OpenMP for Python User Guide

Caleb Huck

June 8th, 2021

1. Introduction

OpenMP (Open Multiprocessing) is a shared-memory, thread-based, parallel library that was originally written for FORTRAN, C, and C++. OpenMP removes the responsibility for thread creation and management from the user, allowing them instead to describe how the code should be run in parallel and what concurrent dependencies exist. This is done through the uses of directives, clauses, and a runtime library. As far as we are aware, this project is the first effort to bring OpenMP to Python. Python is a very different language than any of the original supported languages, and therefore is not a “one-to-one” with the original API. With that in mind, the purpose of this user guide is to inform the reader on how to use this software and the differences from the original OpenMP. Therefore we assume prior understand and experience with OpenMP. If you don’t know OpenMP, or need to brush up before continuing with this document, there are many excellent online resources, such as Tim Mattson’s A ”Hands-on” Introduction to OpenMP or the Lawrence Livermore National Laboratory OpenMP Tutorial or any one of many others available.

2. Quick Note About Data Structures

Python lists are implemented as objects that allow array-like syntax, and therefore do not behave like primitive C arrays. They are automatically locked for both reading and writing, even if reading or writing to different indices in the list. This means that any parallel program with a significant amount of shared list access will not see good speedup, if any at all. As a partial workaround for this problem, we recommend using the jarray.array or jarray.zeros functions. These functions return an object that wraps an underlying primitive Java array with nearly identical behavior and member functions to regular lists, and are not locked for reading (unfortunately, writing is still locked). Their signatures are given below, along with a chart of supported types and an example.

```
jarray.array(sequence, type)
jarray.zeros(length, type)
```

Character Typecode	Corresponding Java Type
z	boolean
c	char
b	byte
h	short
i	int
l	long
f	float
d	double

*Python types can also be used, as shown in the example below

```
from jarray import array, zeros

arr = array([1, 2, 3], 'i')
arr_z = zeros(3, float)

arr.append(4)
for i in range(len(arr_z)):
    arr_z[i] += 1
```

```
arr = [1, 2, 3]
arr_z = [0.0, 0.0, 0.0]

arr.append(4)
for i in range(len(arr_z)):
    arr_z[i] += 1
```

3. Syntax and Usage

This section covers each major OpenMP construct supported, along with the clauses that can be used in conjunction with them. The function will be briefly explained, along with any major differences from the original OpenMP version. Note that all directives are in the form of a Python comment that starts with #omp (we omitted the "pragma" present in the C/C++ OpenMP since it has no meaning in Python). Syntax examples are also provided.

3.1. Parallel

The parallel directive has the effect of spawning threads to run the subsequent code block in parallel. It is the only directive that the num_threads() clause can be used with. If the num_threads() is not included, then the value returned from os.cpu_count() will be used by default. The parallel directive can also include any variable scoping clause or the reduction clause.

```
from omp import *

#omp parallel num_threads(2)
    id = omp_get_thread_num()
    print('Hello from thread: ', id)
```

3.2. For

The `for` directive in OpenMP is a convenience directive for for-loop partitioning among threads. This directive splits the iterations among the available threads according to the schedule set by the `schedule()` clause (if not present, it will be set to static, with a chunk size of approximately the number of iterations divided by the number of threads, by default). Dynamic and guided scheduling is also supported. Because the for-loop construct in Python is really a for-each loop, we enforce a particular structure when using the `for` directive. The for-loop following a `for` directive must be of the form `for [single variable] in range([1, 2, or 3 parameters]):`. If one parameter (`n`) is passed to `range()`, then the returned range will be `[0, n-1]`. If two parameters are passed (`k`, and `n`), then the range will be `[k, n-1]`. Finally, if three parameters are passed (`k`, `n`, and `i`), then the range will be `[k, k+i, k+2i, k+3i, ..., n-1]`. This ensures that OpenMP for-loops will behave as close to C-like for-loops as possible.

```
from omp import *

#omp parallel num_threads(2)
    #omp for schedule(dynamic, 5)
        for i in range(0, 100, 2):
            print('Iteration: ', i')
```

3.3. Parallel For

The `parallel for` directive combines the functions of the previous two directives into a single directive. It is functionally identical to the `parallel` directive, directly followed by the `for` directive.

```
from omp import *

#omp parallel for num_threads(2) schedule(guided, 10)
    for i in range(1, 100):
        print('Iteration: ', i)
```

3.4. Barrier

The `barrier` directive is used for synchronization. It ensures that no thread will continue past the barrier until all threads have made it to the barrier.

```
from omp import *

#omp parallel
    print('first print statement')
    #omp barrier
    print('All threads have completed first print statement')
```

3.5. Critical

The `critical` directive protects critical sections within a parallel region. A critical section is any code that could result in a race condition if it is executed by multiple threads in parallel. The `critical` directive is used to serialize this portion of code so that only one thread may execute it at a time.

```
from omp import *

sum = 0
#omp parallel for shared(sum)
    for i in range(10):
        #omp critical
            sum += 1
```

3.6. Reduction

The `reduction` clause is a convenient method for assigning each thread a private variable, and then performing some aggregation operation across all of them at the end of the block and placing the result back in the variable with the same name from the outer scope. This works by providing a variable name and an operation as arguments to the clause. Then each thread can access the variable just like a private variable without worrying about synchronization. After the block executes, the result will automatically be placed into the original variable. The operations that are supported are as follows: +, -, *, & (bit level AND), | (bit level OR), ˆ (bit level XOR), && (logical AND), || (logical OR), max, and min.

```
from omp import *

sum = 0
#omp parallel for reduction(+:sum)
    for i in range(10):
        sum += 1
```

3.7. Master/Single

Currently, the `master` and `single` directives have the same effect. These directives ensure that only the thread with ID 0 will execute the following block. In a future update, we plan to differentiate between the two by allowing `single` blocks to be executed by the first thread that encounters the directive, regardless of whether or not it is thread 0, which is the behavior of the original OpenMP version.

```
from omp import *

#omp parallel
    print(’printed by all threads’)
    #omp master
        print(’only printed by thread 0’)
    #omp single
        print(’only printed by thread 0’)
```

4. Runtime API

All runtime functions exist in the omp module. The directory where this module exists is added to the python path automatically, so the user can import it normally. Currently our runtime API supports three function calls. The first is `omp_get_thread_num()`. This function returns a unique ID corresponding to the thread that calls it. The IDs are always assigned starting with 0 (the master thread) and then incrementing by one for each additional thread created. Therefore, the programmer can assume that whatever threads are available in a given parallel region have IDs `0` through `p-1` where `p` is the number of threads executing the region.

The second function we provide is `omp_get_num_threads()`. This function returns the number of available threads executing a parallel region.

Finally, `omp_get_wtime()` is a wrapper around the Python `time.time` function and returns the number of seconds since some arbitrary time in the past that is guaranteed not to change during the execution of the program. Below is a table summarizing each API call, along with an example showing how they can be used.

API Call	**Description**
`omp_get_thread_num()`	returns integer thread ID
`omp_get_num_threads()`	returns integer representing the number of threads active in the current scope
`omp_get_wtime()`	returns the number of seconds since some arbitrary time in the past

```
from omp import *

start = omp_get_wtime()
#omp parallel num_threads(5)
    print('printed by thread ', omp_get_thread_num(), ' of ', \
   omp_get_num_threads())
end = omp_get_wtime()
print('parallel block took ', end - start, ' seconds to finish')
```