

COMP4300 - Course Update

- **Assignment 1**
 - Marking nearly complete
- **Assignment 2**
 - Released on 24 April
 - Due 26/05/2025, 11:55PM
 - Start early

1

TEQSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



2

OpenMP: Part I

TEQSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



OpenMP Reference Material

- **Using OpenMP – The Next Step**, R. van der Pas, E. Stotzer, and C. Terboven, Chapter 1
- <http://www.openmp.org/>
- *Introduction to High Performance Computing for Scientists and Engineers*, Hager and Wellein, Chapter 6 & 7
- *High Performance Computing*, Dowd and Severance, Chapter 11
- *Introduction to Parallel Computing*, 2nd Ed, A. Grama, A. Gupta, G. Karypis, V. Kumar
- *Parallel Programming in OpenMP*, R. Chandra, L.Dagum, D.Kohr, D.Maydan. J.McDonald, R.Menon

3

TEQSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



History of Concurrent Programming

Fork

- The concept of "fork" originated in Unix operating systems.
- Forking was a fundamental mechanism for multitasking in Unix.
- The fork() system call is used to create a new process by duplicating the existing process.

Threads

- As computing needs grew, the limitations of process-based concurrency became apparent, particularly the overhead associated with creating and managing processes.
- Threads enabled finer-grained parallelism and improved performance for applications requiring concurrent execution of tasks.

OpenMP

- OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.
- OpenMP uses compiler directives, library routines, and environment variables to control parallelism.

4

TEQSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



Shared Memory Parallel Programming

- Explicit thread programming is messy
 - low-level primitives
 - complex data scoping and initialization not easy to port
 - significant amount of boiler-plate code
 - used by system programmers, but application programmers have OpenMP!
- Many application codes can be supported by higher level constructs with the same performance
 - led to proprietary directive based approaches of Cray, SGI, Sun, etc.
- OpenMP is an API for shared memory parallel programming targeting Fortran, C and C++
 - standardizes the form of the proprietary directives
 - avoids the need for explicitly setting up mutexes, condition variables, data scope, and a good part of explicit initialization
 - When you compile an OpenMP program, the compiler often translates OpenMP directives into pthreads calls to manage parallel execution

OpenMP

- Specifications maintained by OpenMP Architecture Review Board (ARB)
 - members include AMD, Intel, Fujitsu, IBM, NVIDIA
- Versions 1.0 (Fortran '97, C '98), 1.1 and 2.0 (Fortran '00, C/C++ '02), 2.5 (unified Fortran and C, 2005), 3.0 (2008), 3.1 (2011), 4.0 (2013), 4.5 (2015), 5.0 (2018), 6.0 (2024)
- Comprises compiler directives, library routines and environment variables
 - C directives (case sensitive)

```
#pragma omp directive_name [clause-list]
```
 - library calls begin with `omp_`

```
void omp_set_num_threads(nthreads)
```
 - environment variables begin with `OMP_`

```
export OMP_NUM_THREADS=4
```
- OpenMP requires compiler support
 - set `-fopenmp` (gcc) or `-fopenmp` (icc) compiler flags

5

TESSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



6

TESSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



The Parallel Directive

- OpenMP uses a fork/join model, i.e. programs execute serially until they encounter a parallel directive:
 - this creates a group of threads
 - the number of threads is dependent on an environment variable or is set via function call
 - the main thread becomes master with thread id 0

```
#pragma omp parallel [ clause - list ]
    /* structured block */
```

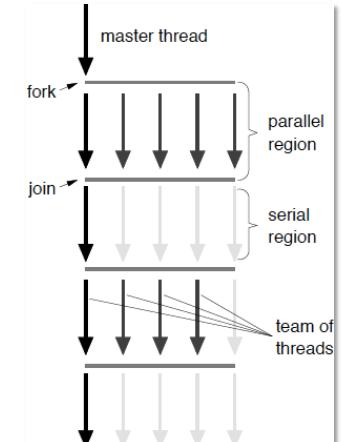
- Each thread executes the *structured block*
- In C/C++ this is a brace-enclosed (`{ code }`) sequence of statements and declarations.

TESSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



The OpenMP Execution Model

- An OpenMP starts in serial mode with one thread executing the serial code (*master thread*)
- At the beginning of the parallel region additional threads are created (*forking* from the master) by the runtime system forming a *thread team*
- All threads are active in the parallel regions, executing the program in parallel.
- At the end of the parallel region threads are *joined*, with only the master continuing through the serial portion.
- This is called the *fork-join model*.



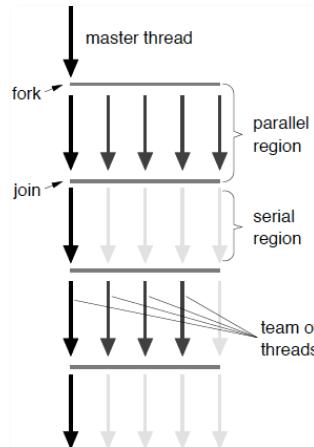
7

TESSA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



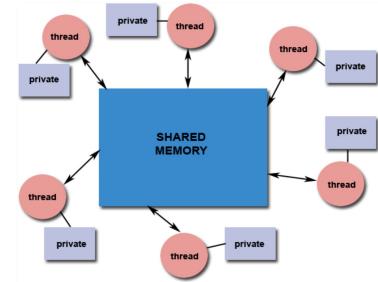
The OpenMP Execution Model

- The number of threads execution in the parallel region can be set through the `OMP_NUM_THREADS` environment variable.
- If the number of threads need to be more dynamic, the `omp_set_num_threads` may be used *prior* to a parallel region.
- An alternative is to use the `num_threads<nt>` clause on the `parallel` directive.
- Because of the join operation, the end of the parallel region is an *implicit synchronization point* (barrier).



The OpenMP Memory Model

- Underlying the OpenMP standard is the `pthreads` memory model, but the distinction between *private* and *shared* is clearer.
- Whether a variable is private or shared as well as their initialization can be defined by *default rules*
- These can also be explicitly controlled through appropriate clauses on a construct.
- It is recommended to not rely on the default rules and explicitly label or "scope" variables.



9

10

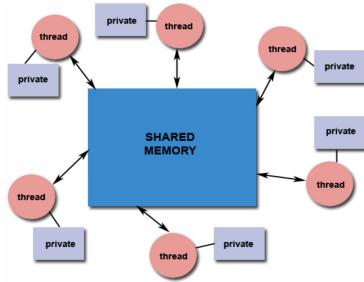


TEACH PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

The OpenMP Memory Model

Private and Shared Variables

- Private variables can be accessed only by the owning thread, no other thread may interfere.
- Threads may even use the same name for a private variable without the risk of any conflict.
- Each thread has read and write access to the same shared variable, that is only one instance of a given shared variable exists.
- Global or static variables are shared by default.



11

12

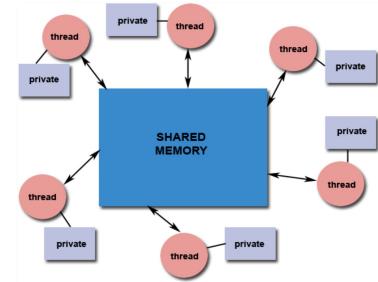


TEACH PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

The OpenMP Memory Model

Default Rules

- Variables declared outside the parallel region are shared by default.
- Global and static variables are also shared by default.
- Variables declared inside the parallel region are private by default.



Data Sharing Clauses

- The `private` (`shared`) clause may be used to make a variable `thread private` (`shared`).
- One needs to be careful with initialization.
- The `firstprivate` clause guarantees that all threads have a pre-initialized copy of a variable
- The `default` clause is used to give a default data sharing attribute (`none`, `shared`, `private`) to all variables.
- When `default(none)` is used, the programmer is forced to specify data-sharing attributes for all variables in the construct.

```
int x = 5; int y = 20;
int z[ NUM_THREADS ] = {0};
#pragma omp parallel default (none) shared (z)
    private (x) firstprivate (y)
{
    x = 10; // x is undefined on entry, but now set
             to 10
    z[omp_get_thread_num ()] = omp_get_thread_num ();
    int w = x + y+ z[omp_get_thread_num ()]; // y pre
                                              - initialized to a value of 20
    ...
    y = 30 // firstprivate var may be modified
}
```

13

TESSA PROVIDER ID: PRV12322 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00123C



OpenMP: The Work-Sharing Directives

- Used to distribute work among threads in a team.
- They specify the way the work has to be distributed among threads.
- Work-sharing directive must bind to a parallel region, otherwise is simply ignored.
- Work-sharing constructs do not have a barrier at entry.
- By default, a barrier is implemented at the end of the work-sharing region. The programmer can suppress the barrier with use of the `nowait` clause.

Functionality	Syntax in C/C++	Syntax in Fortran
Distribute iterations over the threads	<code>#pragma omp for</code>	<code>!\$omp do</code>
Distribute independent work units	<code>#pragma omp sections</code>	<code>!\$omp sections</code>
Only one thread executes the code block	<code>#pragma omp single</code>	<code>!\$omp single</code>
Parallelize array-syntax		<code>!\$omp workshare</code>

15

TESSA PROVIDER ID: PRV12322 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00123C



Data Sharing Attributes

Default Rules

- Variables declared outside the parallel region are **shared** by default.
- Global and static variables are also **shared** by default.
- Variables declared inside the parallel region are **private** by default.

```
int g = 0; // g is shared
int main (){
    int i = 0; // i is shared
    static int a = 7; // a is shared
    #pragma omp parallel
    {
        int b = a + i + g; // b is private
        ...
    }
    return 0;
}
```

14

TESSA PROVIDER ID: PRV12322 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00123C



The `for` Work-Sharing Directives

Used in conjunction with `parallel` directive to partition the `for` loop immediately afterwards

- The loop index (`i`) is made private by default
- Only two directives plus the sequential code (code is easy to read/maintain)
- Limited to loops where number of iterations can be counted

```
# pragma omp parallel shared ( n )
{
    # pragma omp for
    for ( i = 0; i < n; i ++ ) {
        printf ( " Thread % d , iteration % d\ n",
                 omp_get_thread_num , i );
    }
} /* End of parallel region */
```

There is implicit synchronization at the end of the loop

- Can add a `nowait` clause to prevent synchronization

16

TESSA PROVIDER ID: PRV12322 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00123C



The `for` Work-Sharing Directives

- The order in which threads execute is not predictable (OS scheduled).
- The way to map iterations to threads can be specified by the programmer (see later `schedule` clause).
- If the programmer does not specify the mapping between threads and iterations, the compiler decides which strategy to use.

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

17



TEQA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

The `single` Work-Sharing Directives

- This directive specifies that only one thread must execute the code in the structured block following it.
- It does not state which thread should execute the code.

```
# pragma omp parallel shared ( a, b, n )
{
    # pragma omp single
    {
        a = 10;
    } /* A barrier is automatically inserted here */
    # pragma omp for
    for ( i = 0; i < n; ++i)
    {
        b[ i ] = a;
    } /* Another barrier is automatically inserted here */
}
```

19



TEQA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

The `sections` Work-Sharing Directives

- Consider partitioning of fixed number of tasks across threads
- Each section must be a structured block that is independent from the other sections.
- Separate threads will run `taskA` and `taskB`
- Illegal to branch in or out of section blocks

Note:

- Much less common than `for` loop partitioning
- Explicit programming naturally limits number of threads (scalability)
- Potential load imbalance

```
# pragma omp parallel
{
    # pragma omp sections
    {
        # pragma omp section
        task A ()

        # pragma omp section
        task B ()

    } /* End of sections block */

} /* End of parallel region */
```

18



TEQA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Combined parallel Work-Sharing Directives

- When there is only one work-sharing directive it can be combined with the parallel one to improve readability.
- Only clauses that are allowed by both the parallel and the specific work-share directive are allowed, otherwise the code is illegal.
- The compiler may optimize code further (e.g. remove redundant barriers).

Full version	Combined construct
<pre># pragma omp parallel { # pragma omp for for (...) }</pre>	<pre># pragma omp parallel for for (...)</pre>
<pre># pragma omp parallel { # pragma omp sections { # pragma omp section section A ... } }</pre>	<pre># pragma omp parallel sections { # pragma omp section section A ... }</pre>

TEQA PROVIDER ID: PRV12302 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

20



Demo

OpenMP – Pi calculation

- Monte Carlo Pi calculation code
- Get the number of threads from the environment variable OMP_NUM_THREADS
- Detailed performance information:
 - Total Execution Time
 - Initialization Time
 - Computation Time
 - Finalization Time
- Each timing section uses `omp_get_wtime()` which is thread-safe and provides high-resolution timing information appropriate for OpenMP applications
- At the end, the code calculates and displays:
 - Speedup = Sequential Time / Parallel Time
 - Parallel Efficiency = Speedup / Number of Threads

21

TEQSA PROVIDER ID: PRV1202 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C



Demo

OpenMP – Pi calculation

- Monte Carlo Pi calculation code scales well using OpenMP
- Adding more threads than processors delivers better performance for this problem.
 - Thread switching overhead is low.
- Code changes are minimal: -

```
#pragma omp parallel for firstprivate(x, y, z) reduction(+:count)
```

22

TEQSA PROVIDER ID: PRV1202 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

