Introduction to OpenMP

Research Computing
UNIVERSITY OF COLORADO **BOULDER**

Be Boulder.

Introduction to OpenMP

Presenter:	Daniel Trahan
Contact:	Daniel.Trahan@colorado.edu
Website:	www.colorado.edu/rc

Sign in:	http://tinyurl.com/curc-names
Slides:	https://github.com/ResearchComputing/OpenMP-Spring-2020

Overview

- What is OpenMP
- Compiling and Running applications with OpenMP
- Compiler Directives and Parallel Regions
- Memory Management
- Synchronization and Loop Scheduling

Why Parallel?

- Modern CPUs are capped in clock speed…
- Instead of increasing clock speed, add more transistors!
- Multiple processors on a single chip.
- Parallelism required to utilize full power!

What is OpenMP

- OpenMP is a parallelization programming library for C, C++, and Fortran.
    - Built into GNU and Intel Compilers
    - Single Processor/Shared Memory based parallelization strategy
- OpenMP code is follows a strategy of parallel ‘blocks’ of code
    - Serial blocks of code run on one core and parallel blocks of code run on multiple cores.
    - Blocks are specified with OpenMP compiler directives.

Why OpenMP

- Very simple to implement parallelization to an existing block of code.
- No Explicit Communication among processes
- Fork/Join Parallelism
  - Process begins in serial executing on a single core.
  - When reaching a parallel region, processes are forked.
  - At the end of the parallel region, processes are rejoined.

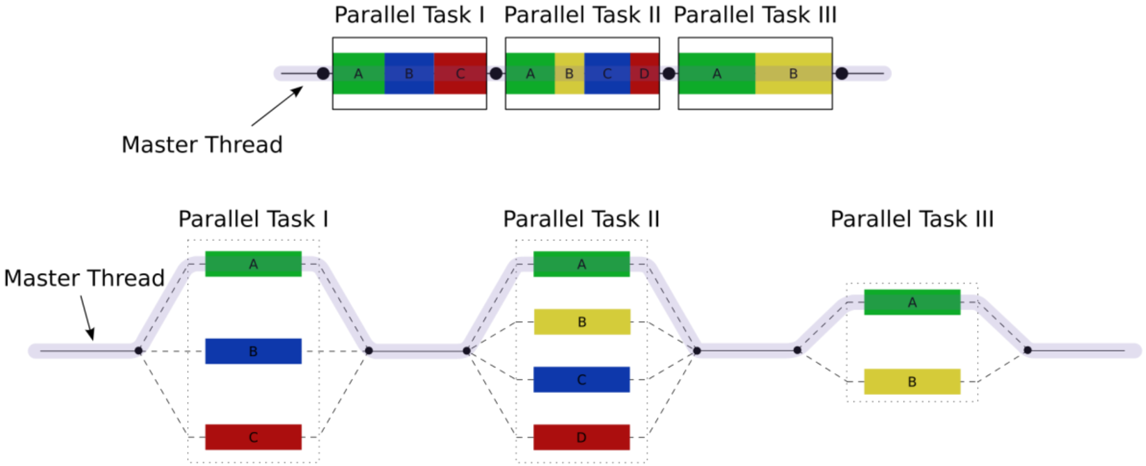


Image from: https://en.wikipedia.org/wiki/Fork%E2%80%93join_model

Getting started with OpenMP

- Declaring a Parallel Region:
  - Special comments in code are read by the compiler and parallelize the application.
  - ‘parallel’ directive
  - Any code within the code block is executed by all cores on your system.
    - Can be modified with environment variable.

```
export OMP_NUM_THREADS=<nthreads>
```

C/C++

```
#pragma omp parallel
{
    #parallelized code…
}
```

Fortran

```
use omp_lib
$omp parallel
    #parallelized code…
$omp end parallel
```

Compiling and Running with OpenMP

- Compiling a parallel application requires no special libraries or complex commands.
- Add a compiler flag!
  - With gcc: `-fopenmp`
  - With intel: `-qopenmp`

C/C++

```
gcc -fopenmp sample.c –o sample.exe
#replace gcc w/ g++ for c++
icc –qopenmp sample.c –o sample.exe
#replace icc w/ icpc for c++
```

Fortran

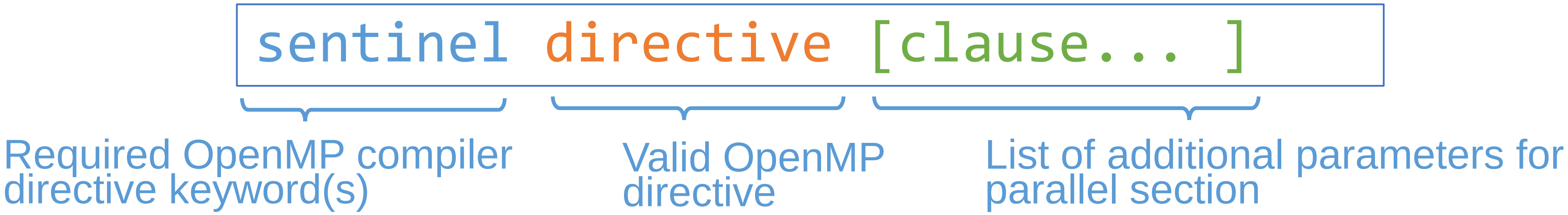
```
gfortran -fopenmp sample.f90 –o sample.exe
ifort –qopenmp sample.f90 –o sample.exe
```

Example 1: Parallel ‘Hello World’

- Now that we know how to set up some basic parallelization, lets write a simple parallel hello world.
  - I will be using C for this example. The set up is pretty much the same for Fortran.
  - See `hello-omp.c` and `hello-omp.f90`

Compiler Directives (1)

- OpenMP understands a variety of other compiler directives with various clauses:
- Anatomy of Compiler Directives



```
sentinel directive [clause... ]
```

Required OpenMP compiler directive keyword(s)

Valid OpenMP directive

List of additional parameters for parallel section

- Ex: C/C++

```
#pragma omp for reduce(+:total)
```

Fortran

```
$!omp parallel shared(var1, var2)
```

Compiler Directives (2)

- A lot of different OpenMP directives!
- Can be broken into 3 different categories
  - Parallelization
  - Work-Sharing
  - Synchronization
- We will be talking about just 4 today:
  - ‘Parallel’, ‘For’, ‘Section’, and ‘Barrier’

OpenMP Library Methods

- OpenMP includes several methods that can be utilized.
    - `omp_get_num_threads()` – Returns total number of threads
    - `omp_set_num_threads()` – Sets the number of available threads
    - `omp_get_thread_num()` – Returns the individual thread number
- Useful to manage parallelism and keep track of threads.
- Tons of more methods!

Directive: Parallel

- As shown previously, the OpenMP `parallel` directive is used to run a block of code by all available cores instead of a single core.
- Syntax:
- Must be invoked at the start of all parallel regions.
- Application will return to running in serial after completion.

C/C++

```
#pragma omp parallel
{ # parallel code }
```

Fortran

```
$!omp parallel
$ parallel code
$!omp end parallel
```

Implicit Memory in OpenMP

- OpenMP has specific implicit rules that it utilizes to handle memory.
    - Any variables created outside the parallel scope are implicitly shared.
    - Any variables created inside the parallel scope are implicitly private.
    - Loop iteration variables are always private.
- Useful to know but can be confusing…

Explicit Memory in OpenMP (1)

- Memory in OpenMP are handled by compiler directive clauses.
  - `shared` variables have the same memory address on every thread.
  - `private` variables have different memory addresses for every thread.

C/C++

```
#pragma omp parallel shared(pub1, pub2, pub3) private(prv1, prv2, prv3)
{ # parallel code }
```

Fortran

```
$!omp parallel shared(pub1, pub2, pub3) private(prv1, prv2, prv3)
$ parallel code
$!omp end parallel
```

Explicit Memory in OpenMP (2)

- Can force explicit programming on all variables declared out of a scope.
  - `default` clause allows users to change behavior of all variables outside of a scope.
  - Accepts `private`, `shared`, or `none` as parameters.
    - `none` requires users manually classify all variables as shared or private.

C/C++

```
#pragma omp parallel default(none) shared(...) private(...)
```

Fortran

```
$!omp parallel default(none) shared(...) private(...)
```

Directive: For/Do

- The OpenMP `for`/`do` directive is a work sharing directive that divides a for loop among various workers.
- Syntax:

C/C++

```
#pragma omp for
# for loop
```

Fortran

```
$!omp do
$ do loop
$!omp end do
```

- Directive will split the next loop in the application among available workers
- Process is not threadsafe by default…

Example 2:

- Let’s parallelize a simple summation loop from all the numbers between 1 and 10000
- Prints out the total sum of the loop.
- Did our application properly compute the sum in parallel?
- Why or Why not?

Thread Safety

- Thread Safety is the assurance that a thread will execute without any unintentional side effects.
- Last example was overwriting a shared sum in a loop.
- Since the ‘for’ directive isn’t thread safe, how can we ensure data integrity of our summation?

Reduction Clause

- With for/do loops that are all computing to a single reducible sum, we can use the `reduction` clause to avoid thread unsafe operations.
- Syntax:

C/C++

```
#pragma omp for reduction(+:reduce-var)
# for loop
```

Fortran

```
$!omp do reduction(+:reduce-var)
$ do loop
$!omp end do
```

- Requires a reduction operator and a reduction variable.
- Reduction variable is locally store and combined at the end.

Directive: Barrier

- The OpenMP `barrier` directive is a synchronization directive that halts execution of threads until all threads reach the barrier line of code.
- Syntax:
- Useful for sections code that must be synchronized before execution.

C/C++

```
#pragma omp barrier
```

Fortran

```
$!omp barrier
```

Directive: Sections and Section (1)

- The OpenMP ‘Sections’ and ‘Section’ directives can be used to separate threads to individual tasks.
- The ‘Sections’ directive designates the body of directives that will be divided into individual chucks run by threads.
- The ‘Section’ directive designates actual code to be run by each thread.
- Each section is run by a single thread regardless of how many workers are available.

Directive: Sections and Section (2)

- Syntax:

C/C++

```
#pragma omp parallel sections
{
    #pragma omp section {
        # Thread 1 code
    }
    #pragma omp section {
        # Thread 2 code
    }
}
```

Fortran

```
$!omp parallel sections
$!omp section
$ Thread 1 code
$!omp section
$ Thread 2 code
$!omp end parallel sections
```

Example 3:

- Use the `sections` and `section` directive to distribute 3 loops to 3 threads that calculate the sum from 1 to 10, 1 to 100, and 1 to 1000.
- Use the reduction clause to reduce all 3 sums to one final sum and print out the result.

Additional Resources

- OpenMP Website: https://www.openmp.org/
- Jaka’s Corner (OpenMP Ex): http://jakascorner.com/blog/
- Laurence Livermore: https://computing.llnl.gov/tutorials/openMP/
- GNU reference: https://gcc.gnu.org/wiki/openmp
- Intel reference: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-openmp-support

Questions and Thank You!

Contact: Daniel.Trahan@colorado.edu

Website: www.colorado.edu/rc

Sign in: http://tinyurl.com/curc-names

Slides: https://github.com/ResearchComputing/OpenMP-Spring-2020

Survey: http://tinyurl.com/curc-survey18