the reduction is inferred automatically for `+=` and `*=` operators (additions are initiated with 0 and multiplications are initiated with 1). For other functions/operators, the reduction variable should hold the identity value right before entering the parallel loop. Reductions in this manner are supported for scalars and for arrays of arbitrary dimensions.

## Global Interpreter Lock (or GIL)

The **GIL** is the mechanism used by the CPython interpreter to assure that only one thread executes certain Python operations at a time. This simplifies the CPython implementation by making the object model implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines. However, some extension modules are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

**Acquiring and Releasing the GIL**:

This may be useful when calling from multi-threaded code into Cython or C code that may block, or when wanting to use Python from a (native) C o Cython thread callback. Releasing the GIL should obviously only be done for thread-safe code or for code that uses other means of protection against race conditions and concurrency issues. Note that acquiring the GIL is a blocking thread-synchronizing operation, and therefore potentially costly. It might not be worth releasing the GIL for minor calculations. Usually, I/O operations and substantial computations in parallel code will benefit from it.

You can release the GIL around a section of code using the **with nogil** statement:

```
with nogil:
    <code to be executed with the GIL released>
```

Code in the body of the with-statement must not raise exceptions or manipulate Python objects in any way, and must not call anything that manipulates Python objects without first re-acquiring the GIL. Cython validates these operations at compile time, but cannot look into external C functions, for example. They must be correctly declared as requiring or not requiring the GIL in order to make Cython's checks effective. A C (Cython) function that is to be used as a callback from C code that is executed without the GIL needs to acquire the GIL before it can manipulate Python objects. This can be done by specifying **with gil** in the function header:

```
cdef void my_callback(void *data) with gil:
    ...
```

If the callback may be called from another non-Python thread, care must be taken to initialize the GIL first. If you are already using cython.parallel in your module, this will already have been taken care of. The GIL may also be acquired through the **with gil** statement:

```
with gil:
    <execute this block with the GIL acquired>
```

You can specify **nogil** in a C function header or function type to declare that it is safe to call without the GIL:

```
cdef void my_gil_free_func(int spam) nogil:
    ...
```

When you implement such a function in Cython, it cannot have any Python arguments or Python object return type. Furthermore, any operation that involves Python objects (including calling Python functions) must explicitly acquire the GIL first, e.g. by using a **with gil** block or by calling a function that has been defined **with gil**. These restrictions are checked by Cython and you will get a compile error if it finds any Python interaction inside of a **nogil** code section.

**Note**: the **nogil** function annotation declares that it is safe to call the function without the GIL. It is perfectly allowed to execute it while holding the GIL. The function does not in itself release the GIL if it is held by the caller. Declaring a function **with gil** (i.e. as acquiring the GIL on entry) also implicitly makes its signature **nogil**.

## Using Parallelism with Cython

Cython supports native parallelism through the **cython.parallel** module. To use this kind of parallelism, the GIL must be released. It currently supports OpenMP, but later on more backends might be supported. One can use **prange** instead of **range** to specify that a loop can be parallelized.

cython.parallel.**prange**(*[start,] stop[, step][, nogil=False][, schedule=None[, chunksize=None]][, num_threads=None]*)

This function can be used for parallel loops. OpenMP automatically starts a thread pool and distributes the work according to the schedule used. Thread-locality and reductions are automatically inferred for variables. If you assign to a variable in a prange block, it becomes lastprivate, meaning that the variable will contain the value from the last iteration. If you use an in-place operator on a variable, it becomes a reduction, meaning that the values from the thread-local copies of the variable will be reduced with the operator and assigned to the original variable after the loop. The index variable is always lastprivate. Variables assigned to in a parallel with block will be private and unusable after the block.

- **start** – The index indicating the start of the loop (same as the start argument in range).
- **stop** – The index indicating when to stop the loop (same as the stop argument in range).
- **step** – An integer giving the step of the sequence (same as the step argument in range). It must not be 0.
- **nogil** – This function can only be used with the GIL released.
- **schedule** – static / dynamic / guided / runtime
- **num_threads** – The num_threads argument indicates how many threads the team should consist of. If not given, OpenMP will decide how many threads to use. Typically, this is the number of cores available on the machine.
- **chunksize** – The chunksize argument indicates the chunksize to be used for dividing the iterations among threads.

## Example

Example with a typed *memoryview* (e.g. a NumPy array):

```
from cython.parallel import prange
def func (double[:] x, double alpha):
    cdef int i
    for i in prange(x.shape[0]):
        x[i] = alpha * x[i]
```

Example with a reduction:

```
from cython.parallel import prange
cdef int i, n = 30, sum = 0

for i in prange(n, nogil=True):
    sum += i
print(sum)
```

cython.parallel.**parallel**(*num_threads=None*)

This directive can be used as part of a with statement to execute code sequences in parallel. This is currently useful to setup thread-local buffers used by a **prange**. A contained prange will be a worksharing loop that is not parallel, so any variable assigned to in the parallel section is also private to the prange. Variables that are private in the parallel block are unavailable after the parallel block.

Example with thread-local buffers:

```
from cython.parallel import parallel, prange
from libc.stdlib cimport abort, malloc, free

cdef int idx, i, n = 100
cdef int * local_buf
cdef size_t size = 10

with nogil, parallel():
    local_buf = <int *> malloc(sizeof(int) * size)
    if local_buf is NULL:
        abort()

    # populate our local buffer in a sequential loop
    for i in xrange(size):
        local_buf[i] = i * 2

    # share the work using the thread-local buffer(s)
    for i in prange(n, schedule='guided'):
        func(local_buf)

    free(local_buf)
```

cython.parallel.**threadid**()

Returns the id of the thread. For n threads, the ids will range from 0 to n-1.

## Compiling

To actually use the OpenMP support, you need to tell the C or C++ compiler to enable OpenMP. For gcc this can be done by adding the –fopenmp flag when invoking the compiler.

**We want to parallelize the sequence alignment code using the prange construct. First, you must identify which execution tasks can be performed independently and in parallel. Then, you have to find out how to modify the code for parallel execution. Ask your teacher for confirmation and/or for help in this assignment. Do not start codifying before going to the next page.**