

# Programming with OpenMP and MPI

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
  - /users/mhowison/MPI
  - (You can copy these folders to your home directory)
- ▶ Assume that you have some proficiency with:
  - Linux command line and a text editor
  - C or Fortran90

# Other upcoming workshops

- ▶ “Profiling and Performance Analysis”
  - Tuesday, March 15, 2-3pm, Saloman 003
- ▶ “Parallel I/O Libraries and Techniques”
  - Monday, April 4, 1-2pm, Petteruti Lounge
- ▶ We will probably repeat this semester's workshop schedule every semester
- ▶ We may also plan a multi-day “boot camp” in the summer, covering the same topics
- ▶ Please let us know if you have specific requests for other topics!

# Quick compiling cheatsheet

## ▶ OpenMP

- `gcc -fopenmp -o executable source.c`
- `gfortran -fopenmp -o executable source.f90`

## ▶ MPI

- `mpicc -o executable source.c`
- `mpif90 -o executable source.f90`
- Running an MPI program:
  - `mpirun -np [number of tasks] executable`

# 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
  - Pass a flag to the compiler to enable it (-fopenmp for GNU)
- ▶ 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:

## C/C++

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

## Fortran 90

```
!$omp <function> <arguments>  
  ...  
!$omp end <function>
```

# 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 ...]
```

```
!$omp parallel [clause ...]
```

Arguments:

```
if (scalar_expression)
private (list)
shared (list)
default (shared | none | private)
firstprivate (list)
reduction (operator: list)
copyin (list)
num_threads (integer-expression)
```

“parallel” example

# 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.

```
#pragma omp barrier
```

```
$_omp barrier
```

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

“barrier” example

# Loop directive

Distributes the iterations of a loop over multiple threads.

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

```
!$omp do [clause ...]
```

Arguments:

```
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 (...) {
```

```
    ...
```

```
}
```

```
!$omp parallel do [clause ...]  
do ...
```

```
    ...
```

```
enddo
```

# 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

### C/C++

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

### Fortran

`+`, `*`, `-`, `.and.`, `.or.`, `.eqv.`, `.neqv.`  
`max`, `min`, `iand`, `ior`, `ieor`

“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 ...]
```

```
!$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
  - Without using the max or min reduction operator in Fortran
  - 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?

# Additional resources

---

- ▶ OpenMP Specification:  
<http://openmp.org/wp/openmp-specifications/>
- ▶ NERSC Tutorial (Fortran only):  
<http://www.nersc.gov/nusers/help/tutorials/openmp>
- ▶ LLNL Tutorial (Fortran and C/C++):  
<https://computing.llnl.gov/tutorials/openMP>

# MPI Basics

- ▶ A framework for *distributed-memory* parallelism:
  - Multiple *tasks* run concurrently across separate nodes
  - Each task has its own private memory
    - Memory is shared by *passing messages* among nodes
    - Messaging requires a high-performance interconnect
- ▶ MPI is implemented as a library with wrappers for compiling:
  - mpicc (C), mpic++ (C++), mpif90 (Fortran 90)
- ▶ Make calls to the MPI library as you would with other APIs
  - An MPI program starts with `MPI_Init()` and ends with `MPI_Finalize()`
- ▶ Run an MPI program using the `mpirun` wrapper on a cluster of nodes (or sometimes on a single node for testing/debugging)

“hello” example

# Communicators

- ▶ MPI tasks can be grouped into sets called “communicators”
- ▶ All available MPI tasks are automatically placed in the `MPI_COMM_WORLD` communicator
- ▶ Can synchronize communicators with barriers:
  - `MPI_Barrier(communicator)`
- ▶ Advanced topic: constructing more complicated hierarchies, e.g. to mirror underlying hardware

# Point-to-point

- ▶ Move data from one MPI task to another (similar to a TCP connection)
- ▶ In regular MPI (no fancy constructs), this is always *two-sided*
  - The sender has to call `MPI_Send`
  - The receiver has to call `MPI_Recv`
  - These calls are *blocking*: your program waits until the transaction is complete before continuing
- ▶ Advanced topics:
  - Non-blocking send/receive for asynchronous communication
  - One-sided messaging to decrease latency

“pingpong” example

# Data types

- ▶ For portability, you must tell MPI what kind of data you are transmitting
- ▶ Most basic types are predefined (e.g. MPI\_DOUBLE, MPI\_INT, MPI\_CHAR, etc.)
- ▶ Advanced topic: “derived” data types
  - You can aggregate basic types into, for example, vectors or arrays
  - You can create custom types for structs

# “Collective” calls

- ▶ ([https://computing.llnl.gov/tutorials/mpi/#Collective\\_Communication\\_Routines](https://computing.llnl.gov/tutorials/mpi/#Collective_Communication_Routines))
- ▶ Some common messaging paradigms are already implemented/optimized:
  - Broadcasting a message from one task to all tasks
    - MPI\_Bcast
  - Computing (or “reducing”) a value across all tasks
    - MPI\_Reduce with MPI\_SUM, MPI\_MIN, MPI\_MAX, etc.
  - From one task, sending a unique message to every other task
    - MPI\_Scatter
  - From each task, sending a unique message to one task
    - MPI\_Gather
  - Sending a unique message from each task to every other task
    - MPI\_Alltoall

“coin” example

# Exercises

---

- ▶ Write a program that prints out a message from each rank *in order*
  - Using a barrier as in the OpenMP “barrier” example
  - OR using point-to-point messages to send a “token” message through the ranks

# Additional resources

---

- ▶ MPI Specification:  
<http://www.mpi-forum.org/docs/docs.html>
- ▶ NERSC Tutorial (Fortran and C/C++):  
<http://www.nersc.gov/nusers/help/tutorials/mpi/intro/>
- ▶ LLNL Tutorial (Fortran and C/C++):  
<https://computing.llnl.gov/tutorials/mpi/>