Administrivia

- HW1 peer-review assignments out soon.
- HW2 out next week.
- Get Cython installed & working for HW2.
  - Test repo posted - please give feedback!
- RC / Odyssey accounts?
- No class Monday.
- Reading on Piazza for Wednesday.
OpenMP on OSX

- brew update
- brew install clang-omp
OpenMP / Cython
Why OpenMP?

- C-level (fast) native threads and locks
- Syntax-level support for parallel constructs.
  - #pragmas in C
- Simpler than library-based threading (e.g., pthreads)
OpenMP model

Fork-join model.

A group of threads is created, does some work. At the end of the parallel region, “join” to single threaded.
Cython support

Parallel Block:
with nogil, parallel():
    work_work_work_work()

Parallel Range:
for x in prange(N):
    work_work_work_work()
with nogil, parallel(num_threads=100):
    # Everything here gets run 100 times,
    # in 100 independent threads.
    work_work_work_work()
Local variables

Variables assigned to in the parallel block become thread-local.

cdef int val

with nogil, parallel(num_threads=100):

    val = threadid()
    # val is private to the current thread.
Local variables

WARNING:
this only works for C-typed variables

with nogil, parallel(num_threads=100):

    # This dies with a compiler error.
    with gil:
        val = threadid()
Using Python Functions

We can grab the GIL to get access to python functions.

```python
with nogil, parallel(num_threads=100):
    # grab the gil to call print()
    with gil:
        print("Hello from {}".format(threadid()))
```
Parallel Loops

for idx in prange(100, nogil=True, num_threads=10):
    val = threadid()
    with gil:
        print("Idx {} thread {}").format(idx, val)
prange() and variables

- Assigned variables become “lastprivate” in loops.
  - Private to thread
  - At end of loop, equal to last iteration.

```python
for idx in prange(100, nogil=True, num_threads=10):
    val = threadid()
    with gil:
        print("Idx {} thread {}").format(idx, val))
print("Idx {} thread {}").format(idx, val))
```
prange() and variables

- In-place operators convert to parallel reductions

```python
cdef int val = 0

for idx in prange(100, nogil=True, num_threads=10):
    val += idx

print("Sum {}".format(val))```
Scheduling Loops

Can change how indices are assigned to threads using the `schedule` keyword.

Options: static, dynamic, guided, runtime.

```python
for idx in prange(100, nogil=True, num_threads=10, schedule=dynamic):
    val = threadid()
    with gil:
        print("Idx {} thread {}".format(idx, val))
```
Scheduling Loops

- **Static** = Preassigned equal size contiguous blocks
- **Dynamic** = On-demand chunks
- **Guided** = Dynamic with chunks getting smaller
- **Runtime** = One of the options above, but chosen by environment variable (easy testing).
Nest parallel() and prange()

- Nesting a prange() inside a parallel()
  - prange() still a work-sharing loop.
  - num_threads determined by outer parallel()
- prange() inside prange()
  - inner prange() becomes a regular loop.
  - To get true parallelism over two loops, combine the indices.
Barriers

• At the end of every `parallel()` block and `prange()`, there is an implicit barrier.

• All threads must reach the barrier before computation continues.
OpenMP primitives

- Available via "cimport openmp"
- But not very well documented.
- Listed in openmp.pxd (on github)
Locks

from openmp cimport *

cdef omp_lock_t lock

omp_init_lock(&lock)

with nogil, parallel(num_threads=100):

    omp_set_lock(&lock)
    #
    # locked region
    #
    omp_unset_lock(&lock)

omp_destroy_lock(& lock)
Next time

• Reading discussion:  
  “Scalability! But at what COST?”

• HW1 Problem 5 (connected components):
  • I’ll publish single-threaded code.
  • Anyone can submit new code.
  • I’ll post a leaderboard.