OpenMP for Python User Guide

Caleb Huck

June 3rd, 2021 ********** change

1. DOWNLOADING/INSTALLING

We have tried to make the install process of our software as easy and straight-forward as possible. The software can be installed in any directory. The only requirement for it to work is a single environment variable and a directory that must be included in the PATH environment variable. Below is the complete list of steps needed to get up and running on Windows, Mac, or Linux:

1. Make sure that Java 8 or later is installed on the system. This can be checked by running the command `java -version` in the terminal
2. Open a terminal in the directory where you want to download the software (this can be any directory) and run the command:
   `git clone https://github.com/calebhuck/OpenMPy.git .`
3. Create an environment variable called JYTHON_HOME and set to value to be the top level directory of the file structure you just downloaded (the directory where jython-dev.jar resides)
4. Add the $JYTHON_HOME/bin/ folder to your current PATH environment variable (this is where the launcher script for Jython resides)
5. Open a new terminal and run the command `jython --version`. If everything was done correctly, the output should be `Jython 2.7.3a1-DEV`.

2. SYNTAX AND USAGE

This section covers each major OpenMP construct supported, along with the clauses that can be used in conjunction. 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 always 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.

2.1. *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)
```

2.2. *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')
```

2.3. *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)
```

2.4. *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. An excellent example of this is the implicit barrier at the end of a for directive. By default, all of the threads will wait for the others to finish their iterations before continuing with the program.

```
from omp import *

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

2.5. *Critical*

The critical directive protects the critical section of a parallel block of code. A critical section is any code that could result in race conditions 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
    for i in range(10):
        #omp critical
            sum += 1
```

2.6. *Reduction*

The reduction clause is a convenient method for assigning each thread a private variable, and then performing some 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), ^ (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
```

2.7. *Master/Single*

Currently, Master and Single have the same effect. These directives ensure that only the thread with id 0 will execute the subsequent 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.

```
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')
```

3. RUNTIME API

All runtime functions exist in the omp module, which must be imported to use them. This can be done with "`from omp import *`". The directory where this module exists is added to the python path automatically, so the user doesn't need to worry about where it is located. Currently our runtime API supports two (add omp_get_w_time()) 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.

1. `omp_get_thread_num()` - returns integer thread id
2. `omp_get_num_threads()` - returns integer representing the number of threads active in the current scope

```
from omp import *

#omp parallel num_threads(5)
    print('printed by thread ', omp_get_thread_num(), ' of ', \
   omp_get_num_threads())
```