Practical stuff!

Ways of actually get stuff done in HPC:

- Message Passing (send, receive, broadcast, ...)  ✔  MPI
- Shared memory (load, store, lock, unlock)
- Transparent (compiler works magic)  Hahahahaha!  ✔  OpenMP
- Directive-based (compiler needs help)
- Task farming (scientific term for large transaction processing)  MapReduce/Hadoop etc
Confession!

A lot of blatant plagiarism of BLAINE BARNEY’s tutorial at LLNL!

https://computing.llnl.gov/tutorials/openMP
OpenMP - History

- Another specification: openmp.org
- “Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industries, government and academia”
- Implementation of the shared memory programming model
- Is an Application Program Interface (API) that may be used to explicitly direct multi-threaded shared memory parallelism
- API = 3 parts:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Evolution:
  - Early 90’s, vendors made directive-driven Fortran extensions for shared mem machines
  - 1994: standard – ANSI X3H% - never adopted as distributed machines emerged
  - Newer shared mem machines came along
  - API specs separate releases for C and Fortran until combined in 2005
  - Latest: OpenMP 4.0 July 2013
OpenMP - History

• OpenMP is NOT:
  – Meant for distributed memory parallel systems (by itself)
  – Implemented identically by all vendors
  – Guaranteed to make most efficient use of shared memory
  – Required to check for data dependencies, deadlocks etc (programmer!)

• OpenMP is:
  – A standardisation built and endorsed by many
  – “lean and mean” and simple (although trend away 😞)
  – portable
  – C/C++, Fortran

😊 OpenMP requires a compiler that supports OpenMP
😊 Amdahl’s law! Lot of the code is still sequential
😊 Easy to build up code incrementally by adding directives
😊 Directives are like comments so sequential code still runs
😊 Code is lightweight
😊 Data decomposition is largely invisible/automatic
OpenMP

• Designed for multi-processor/core shared memory architectures, UMA or NUMA

• Parallelism by use of THREADS
  • Threads are smallest units of processing
  • Many threads share resources of a single process
  • Typically, number of threads matches number of cores, but not necessarily (can overlap some work with virtual threads)
OpenMP

Programming model:
- Explicit not automatic (despite compiler-based; not Holy Grail!) => programmer has complete control over parallelism
- Easy: take sequential program and insert compiler directives to parallellise
- Complex: insert subroutines to set multiple levels of parallelism with locks etc
- Fork-join model:
  - Start as single process = master thread
  - Fork: master thread creates team of parallel threads
  - Join: when team threads complete parallel work, synchronize and terminate
  - Control returns to master thread
Programming model (cont)

Notice:

- Parallelism is achieved by marking sequential codes with embedded compiler directives
- Nested parallelism is allowed
- Threads are dynamically created and destroyed
- I/O is up to the programmer!
- Cache level coherency in threads needs to be ensured by the programmer
  (FLUSH often 😊)
OpenMP: API

Programming model:
- Compiler directives (44)
- Runtime Library routines (35)
- Environment variables (13)

- Compiler directives
  - Comments in source code, interpreted by compiler if required
  - Spawn parallel region
  - Divide code work amongst threads
  - Distribute loop iterations amongst threads
  - Synchronize
  - Serial sections

```
  $OMP...
```

```
C$OMP...
```

e.g.

```
Fortran
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)

C/C++
#pragma omp parallel default(shared) private(betta,pi)
```

Optional clauses

```
```

```
```

“sentinel”:

```
! $OMP -- free format source
C $OMP, *$OMP -- fixed format source
```

Directive

```
```
OpenMP: API

Programming model:
- Compiler directives (44)
- Runtime Library routines (35)
- Environment variables (13)

- **Runtime routines**
  - Setting and querying number of threads
  - Querying thread identifiers
  - Setting/querying nested parallelism
  - etc etc
  e.g.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>INTEGER FUNCTION OMP_GET_NUM_THREADS()</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td><code>#include &lt;omp.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>int omp_get_num_threads(void)</code></td>
</tr>
</tbody>
</table>
OpenMP: API

Programming model:
- Compiler directives (44)
- Runtime Library routines (35)
- Environment variables (13)

• Environment variables
  - Set number of threads
  - Specify how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism, dynamic threads

  e.g.

<table>
<thead>
<tr>
<th>csh/tcsh</th>
<th>setenv OMP_NUM_THREADS 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh/bash</td>
<td>export OMP_NUM_THREADS=8</td>
</tr>
</tbody>
</table>
OpenMP: Typical code

**Fortran**

```fortran
PROGRAM HELLO
INTEGRER VAR1, VAR2, VAR3

Serial code

!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)

Parallel section executed by all threads

Other OpenMP directives

Run-time Library calls

All threads join master thread and disband

!$OMP END PARALLEL

Resume serial code

END
```

**C**

```c
#include <omp.h>

main () {  
  int var1, var2, var3;

  Serial code
  
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    Parallel section executed by all threads
    
    Other OpenMP directives
    
    Run-time Library calls
    
    All threads join master thread and disband
  }

  Resume serial code
  
}
```
### OpenMP

#### How to compile:

<table>
<thead>
<tr>
<th>Compiler / Platform</th>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Linux Opteron/Xeon</td>
<td>icc icpc ifort</td>
<td>-openmp</td>
</tr>
<tr>
<td>PGI Linux Opteron/Xeon</td>
<td>pgcc pgCC pgf77 pgf90</td>
<td>-mp</td>
</tr>
<tr>
<td>GNU Linux Opteron/Xeon IBM Blue Gene</td>
<td>gcc g++ g77 gfortran</td>
<td>-fopenmp</td>
</tr>
<tr>
<td>IBM Blue Gene</td>
<td>bgxlc_r, bgcc_r bgx1c_r, bgxlc++_r bgxlc89_r bgxlc99_r bgxlf_r bgxlf90_r bgxlf95_r bgxlf2003_r</td>
<td>-qsmp=omp</td>
</tr>
</tbody>
</table>

*Be sure to use a thread-safe compiler - its name ends with _r*

- Compiler Documentation:
  - PGI: [www.pgroup.com](http://www.pgroup.com)
  - GNU: [gnu.org](http://gnu.org)
  - All: See the relevant man pages and any files that might relate in /usr/local/docs
Let's just do an example: Hello World!

```fortran
PROGRAM HELLO

implicit none
INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS, &
  OMP_GET_THREAD_NUM

! Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(NTHREADS, TID)

! Obtain thread number
TID = OMP_GET_THREAD_NUM()
PRINT *, 'Hello World from thread = ', TID

! Only master thread does this
IF (TID .EQ. 0) THEN
  NTHREADS = OMP_GET_NUM_THREADS()
  PRINT *, 'Number of threads = ', NTHREADS
END IF

! All threads join master thread and disband
!$OMP END PARALLEL

END
```

**Compiler directives**

**Parallel region**

**Library routine calls**
Example

Notice you needed a bunch of things:

gfortran -f openmp <file.f90> -o file_exec
setenv OMP_NUM_THREADS <nthreads>

Mencia, Muscat, Jerez (no qsub, interactive or Unix batch only):

./file_exec

Grape (qsub):

qsub -q newest -I
./file_exec
OpenMP - Directives

**Parallel region**: A block of code that will execute on multiple threads

When a thread reaches a PARALLEL directive, thread creates team of threads and becomes master (0)

Implied barrier at END PARALLEL

If any thread terminates, all terminate

Region must be all in one routine/code file

Cannot branch (GOTO) out!

---

**Number of threads set by**

- IF (must be TRUE to create threads)
- NUM_THREADS clause
- library function OMP_NUM_THREADS
- env variable OMP_NUM_THREADS
- Default (no of procs on node)

---

**Fortran**

```fortran
!$OMP PARALLEL [clause ...]
IF (scalar_logical_expression)
PRIVATE (list)
SHARED (list)
DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
FIRSTPRIVATE (list)
REDUCTION (operator: list)
COPYIN (list)
NUM_THREADS (scalar-integer-expression)

block

!$OMP END PARALLEL
```

**C/C++**

```c
#pragma omp parallel [clause ...] newline
  if (scalar_expression)
  private (list)
  shared (list)
  default (shared | none)
  firstprivate (list)
  reduction (operator: list)
  copyin (list)
  num_threads (integer-expression)

  structured_block
```

Data scope attribute clauses like PRIVATE, SHARED dealt with later
Work sharing constructs:

The main routines for telling the parallel region how to divide the work

- **DO** – divides a DO loop amongst thread.  Data parallelism
- **SECTIONS** – divides work into discrete sections for each thread.  Functional parallelism
- **SINGLE** – forces region to execute on single thread.  Sequential.
- **WORKSHARE** – FORTRAN only!  Specific types of Fortran work can be enclosed and forked
- **TASK** – Task scheduling of code blocks

Go through these one by one …
OpenMP – Directives: DO

**Fortran**

```fortran
!$OMP DO (clause ...) 
    SCHEDULE (type [,chunk])
    ORDERED
    PRIVATE (list)
    FIRSTPRIVATE (list)
    LASTPRIVATE (list)
    SHARED (list)
    REDUCTION (operator | intrinsic : list)
    COLLAPSE (n)

do_loop

!$OMP END DO [ NOWAIT ]
```

**C/C++**

```c
#pragma omp for (clause ...) 
    schedule (type [,chunk])
    ordered
    private (list)
    firstprivate (list)
    lastprivate (list)
    shared (list)
    reduction (operator : list)
    collapse (n)
    nowait

for_loop
```

**SCHEDULE:**

- **Static** – loop divided into pieces of size chunk (def: evenly) then statically assigned to threads
- **Dynamic** – def chunk =1. Threads assigned dynamically i.e. start one chunk, and grab another when finished
- **Guided** – dynamic but in blocks that decrease in size, since blocksize = n_iter_remaining/n_threads
- **Runtime** – schedule determined at runtime by env variable OMP_SCHEDULE
- **AUTO** – compiler decides!

**NOWAIT:** No barrier synch at end of loop
OpenMP – Directives: DO

Data dependency:

```plaintext
 !$OMP DO
 Do  i=1,10
   ...
   A(i)=A(i-1)
   ...
 End Do
 !$OMP END DO
```

Simple test: Can serial loop be executed in reverse order with same result?
OpenMP – Directives: DO

Which is better?

A

Do i=1,100
    Do j = 1,100
        !$OMP DO
            Do k = 1,100
                A(i,j,k)=i*j*k
                End Do
        !$OMP END DO
    End Do
End Do

B

!$OMP DO
Do i=1,100
    Do j = 1,100
        Do k = 1,100
            A(i,j,k)=i*j*k
            End Do
        !$OMP END DO
    End Do
End Do

B.

(i) Work per thread is a LOT more

(ii) Less creation/destruction of threads => minimise overhead
OpenMP – Directives: DO

Can do even better?

```fortran
Do i=1,10
  Do j = 1,10
    !$OMP DO
      Do k = 1,10
        A(i,j,k)=i*j*k
      End Do
    !$OMP END DO
  End Do
  !$OMP END DO
End Do
```

```fortran
!$OMP DO
  Do i=1,10
    Do j = 1,10
      Do k=1,10
        A(i,j,k)=i*j*k
      End Do
    End Do
  End Do
!$OMP END DO
```

Fortran arrays are stored in column-major format
i.e. columns (first dimension – rows - changing) are contiguous in memory
Better cache performance
DO THIS ALWAYS FOR FORTRAN!
Enclosed sections of code are computed in parallel, one section per thread
Implicit barrier at end of SECTIONS unless NOWAIT
Cannot orphan SECTION i.e. have it physically outside a SECTIONS extent (see later)
OpenMP – Directives: WORKSHARE

FORTRAN ONLY!
Non DO loop, special Fortran stuff
Divides the block into “separate units of work”
ONLY CERTAIN ACTIONS CAN BE CONTAINED UNDER THE DIRECTIVE!

- array assignments
- MUTMUL, DOT_PRODUCT, SUM, PRODUCT, MAXVAL, MINVAL, MAXLOC, MINLOC, RESHAPE
- scalar assignments
- FORALL statements
- FORALL constructs
- WHERE statements
- WHERE constructs
- atomic constructs
- critical constructs

Block of code is parallelized sequentially (!), unit by unit (note: incurs overhead)
Variables which are referenced or modified within construct MUST be shared variables

---

**e.g.**

```
!$OMP PARALLEL!
!
INTEGER N, I, J
PARAMETER (N=100)
REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), FIRST, LAST
!
! Some initializations
!
!$OMP DO
!
DO I = 1, N
    DO J = 1, N
        AA(J,I) = I * 1.0
        BB(J,I) = J + 1.0
    END DO
!
ENDDO
!
!$OMP WORKSHARE!
!
CC = AA * BB
DD = AA + BB
FIRST = CC(1,1) + DD(1,1)
LAST = CC(N,N) + DD(N,N)
!
ENDDO
!
ENDDO
!
!$OMP END PARALLEL
!
END
```

---

```
!$OMP DO
Do i=2,1000
    B(i-1)=10*i
    A(i)=A(i)+B(i)
End Do
!
ENDDO
!
!$OMP WORKSHARE
!
Forall (i=1:999)
    B(i)=10*(i+1)
End Forall
!
ENDDO
!
ENDDO
!
!$OMP END WORKSHARE
```

**Bad**

**Good**
OpenMP – Directives: Composites

PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE

Same as PARALLEL followed by DO etc

Just for convenience

Note: PARALLEL incurs significant overhead so do not do multiple PARALLEL DO’s in a row; rather do one PARALLEL and multiple DO’s.

```fortran
PROGRAM VECTOR_ADD

INTEGER N, I, CHUNKSIZE, CHUNK
PARAMETER (N=1000)
PARAMETER (CHUNKSIZE=100)
REAL A(N), B(N), C(N)

! Some initializations
DO I = 1, N
   A(I) = I * 1.0
   B(I) = A(I)
ENDDO
CHUNK = CHUNKSIZE

!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)

   DO I = 1, N
      C(I) = A(I) + B(I)
   ENDDO

!$OMP END PARALLEL DO

END
```
OpenMP – Directives: **Single**

Code only operates on one thread of the team (first to arrive)

Important for parts of the code that are not thread-safe

Other threads wait at the end of the block unless NOWAIT

```
Fortran
!
$OMP SINGLE [clause ...]
PRIVATE (list)
FIRSTPRIVATE (list)

block

$OMP END SINGLE [ NOWAIT ]
```

```
C/C++

#pragma omp single [clause ...] newline
private (list)
firstprivate (list)
nowait

structured_block
```
NEW OpenMP 3.0!!

**TASKS**

Task specifies a block of code for task scheduling

Like section but scheduled

(Note: synchronize with TASKWAIT)

Parallel region is an implicit task; TASK is explicit

---

**Two models now: threading and tasking**

Threading: thread and work go together

Tasking: task generation and task execution separate. Thread is execution engine but may be deferred
OpenMP – Directives: Synchronisation

```
!$OMP MASTER
<block>
!$OMP END MASTER

!$OMP CRITICAL [name]
<block>
!$OMP END CRITICAL [name]

!$OMP BARRIER

!$OMP ATOMIC

!$OMP FLUSH (list)

!$OMP DO ORDERED
<...>
!$OMP ORDERED
<block>
!$OMP END ORDERED
<...>
!$OMP END DO

!$OMP THREADPRIVATE (list) – for common blocks ☀️ or private vars global to thread.
```

Only executed by the master
(no barrier for other threads)

Must be executed by one thread at a time
Threads BLOCK until other thread in critical region has completed
No name => treated as SAME critical section (names are global entities in Fortran)

Synchronizes all threads in the team

Mini-critical region. Must be updated atomically

Point at which consistent view of memory demanded. Thread variables written out of register and cache into main memory. Important EVEN IF shared memory is CACHE COHERENT!
Confusing! Flush is implied at end of some directives (e.g. BARRIER, PARALLEL, CRITICAL, DO, SECTION, …)

Fine tuning of loops: specifies which iterations of a loop are executed in the same order as if in serial. (e.g. writing to a file)

Must be in dynamic extent of DO (or PARALLEL DO) (see later … soon!)

Threads may have to wait if previous iteration not completed

```
!$omp parallel do private(myval) ordered
do i = 1, n
   myval = do_lots_of_work(i)
!$omp ordered
print*, i, myval
!$omp end ordered
enddo
```
**OpenMP – Directives: Extents**

**DIRECTIVE SCOPING: EXTENTS**

<table>
<thead>
<tr>
<th>Program Test</th>
<th>Subroutine SUB1</th>
</tr>
</thead>
<tbody>
<tr>
<td>![OMP PARALLEL]</td>
<td>![OMP CRITICAL]</td>
</tr>
<tr>
<td>![OMP DO]</td>
<td>![OMP END CRITICAL]</td>
</tr>
<tr>
<td>![CRITICAL]</td>
<td>![OMP SECTIONS]</td>
</tr>
<tr>
<td>![OMP END PARALLEL]</td>
<td>![OMP END SECTIONS]</td>
</tr>
</tbody>
</table>

**STATIC EXTENT**

The *do* directive occurs within an enclosing parallel region

**ORPHANED DIRECTIVES**

The *critical* and *sections* directives occur outside an enclosing parallel region

**DYNAMIC EXTENT**

The *critical* and *sections* directives occur within the dynamic extent of the *do* and *parallel* directives.

**Static (lexical):** textually enclosed, not spanning multiple routines or code files

**Orphaned:** Appears independently from enclosing directive i.e. OUTSIDE the static extent of the other

**Dynamic:** the composition of the two

**Sooo …?**

OpenMP has a bunch of rules about binding and nesting depending on the extent (see later)
OpenMP – Directives: Extents

e.g.

```
PROGRAM ORPHAN
COMMON /DOTDATA/ A, B, SUM
INTEGER I, VECLEN
PARAMETER (VECLEN = 100)
REAL*8 A(VECLEN), B(VECLEN), SUM

DO I=1, VECLEN
   A(I) = 1.0 * I
   B(I) = A(I)
ENDDO
SUM = 0.0
!$OMP PARALLEL
CALL DOTPROD
!$OMP END PARALLEL
WRITE(*,*) "Sum = ", SUM
END

SUBROUTINE DOTPROD
COMMON /DOTDATA/ A, B, SUM
INTEGER I, TID, OMP_GET_THREAD_NUM, VECLEN
PARAMETER (VECLEN = 100)
REAL*8 A(VECLEN), B(VECLEN), SUM

TID = OMP_GET_THREAD_NUM()
!$OMP DO REDUCTION(+:SUM)
DO I=1, VECLEN
   SUM = SUM + (A(I)*B(I))
   PRINT *, ' TID=', TID,'I=', I
ENDDO
RETURN
END
```

static extent

dynamic extent

orphaned
OpenMP – Directives: Data scope clauses

Shared memory => most data shared by default

Default GLOBAL variables include:

- FORTRAN – COMMON blocks (!), SAVE variables, MODULE variables
- C – file scope variables, static

Default PRIVATE variables include:

- Loop index variables
- Stack variables in subroutines called from parallel regions

Explicit scoping via Data Scope Attribute Clauses:

Controls which and how variables transferred into threads

- **PRIVATE (list)**
  - Variables in list are private to each thread
  - New object of same type created for each thread.
  - Uninitialised (see: FIRSTPRIVATE, LASTPRIVATE)

- **SHARED (list)**
  - Variables in list are shared amongst team
  - Only one memory location and all threads can write to it
  - Performs a reduction operation on variables in list

- **REDUCTION (op|intrinsic:list)**
  - Private copy created for each thread; reduction applied to all private copies and applied to global shared variable

- DEFAULT, FIRSTPRIVATE, LASTPRIVATE, COPYIN, COPYPRIVATE,
  - Sets default
  - First private val set by prior serial
  - Serial val set by last private
  - Copies for threadpriv
  - Broadcast private val from SINGLE to other threads
OpenMP – Directives: Data scope clauses

e.g.

```fortran
PROGRAM DOT_PRODUCT

INTEGER N, CHUNKSIZE, CHUNK, I
PARAMETER (N=100)
PARAMETER (CHUNKSIZE=10)
REAL A(N), B(N), RESULT

! Some initializations
DO I = 1, N
  A(I) = I * 1.0
  B(I) = I * 2.0
ENDDO
RESULT = 0.0
CHUNK = CHUNKSIZE

!$OMP PARALLEL DO
!$OMP& DEFAULT(SHARED), PRIVATE(I)
!$OMP& SCHEDULE(STATIC, CHUNK)
!$OMP& REDUCTION(+:RESULT)
DO I = 1, N
  RESULT = RESULT + (A(I) * B(I))
ENDDO

!$OMP END PARALLEL DO

PRINT *, 'Final Result= ', RESULT
END
```
OpenMP – Directives: Clauses: Summary

The table below summarizes which clauses are accepted by which OpenMP directives.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Directive</th>
<th>PARALLEL</th>
<th>DO/for</th>
<th>SECTIONS</th>
<th>SINGLE</th>
<th>PARALLEL DO/for</th>
<th>PARALLEL SECTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIVATE</td>
<td></td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td>..</td>
<td>..</td>
<td></td>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRSTPRIVATE</td>
<td></td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASTPRIVATE</td>
<td></td>
<td>..</td>
<td>..</td>
<td></td>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REDUCTION</td>
<td></td>
<td>..</td>
<td>..</td>
<td></td>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPYIN</td>
<td></td>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPYPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCHEDULE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDERED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOWAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following OpenMP directives do not accept clauses:
- MASTER
- CRITICAL
- BARRIER
- ATOMIC
- FLUSH
- ORDERED
- THREADPRIVATE

Implementations may (and do) differ from the standard in which clauses are supported by each directive.
Urgh!

**Directive Binding:**
- The DO/for, SECTIONS, SINGLE, MASTER and BARRIER directives bind to the dynamically enclosing PARALLEL, if one exists. If no parallel region is currently being executed, the directives have no effect.
- The ORDERED directive binds to the dynamically enclosing DO/for.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL.

**Directive Nesting:**
- A worksharing region may not be closely nested inside a worksharing, explicit task, critical, ordered, atomic, or master region.
- A barrier region may not be closely nested inside a worksharing, explicit task, critical, ordered, atomic, or master region.
- A master region may not be closely nested inside a worksharing, atomic, or explicit task region.
- An ordered region may not be closely nested inside a critical, atomic, or explicit task region.
- An ordered region must be closely nested inside a loop region (or parallel loop region) with an ordered clause.
- A critical region may not be nested (closely or otherwise) inside a critical region with the same name. Note that this restriction is not sufficient to prevent deadlock.
- parallel, flush, critical, atomic, taskyield, and explicit task regions may not be closely nested inside an atomic region.

The term "closely nested region" means a region that is dynamically nested inside another region with no parallel region nested between them.
OpenMP – Runtime Library Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_SET_NUM_THREADS</td>
<td>Sets the number of threads that will be used in the next parallel region</td>
</tr>
<tr>
<td>OMP_GET_NUM_THREADS</td>
<td>Returns the number of threads that are currently in the team executing the parallel region from which it is called</td>
</tr>
<tr>
<td>OMP_GET_MAX_THREADS</td>
<td>Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function</td>
</tr>
<tr>
<td>OMP_GET_THREAD_NUM</td>
<td>Returns the thread number of the thread, within the team, making this call.</td>
</tr>
<tr>
<td>OMP_GET_THREAD_LIMIT</td>
<td>Returns the maximum number of OpenMP threads available to a program</td>
</tr>
<tr>
<td>OMP_GET_NUM_PROCS</td>
<td>Returns the number of processors that are available to the program</td>
</tr>
<tr>
<td>OMP_IN_PARALLEL</td>
<td>Used to determine if the section of code which is executing is parallel or not</td>
</tr>
<tr>
<td>OMP_GET_DYNAMIC</td>
<td>Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions</td>
</tr>
<tr>
<td>OMP_GET_DYNAMIC</td>
<td>Used to determine if dynamic thread adjustment is enabled or not</td>
</tr>
<tr>
<td>OMP_SET_DYNAMIC</td>
<td>Used to enable or disable nested parallelism</td>
</tr>
<tr>
<td>OMP_SET_NESTED</td>
<td>Used to determine if nested parallelism is enabled or not</td>
</tr>
<tr>
<td>OMP_SET_SCHEDULE</td>
<td>Sets the loop scheduling policy when &quot;runtime&quot; is used as the schedule kind in the OpenMP directive</td>
</tr>
<tr>
<td>OMP_GET_SCHEDULE</td>
<td>Returns the loop scheduling policy when &quot;runtime&quot; is used as the schedule kind in the OpenMP directive</td>
</tr>
<tr>
<td>OMP_SET_MAX_ACTIVE_LEVELS</td>
<td>Sets the maximum number of nested parallel regions</td>
</tr>
<tr>
<td>OMP_GET_MAX_ACTIVE_LEVELS</td>
<td>Returns the maximum number of nested parallel regions</td>
</tr>
<tr>
<td>OMP_GET_LEVEL</td>
<td>Returns the current level of nested parallel regions</td>
</tr>
<tr>
<td>OMP_GET_ANCESTOR_THREAD_NUM</td>
<td>Returns, for a given nested level of the current thread, the thread number of ancestor thread</td>
</tr>
<tr>
<td>OMP_GET_TEAM_SIZE</td>
<td>Returns, for a given nested level of the current thread, the size of the thread team</td>
</tr>
<tr>
<td>OMP_GET_ACTIVE_LEVEL</td>
<td>Returns the number of nested, active parallel regions enclosing the task that contains the call</td>
</tr>
<tr>
<td>OMP_IN_FINAL</td>
<td>Returns true if the routine is executed in the final task region; otherwise it returns false</td>
</tr>
<tr>
<td>OMP_INIT_LOCK</td>
<td>Initializes a lock associated with the lock variable</td>
</tr>
<tr>
<td>OMP_DESTROY_LOCK</td>
<td>Disassociates the given lock variable from any locks</td>
</tr>
<tr>
<td>OMP_SET_LOCK</td>
<td>Acquires ownership of a lock</td>
</tr>
<tr>
<td>OMP_UNSET_LOCK</td>
<td>Releases a lock</td>
</tr>
<tr>
<td>OMP_TEST_LOCK</td>
<td>Attempts to set a lock, but does not block if the lock is unavailable</td>
</tr>
<tr>
<td>OMP_INIT_NEST_LOCK</td>
<td>Initializes a nested lock associated with the lock variable</td>
</tr>
<tr>
<td>OMP_DESTROY_NEST_LOCK</td>
<td>Disassociates the given nested lock variable from any locks</td>
</tr>
<tr>
<td>OMP_SET_NEST_LOCK</td>
<td>Acquires ownership of a nested lock</td>
</tr>
<tr>
<td>OMP_UNSET_NEST_LOCK</td>
<td>Releases a nested lock</td>
</tr>
<tr>
<td>OMP_TEST_NEST_LOCK</td>
<td>Attempts to set a nested lock, but does not block if the lock is unavailable</td>
</tr>
<tr>
<td>OMP_GET_WTIME</td>
<td>Provides a portable wall clock timing routine</td>
</tr>
<tr>
<td>OMP_GET_WTICK</td>
<td>Returns a double-precision floating point value equal to the number of seconds between successive clock ticks</td>
</tr>
</tbody>
</table>

In Fortran, some are functions and some are subroutines 😊

e.g.
Call OMP_SET_NUM_THREADS(8)

e.g.
Integer nt
nt=OMP_GET_NUM_THREADS()

Locks are for blocking:
Init_lock -- initiates var
Set_lock – wait until lock available
Unset_lock – releases lock
Test_lock – like set but does not block if lock not available
Destroy_lock – removes lock variable

LOCKS = INTEGER*8 (to hold an address)
OpenMP – Lock example

PROGRAM BUGS

INTEGER*8 LOCKA, LOCKB
INTEGER NTHREADS, TID, I,
+ OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
PARAMETER (N=1000000)
REAL A(N), B(N), PI, DELTA
PARAMETER (PI=3.1415926535)
PARAMETER (DELTA=.01415926535)

C Initialize the locks
CALL OMP_INIT_LOCK(LOCKA)
CALL OMP_INIT_LOCK(LOCKB)

C Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL SHARED(A, B, NTHREADS, LOCKA, LOCKB) PRIVATE(TID)

C Obtain thread number and number of threads
TID = OMP_GET_THREAD_NUM()
!$OMP MASTER
NTHREADS = OMP_GET_NUM_THREADS()
PRINT *, 'Number of threads = ', NTHREADS
!$OMP END MASTER
PRINT *, 'Thread', TID, 'starting...'
!$OMP BARRIER

!$OMP SECTIONS

!$OMP SECTION
PRINT *, 'Thread', TID, 'initializing A()
CALL OMP_SET_LOCK(LOCKA)
DO I = 1, N
A(I) = I * DELTA
ENDO
CALL OMP_SET_LOCK(LOCKB)
PRINT *, 'Thread', TID, 'adding A() to B()
DO I = 1, N
B(I) = B(I) + A(I)
ENDO
CALL OMP_UNSET_LOCK(LOCKA)
CALL OMP_UNSET_LOCK(LOCKB)

!$OMP SECTION
PRINT *, 'Thread', TID, 'initializing B()
CALL OMP_SET_LOCK(LOCKB)
DO I = 1, N
B(I) = I * PI
ENDO
CALL OMP_SET_LOCK(LOCKA)
PRINT *, 'Thread', TID, 'adding B() to A()
DO I = 1, N
A(I) = A(I) + B(I)
ENDO
CALL OMP_UNSET_LOCK(LOCKA)
CALL OMP_UNSET_LOCK(LOCKB)

!$OMP END SECTIONS NONWAIT
PRINT *, 'Thread', TID, 'done.'
!$OMP END PARALLEL
END
OpenMP – Environment Variables

**OMP_SCHEDULE**

Applies only to DO, PARALLEL DO (Fortran) and for, parallel for (C/C++) directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors. For example:

```
setenv OMP_SCHEDULE "guided, 4"
setenv OMP_SCHEDULE "dynamic"
```

**OMP_NUM_THREADS**

Sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

**OMP_DYNAMIC**

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE. For example:

```
setenv OMP_DYNAMIC TRUE
```

**OMP_PROC_BIND**

Enables or disables threads binding to processors. Valid values are TRUE or FALSE. For example:

```
setenv OMP_PROC_BIND TRUE
```

**OMP_NESTED**

Enables or disables nested parallelism. Valid values are TRUE or FALSE. For example:

```
setenv OMP_NESTED TRUE
```

If nested parallelism is supported, it is often only nominal, in that a nested parallel region may only have one thread.
OpenMP – Environment Variables

**OMP_STACKSIZE**
Controls the size of the stack for created (non-Master) threads. Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k"
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE "10 M"
setenv OMP_STACKSIZE "20 m"
setenv OMP_STACKSIZE "1G"
setenv OMP_STACKSIZE 20000
```

**OMP_WAIT_POLICY**
Provides a hint to an OpenMP implementation about the desired behavior of waiting threads. A compliant OpenMP implementation may or may not abide by the setting of the environment variable. Valid values are ACTIVE and PASSIVE. ACTIVE specifies that waiting threads should mostly be active, i.e., consume processor cycles, while waiting. PASSIVE specifies that waiting threads should mostly be passive, i.e., not consume processor cycles, while waiting. The details of the ACTIVE and PASSIVE behaviors are implementation defined. Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

**OMP_MAX_ACTIVE_LEVELS**
Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of OMP_MAX_ACTIVE_LEVELS is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer. Example:

```
setenv OMP_MAX_ACTIVE_LEVELS 2
```

**OMP_THREAD_LIMIT**
Sets the number of OpenMP threads to use for the whole OpenMP program. The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of OMP_THREAD_LIMIT is greater than the number of threads an implementation can support, or if the value is not a positive integer. Example:

```
setenv OMP_THREAD_LIMIT 8
```
Final tips

Thread stack size
Implementation specific and can be small!
e.g. gfortran ~ 2MB (~500x500 double values)

OpenMP 3.0:
```
setenv OMP_STACKSIZE 10M
```

Linux:
```
setenv KMP_STACKSIZE 12000000
limit stacksize unlimited
```

Thread binding
Performance may be better if threads bound to processors (thread affinity) due to cache re-use

OpenMP 3.1:
```
setenv OMP_PROC_BIN TRUE
```
1. Write a “hello world”

2. Write a program that initialises two matrices $A$ and $B$ with some values, calculates the product $C = AB$ and finds the minimum value in the $C$ matrix and where it is in the matrix.