Programming with OpenMP

Mark Howison

User Services & Support
This talk will...

- Focus on the **basics** of parallel programming
  - Will not inundate you with the details of memory hierarchies, hardware architectures, network topology, etc.
  - Advanced topics may be the impetus for future workshops

- Use examples found on Oscar in:
  - `/users/mhowison/OMP`
  - (You can copy these folders to your home directory)

- Assume that you have some proficiency with:
  - Linux command line and a text editor
  - C
OpenMP Basics

- A framework for *threaded* parallelism:
  - Multiple threads run concurrently
    - Usual mapping is 1 thread → 1 core
  - *Shared memory* is accessible to all threads
    - Danger: could overwrite another threads memory!
    - Threads can also have their own private variables
- OpenMP is implemented in all modern compilers
  - gcc -fopenmp -o executable source.c
- Add *directives* to your code to give the compiler information about parallel execution
  - Different from other libraries that rely primarily on function calls
  - *Explicit* programming model: you get full control over thread creation
Directive Syntax

- Directives look like comments surrounding parallel regions of code:

```
#pragma omp <function> <arguments>
{
  ...
}
```
Parallel directive

Creates a set of threads, executes a block of code in parallel across all threads, then joins the threads.

#pragma omp parallel [clause ...]

Clauses:

if (scalar_expression)
private (list)
shared (list)
default (shared | none | private)
firstprivate (list)
reduction (operator: list)
copyin (list)
um_threads (integer-expression)
Number of threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1) Evaluation of the if clause
  2) Setting of the num_threads clause
  3) Use of the omp_set_num_threads() library function
  4) Setting of the OMP_NUM_THREADS environment variable
  5) Implementation default—usually the number of cores

- Threads are numbered from 0 (master thread) to N-1

- Exercise: try running the “parallel” example with different numbers of threads.
Barrier directive

Each thread waits at the barrier until *all* threads have reached it.

```c
#pragma omp barrier
```

(There is an *implicit* barrier at the end of every parallel directive)
Loop directive

Distributes the iterations of a loop over multiple threads.

```
#pragma omp for [clause ...]
```

Clauses:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(operator: list)`
- `schedule(kind[, chunk_size])`
- `collapse(n)`
- `ordered`
- `nowait`

“loop” example
Loop short-hand

Parallel loops are so common that they have a short-hand. Instead of creating a loop directive inside of a parallel directive, you can combine them into one directive:

```
#pragma omp parallel for [clause ...]
for (...) {
    ...
}
```
Reduction operators

A “reduction” is the process of applying an operator to all values of an array to produce a single value. The reduction argument guarantees safe calculations across threads that prevent race conditions.

Operators

+, *, -, &, ^, |, &&, ||

“reduction” example
Single directive

Only a single thread (the first to reach it) will perform this block of code, while the other threads wait.

```
#pragma omp single [clause ...]
```

Arguments

- `private(list)`
- `firstprivate(list)`
- `copyprivate(list)`
- `nowait`
More directives...

**critical** – block is executed serially by each thread

**sections** – creates a list of tasks that are executed concurrently by different threads

**workshare** – divides a block of code into discrete units of work that are distributed among available threads
Exercises

- Write a program that finds the maximum (or minimum) value across threads
  - Hint: start with sharing memory between threads
  - Harder questions:
    - How scalable is your solution?
    - Can you improve it?
    - What is the upper bound on efficiency for a reduction operation like this?

- Write a linear algebra routine: matrix-vector multiply, matrix-matrix multiply, etc.
Additional resources

- NERSC Tutorial (Fortran only): http://www.nersc.gov/nusers/help/tutorials/openmp
- LLNL Tutorial (Fortran and C/C++): https://computing.llnl.gov/tutorials/openMP