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!
- Directive-based (compiler needs help) ✓ **OpenMP**
- Task farming (scientific term for large transaction processing)

MapReduce/Hadoop etc

Usual confessions!

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

https://computing.llnl.gov/tutorials/openMP

Other really good sources added to the webpage:

Parallel Programming in Fortran 95 using OpenMP, Miguel Hermanns, Universidad Politecnica de Madrid

Advanced OpenMP Topics, NAS seminar, NASA

OpenMP by Example, TACC training

etc

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 X3H5 - never adopted as distributed machines emerged
  - Newer shared mem machines came along
  - 1997: OpenMP standard took up where ANSI left off: OpenMP Architecture Review Board (ARB): Compaq, HP, Intel. IBM, Silicon Graphics, Sun, Fujitsu, DoE-ASCI, …
  - API specs separate releases for C and Fortran until combined in 2005
  - Latest: OpenMP 5.0 Nov 2018

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 standardization built and endorsed by many
  - "lean and mean" and simple (although trend away with later versions ☹)
  - 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

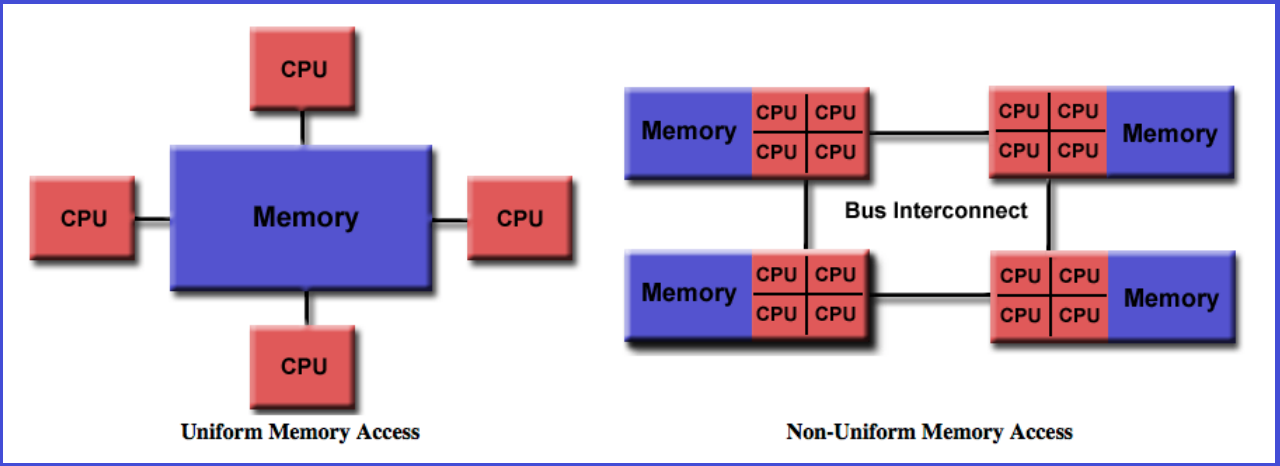
☺ Directives are like comments so sequential code still runs

☺ Code is lightweight

☺ Data decomposition is largely invisible/automatic

OpenMP

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

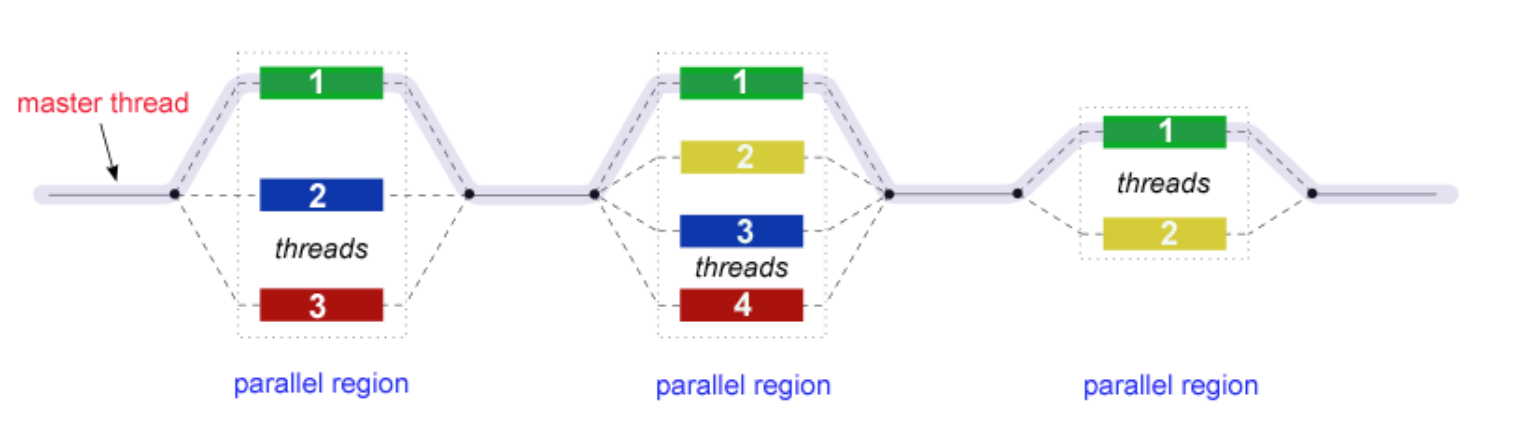


- Parallelism by use of THREADS
- Threads are smallest units of processing
- Many threads share resources of a single process
- Typically, number of threads matches number of cores, but not necessarily (can overlap some work with virtual threads)

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 parallellize
- 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



OpenMP

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

"sentinel":

`!$OMP` -- free format source

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

Directive

Optional clauses

e.g.

Fortran	`!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)`
C/C++	`#pragma omp parallel default(shared) private(beta,pi)`

or

```
!$OMP  directive
    [ structured block of code ]
!$OMP end  directive
```

OpenMP: API

Programming model:

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

- Compiler directives

Fortran	`!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)`
C/C++	`#pragma omp parallel default(shared) private(beta,pi)`

Extending lines: Note that there is ANOTHER sentinel:

```
!$OMP PARALLEL DEFAULT(private) SHARED(vars...., &
!$OMP& more_vars..., &
!$OMP& more_vars... &
!$OMP& )
```

An OpenMP-compliant compiler interprets this as conditional text that should be included whereas a non-OpenMP-compliant compiler will interpret it as a comment.

OpenMP: API

Programming model:

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

- Runtime routines that you can call
  - Setting and querying number of threads
  - Querying thread identifiers
  - Setting/querying nested parallelism
  - etc etc

e.g.

Fortran	`INTEGER FUNCTION OMP_GET_NUM_THREADS()`
C/C++	`#include <omp.h>` `int omp_get_num_threads(void)`

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.

csh/tcsh	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

OpenMP: Typical code

Fortran

```
      PROGRAM HELLO

      INTEGER VAR1, VAR2, VAR3

      Serial code
            .
            .
            .

      Beginning of parallel section. Fork a team of threads.
      Specify variable scoping

!$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

```
#include <omp.h>

main ()  {

int var1, var2, var3;

Serial code
      .
      .
      .

Beginning of parallel section. Fork a team of threads.
Specify variable scoping

#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:

Compiler / Platform	Compiler	Flag

Intel
Linux Opteron/Xeon
	`icc`
	`icpc`
	`ifort`
		`-openmp`

PGI
Linux Opteron/Xeon
	`pgcc`
	`pgCC`
	`pgf77`
	`pgf90`
		`-mp`

GNU
Linux Opteron/Xeon
IBM Blue Gene
	`gcc`
	`g++`
	`g77`
	`gfortran`
		`-fopenmp`

IBM
Blue Gene
	`bgxlc_r, bgcc_r`
	`bgxlC_r, bgxlc++_r`
	`bgxlc89_r`
	`bgxlc99_r`
	`bgxlf_r`
	`bgxlf90_r`
	`bgxlf95_r`
	`bgxlf2003_r`
	
	*Be sure to use a thread-safe compiler - its name ends with **_r**
		`-qsmp=omp`

- Compiler Documentation:
  - IBM BlueGene: www-01.ibm.com/software/awdtools/fortran/ and www-01.ibm.com/software/awdtools/xlcpp
  - Intel: www.intel.com/software/products/compilers/
  - PGI: www.pgroup.com
  - GNU: gnu.org
  - All: See the relevant man pages and any files that might relate in `/usr/local/docs`

OpenMP: Example – Hello World

Let's just do an example: Hello World!

```
      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

Notice master thread = 0

Example

Notice you needed a bunch of things:

```
gfortran –fopenmp <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

```
!$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++

```
#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

Implied synchronization at the end of the parallel region.

OpenMP - Directives

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

```
!$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++

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

   for_loop
```

```
      PROGRAM VEC_ADD_DO

      INTEGER N, CHUNKSIZE, CHUNK, I
      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 SHARED(A,B,C,CHUNK) PRIVATE(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

      END
```

SCHEDULE:

Static – loop divided into pieces of size chunk (default: evenly) then statically assigned to threads

Dynamic – default 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 implied barrier synch at end of loop (Note: be careful to FLUSH if re-use constructs from within loop etc)

OpenMP – Directives: DO

Data dependency:

Simple test: Can serial loop be executed in reverse order with same result?

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

```
!$OMP DO ORDERED
  do i = 1, 1000
    !$OMP ORDERED
      A(i) = A(i-1)
    !$OMP END ORDERED
  enddo
!$OMP END DO
```

BUT in this case, this serializes the code!

ORDERED construct only useful for ordering a nested segment really:

Read-after-write (RAW) data dependency

```
!$OMP DO ORDERED
do i = 1, 100
  block1

  !$OMP ORDERED
    block2
  !$OMP END ORDERED

  block3
enddo
!$OMP END DO
```

Aside: Data dependencies

```
1. A=3
2. B=A
3. C=B
```

Read-After-Write (RAW, flow, true) dependency:

Instruction requires result from previous instruction

Instruction 3 truly depends on 2; 2 truly depends on 1; 3 truly depends on 1

No parallelism.

```
1. B=3
2. A=B+1
3. B=7
```

Write-After-Read (WAR, anti-) dependency:

Instruction requires a result that is later updated

Instruction 3 anti-depends on 2

No parallelism; but can be fixed by renaming

```
1. B=3
2. B2=B
3. A=B2+1
4. B=7
```

WAR btw 3-4 removed

Now RAW btw 1-2-3

```
1. B=3
2. A=B+1
3. B=7
```

Write-After-Write (WAW, output) dependency:

Reording instruction changes final output of a variable

WAW btw 1-3. Reordering changes final value of A

No parallelism; but can be fixed by renaming

```
1. B2=3
2. A=B2+1
3. B=7
```

OpenMP – Directives: DO

Data dependency:

Simple test: Can serial loop be executed in reverse order with same result?

```
real(8) :: A(1000)

do i = 1, 999
  A(i) = A(i+1)
enddo
```

Write-after-read (WAR) data dependency

Solvable!

```
real(8) :: A(1000), dummy(2:1000:2)

!Saves the even indices
!$OMP DO
  do i = 2, 1000, 2
    dummy(i) = A(i)
  enddo
!$OMP END DO

!Updates even indices from odds
!$OMP DO
  do i = 0, 998, 2
    A(i) = A(i+1)
  enddo
!$OMP END DO

!Updates odd indices with evens
!$OMP DO

  do i = 1, 999, 2
    A(i) = dummy(i+1)
  enddo
!$OMP END DO
```

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

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

B.

(i) Work per thread is a LOT more

(ii) Less creation/destruction of threads => minimizes overhead too

OpenMP – Directives: DO

Can do even better?

```
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
End Do
```

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

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

OpenMP – Directives: SECTIONS

Fortran

```
!$OMP SECTIONS [clause ...]
               PRIVATE (list)
               FIRSTPRIVATE (list)
               LASTPRIVATE (list)
               REDUCTION (operator | intrinsic : list)

!$OMP  SECTION

   block

!$OMP  SECTION

    block

!$OMP END SECTIONS  [ NOWAIT ]
```

C/C++

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {
  #pragma omp section   newline

     structured_block

  #pragma omp section   newline

     structured_block

  }
```

```
      PROGRAM VEC_ADD_SECTIONS

      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N), D(N)

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.5
        B(I) = I + 22.35
      ENDDO

!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO

!$OMP SECTION
      DO I = 1, N
         D(I) = A(I) * B(I)
      ENDDO

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL

      END
```

Functional parallelism (MPMD)

Enclosed sections of code are computed in parallel, one section per thread (unless more sections than threads, then some threads have multiple sections executed serially)

Implicit barrier at end of SECTIONS unless NOWAIT

Cannot orphan SECTION i.e. have it physically outside a SECTIONS extent (see later)

No conditional branching out of sections

OpenMP – Directives: WORKSHARE

```
      PROGRAM WORKSHARE

      INTEGER N, I, J
      PARAMETER (N=100)
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), FIRST, LAST

!     Some initializations
      DO I = 1, N
        DO J = 1, N
          AA(J,I) = I * 1.0
          BB(J,I) = J + 1.0
        ENDDO
      ENDDO

!$OMP PARALLEL SHARED(AA,BB,CC,DD,FIRST,LAST)

!$OMP WORKSHARE
      CC = AA * BB
      DD = AA + BB
      FIRST = CC(1,1) + DD(1,1)
      LAST = CC(N,N) + DD(N,N)
!$OMP END WORKSHARE NOWAIT

!$OMP END PARALLEL

      END
```

FORTRAN ONLY!

Parallelizes Fortran commands that have hidden implicit DO loops (like full array operations, FORALL, WHERE, array intrinsics like MATMUL, etc)

ONLY CERTAIN ACTIONS CAN BE CONTAINED UNDER THE DIRECTIVE!

- ✓ array assignments
- ✓ MUTMUL, DOT_PRODUCT, SUM, PRODUCT. MAXVAL, MINVAL, MAXLOC, MINLOC, RESHAPE, TRANSPOSE, PACK, CSHIFT, …
- ✓ scalar assignments
- ✓ FORALL statements
- ✓ FORALL constructs
- ✓ WHERE statements and constructs
- ✓ ATOMIC constructs
- ✓ CRITICAL constructs

Block of code is parallelized sequentially (!), parallelizing each unit of work (line) one at a time (note: incurs overhead, syncronizing)

Variables which are referenced or modified within construct MUST be shared variables

e.g.

```
!$OMP DO
Do i=2,1000
  B(i)=10*(i+1)
  A(i)=A(i)+B(i)
End Do
!$OMP END DO
```

Bad

(RAW)

```
!$OMP WORKSHARE
Forall (i=1:999)
  B(i)=10*(i+1)
End Forall
A=A+B
!$OMP END WORKSHARE
```

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.

```
      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`

OpenMP – Directives: Task

NEW OpenMP 3.0!!

Task specifies a block of code for task scheduling

Like section but scheduled

(Note: synchronize with TASKWAIT)

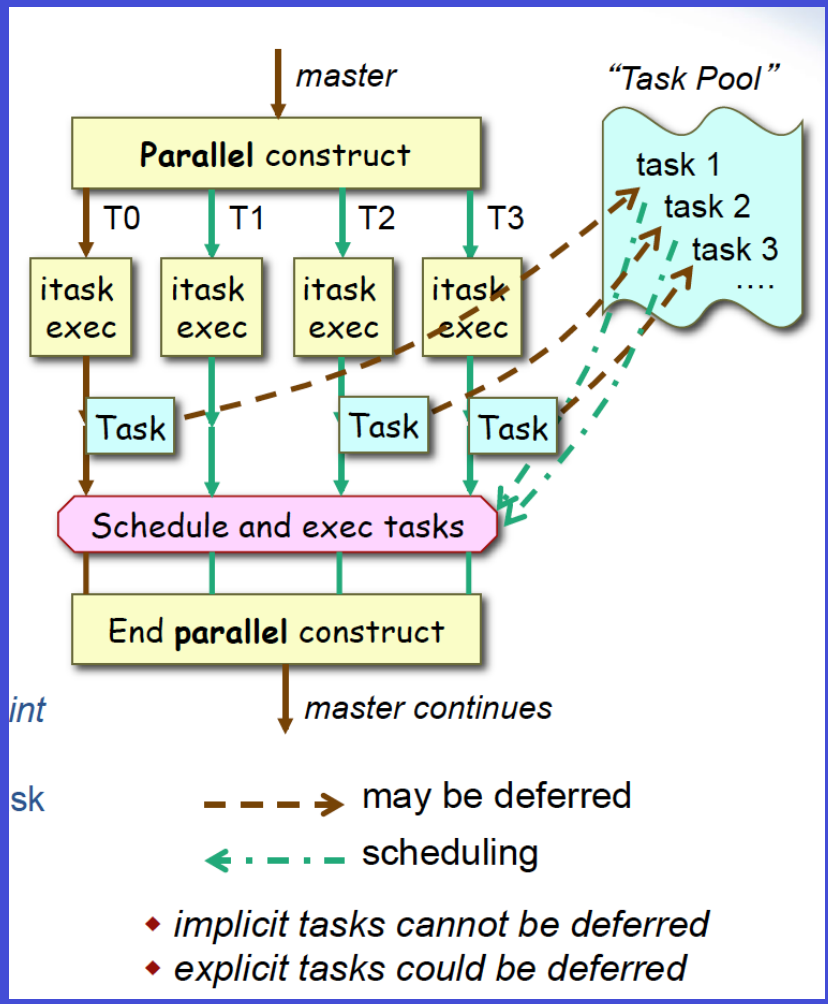
Parallel region is an implicit task; TASK is explicit

Fortran

```
!$OMP TASK [clause ...]
             IF (scalar logical expression)
             FINAL (scalar logical expression)
             UNTIED
             DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
             MERGEABLE
             PRIVATE (list)
             FIRSTPRIVATE (list)
             SHARED (list)

   block

!$OMP END TASK
```

C/C++

```
#pragma omp task [clause ...]  newline
                   if (scalar expression)
                   final (scalar expression)
                   untied
                   default (shared | none)
                   mergeable
                   private (list)
                   firstprivate (list)
                   shared (list)

     structured_block
```



Two models now: threading and tasking

Threading: thread and work go together

Tasking: task generation and task execution separate. Thread is execution engine but no direct control of when task executed. More dynamic environment.

OpenMP – Directives: Synchronization

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

Only executed by the master

(no barrier for other threads)

Same as single except thread is master not first to arrive.

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

Must be executed by one thread at a time

Threads BLOCK until executing thread in critical region has completed

All critical sections with no name are treated as SAME critical section (names are global entities in Fortran). Recommendation: always give them names!

```
!$OMP BARRIER
```

Synchronizes all threads in the team. All threads must hit the same BARRIER! Do not call in a SECTION for example

```
!$OMP ATOMIC
```

Mini (one line) critical region. Must be updated atomically (i.e. only one thread can access at a time). Good for writing to shared variables, for example doing a global sum a=a+i.

```
!$OMP FLUSH (list)
```

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, …)

```
!$OMP DO ORDERED
<…>
!$OMP ORDERED
<block>
!$OMP END ORDERED
<…>
!$OMP END DO
```

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: Synchronization

ATOMIC -- quick example:

Trying to avoid race/overwrite of shared variable A using ATOMIC

```
!$OMP DO
  do i = 1, 10000
    !$OMP ATOMIC
      A = A + i
  enddo
!$OMP END DO
```

Not efficient!

Better:

```
Atmp = 0

!$OMP DO
  do i = 1, 1000
      Atmp = Atmp + i
  enddo
!$OMP END DO

!$OMP ATOMIC
A = A + Atmp
```

OpenMP – Directives: Synchronization

IMPLICIT DATA SYNCHRONIZATIONS:

YES!

- !$OMP BARRIER
- !$OMP CRITICAL and !$OMP END CRITICAL
- !$OMP END DO
- !$OMP END SECTIONS
- !$OMP END SINGLE
- !$OMP END WORKSHARE
- !$OMP ORDERED and !$OMP END ORDERED
- !$OMP PARALLEL and !$OMP END PARALLEL
- !$OMP PARALLEL DO and !$OMP END PARALLEL DO
- !$OMP PARALLEL SECTIONS and !$OMP END PARALLEL SECTIONS
- !$OMP PARALLEL WORKSHARE and !$OMP END PARALLEL WORKSHARE

NO!

- !$OMP DO
- !$OMP MASTER and !$OMP END MASTER
- !$OMP SECTIONS
- !$OMP SINGLE
- !$OMP WORKSHARE

Obviously NOWAIT overrides, if you use it.

Recap from Tues

OpenMP – Directives: Data environment

One separate directive (now)

+ data scoping associated with existing directives (see next)

```
!$OMP THREADPRIVATE (list)
```

-- variable is local to each thread but global in extent to the thread

– equivalent to SAVE'd declared variables in FORTRAN (or COMMON variables)

```
integer, save :: a
!$OMP THREADPRIVATE(a)

!$OMP PARALLEL
  a = OMP_get_thread_num()
!$OMP END PARALLEL

!$OMP PARALLEL
  ...
!$OMP END PARALLEL
```

Keeps thread id in a as allocated from first parallel region

OpenMP – Directives: Extents

DIRECTIVE SCOPING: EXTENTS

<table>
<tr>
<td>

```
      PROGRAM TEST
      ...
!$OMP PARALLEL
      ...
!$OMP DO
      DO I=...
      ...
      CALL SUB1
      ...
      ENDDO
      ...
      CALL SUB2
      ...
!$OMP END PARALLEL
```

</td>
<td>

```
      SUBROUTINE SUB1
      ...
!$OMP CRITICAL
      ...
!$OMP END CRITICAL
      END


      SUBROUTINE SUB2
      ...
!$OMP SECTIONS
      ...
!$OMP END SECTIONS
      ...
      END
```

</td>
</tr>
<tr>
<td>STATIC EXTENT
The DO directive occurs within an enclosing parallel region</td>
<td>ORPHANED DIRECTIVES
The CRITICAL and SECTIONS directives occur outside an enclosing parallel region</td>
</tr>
<tr>
<td colspan="2">DYNAMIC EXTENT
The CRITICAL and SECTIONS directives occur within the dynamic extent of the DO and PARALLEL directives.</td>
</tr>
</table>

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

orphaned

dynamic extent

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

Fortran
x = x operator expr ***x = expr operator x*** *(except subtraction)* ***x = intrinsic(x, expr)*** ***x = intrinsic(expr, x)***
x is a scalar variable in the list ***expr*** is a scalar expression that does not reference ***x*** ***intrinsic*** is one of MAX, MIN, IAND, IOR, IEOR ***operator*** is one of +, *, -, .AND., .OR., .EQV., .NEQV.

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? Final value? (see: `FIRSTPRIVATE, LASTPRIVATE, etc`)

`SHARED (list)`

Variables in list are shared amongst team

Only one memory location and all threads can write to it

`REDUCTION (op|intrinsic:list)`

Performs a reduction operation on variables in 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 | Copys for threadpriv | Broadcast private val from SINGLE to other threads

OpenMP – Directives: Data scope clauses

`DEFAULT(DEFSCOPE)` – sets default scope `SHARED` (the default `DEFAULT`!) | `PRIVATE` | `NONE`

`FIRSTPRIVATE(VAR)` – sets first value of private var to value it had coming into region (otherwise is undefined)

`LASTPRIVATE(VAR)` – sets exit value of var to last value it would have had in sequential code (Note: takes place at time of synchronization, so watch out for NOWAIT!)

`COPYIN(VAR)` – sets first value of a `THREADPRIVATE` var to the value supplied from the MASTER thread

`COPYPRIVATE(VAR)` – broadcasts the result of a `SINGLE` region to the other threads

```
!$OMP SINGLE
  read(1,*) a
!$OMP END SINGLE COPYPRIVATE(a)
```

OpenMP – Directives: Data scope clauses

e.g.

```
      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: Synchronization

REMEMBER FROM EARLIER??? ATOMIC -- quick example:

Trying to avoid race/overwrite of shared variable A using ATOMIC

```
!$OMP DO
  do i = 1, 10000
    !$OMP ATOMIC
      A = A + i
  enddo
!$OMP END DO
```

Not efficient!

Better:

```
Atmp = 0

!$OMP DO
  do i = 1, 1000
      Atmp = Atmp + i
  enddo
!$OMP END DO

!$OMP ATOMIC
A = A + Atmp
```

NOTE: REDUCTION solves this (earlier) problem too!

```
!$OMP DO REDUCTION(+:a)
  do i = 1, 1000
    a = a + i
  enddo
!$OMP END DO
```

(Note again: shared variable only updated at time of synch)

OpenMP – Directives: Data scope clauses

Other clauses:

`IF (scalar_logical_expr)` – only do parallel op if expr evaluates true e.g. if n_iters in loop is big enough

`NUM_THREADS (scalar_integer_expr)` – number of threads to use. Overrrides other settings such as those from `GET_NUM_THREADS`

`NOWAIT` – threads do not have to finish at same time

`SCHEDULE(TYPE, CHUNK)` – sets the way DO loops iterations are scheduled:

- Static – loop divided into pieces of size chunk (default: evenly) then statically assigned to threads
- Dynamic – default 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!

`ORDERED` – sets loops to sequential

OpenMP – Directives: Clauses: Summary

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

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

- 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.

OpenMP – Directives: Binding/nesting

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

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

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 Main
use omp_lib
implicit none

integer(kind = OMP_lock_kind) :: lck
integer(kind = OMP_integer_kind) :: ID

call OMP_init_lock(lck)

!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
  ID = OMP_get_thread_num()

  call OMP_set_lock(lck)
  write(*,*) "My thread is ", ID
  call OMP_unset_lock(lck)

!$OMP END PARALLEL

call OMP_destroy_lock(lck)

end program Main
```

This dumb example is equivalent to:

```
program Main
use omp_lib
implicit none

integer(kind = OMP_integer_kind) :: ID

!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
  ID = OMP_get_thread_num()

!$OMP CRITICAL
  write(*,*) "My thread is ", ID
!$OMP END CRITICAL

!$OMP END PARALLEL

end program Main
```

OpenMP – Lock example

```
      PROGRAM BUG5

      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
      ENDDO
      CALL OMP_SET_LOCK(LOCKB)
      PRINT *, 'Thread',TID,' adding A() to B()'
      DO I = 1, N
         B(I) = B(I) + A(I)
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKB)
      CALL OMP_UNSET_LOCK(LOCKA)

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

!$OMP END SECTIONS NOWAIT

      PRINT *, 'Thread',TID,' done.'

!$OMP END PARALLEL

      END
```

```
      PROGRAM BUG5

      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
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKA)
      CALL OMP_SET_LOCK(LOCKB)
      PRINT *, 'Thread',TID,' adding A() to B()'
      DO I = 1, N
         B(I) = B(I) + A(I)
      ENDDO
      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
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKB)
      CALL OMP_SET_LOCK(LOCKA)
      PRINT *, 'Thread',TID,' adding B() to A()'
      DO I = 1, N
         A(I) = A(I) + B(I)
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKA)

!$OMP END SECTIONS NOWAIT

      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
```

Things we didn't talk about much

OpenMP 3.0-5.0

(Task parallelism)

Dynamics creation of threads

Nested parallelism

Support for co-processors/accelerators

Support for NUMA systems

OpenMP 5.0 press release:

- **Full support for accelerator devices**. OpenMP now has full support for accelerator devices, including mechanisms to require unified shared memory between the host system and coprocessor devices, the ability to use device-specific function implementations, better contro of implicit data mappings, and the ability to override device offload at runtime. In addition, it supports reverse offload, implicit function generation, and the ability to copy object-oriented data structures easily.
- **Improved debugging and performance analysis.** Two new tool interfaces enable the development of third party tools to support intuitive debugging and deep performance analysis.
- **Support for the latest versions of C, C++, and Fortran.** OpenMP now supports important features of Fortran 2008, C11, and C++17.
- **Support for a fully descriptive loop construct.** The loop construct lets the compiler optimize a loop while not forcing any specific implementation. This construct allows the compiler more freedom to choose a good implementation for a specific target than do other OpenMP directives.
- **Multilevel memory systems.** Memory allocation mechanisms are available that place data in different kinds of memories, such as high-bandwidth memory. New OpenMP features also make it easier to deal with the NUMA-ness of modern HPC systems.
- **Enhanced portability.** The declare variant directive and a new meta-directive allow programmers to improve performance portability by adapting OpenMP pragmas and user code at compile time.

THE END

OPENMP: Homework 6

Write, in parallel, using OpenMP

1. A "hello world"
2. A program that initializes 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. Make two versions:

   (a) Use do loops to do the matrix multiply and OpenMP PARALLEL DO

   (b) Use Fortran MATMUL and OpenMP workshare

   Run some tests for large matrices to see which one works best.

Submit a tar file *named with your name <name.tar>* that contains

- the .f90 files for your program
- A README of instructions on how to compile and run
- Sample output that shows that the programs work
- An analysis which shows which one works best for question 2