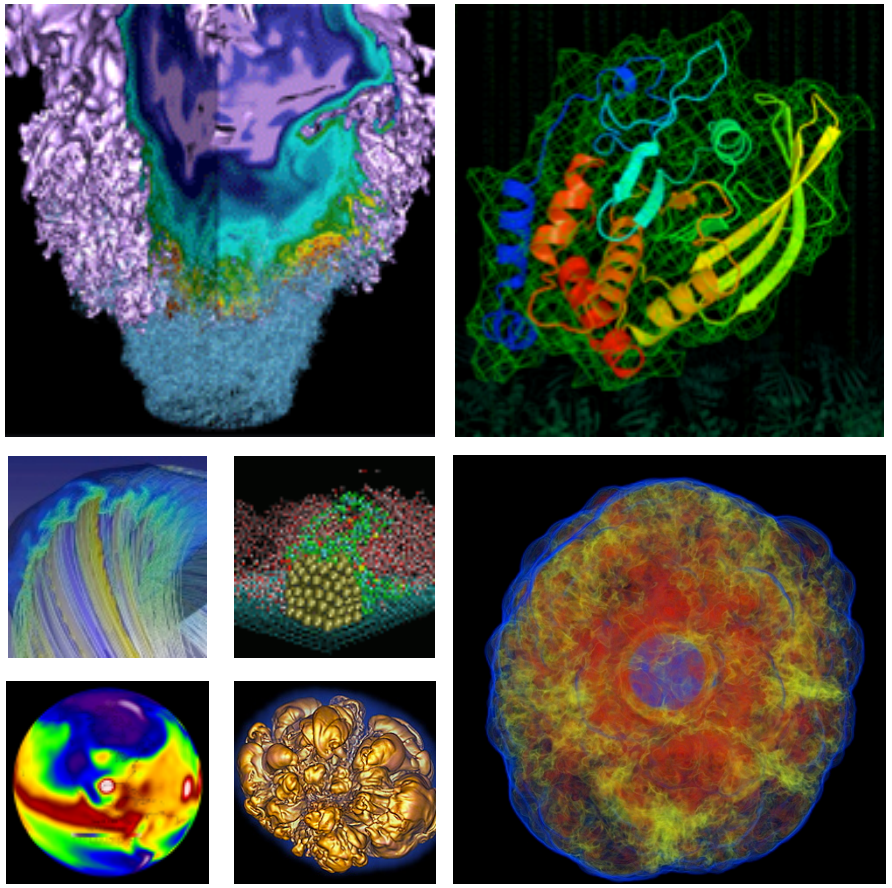
# Introduction to OpenMP Programming



NERSC

NERSC Staff

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

# Agenda

- **Basic information**
  - An selective introduction to the programming model.
  - Directives for work parallelization and synchronization.
  - Some hints on usage
- **Hands-on Lab**
  - Writing compiling and executing simple OpenMP programs.
- **Presentation available at**
  - `module load training`
  - `cp $EXAMPLES/NUG/Presentations/IntroToOpenMP.pptx`

# Agenda

- **New stuff**
  - Constructs introduced in OpenMP 3.0
    - Not tasking
- **Hands-on Lab**

# What is OpenMP?

- **OpenMP = Open Multi-Parallelism**
- **It is an API to explicitly direct *multi-threaded shared-memory parallelism*.**
- **Comprised of three primary API components**
  - Compiler directives
  - Run-time library routines
  - Environment variables

# Why use OpenMP?

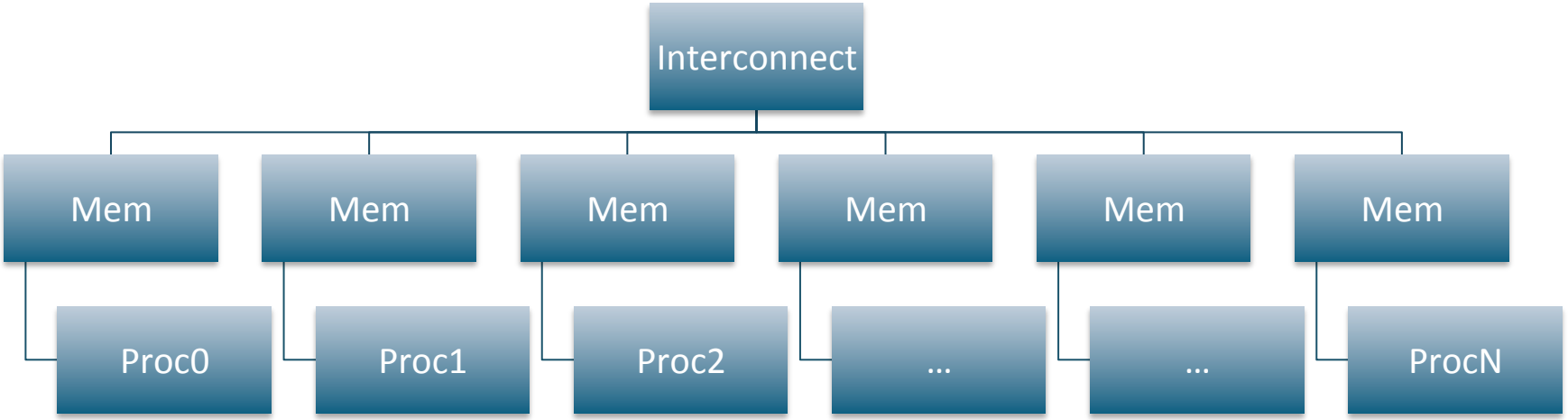
- **Moving to the many-core era, we are concerned with**
  - Reducing MPI communications
    - May improve run time
  - Improving scaling by exploiting
    - Fine-grained/Multi-level parallelism (e.g. loop level)
    - Parallelism not easily amenable to use of MPI
      - Graph algorithms
      - Master-slave work queues
  - Vectorization
    - New directives proposed **but** still should try to handle this yourself
  - Targeting new architectures
    - New directives proposed, bit of a wait and see

# How is OpenMP not MPI?

**MPI is an API for controlling *distributed-memory* parallelism on multi-processor architectures.**

**Each task has it's own unique memory**

**Information is passed between memory locations through the interconnect via the MPI API.**

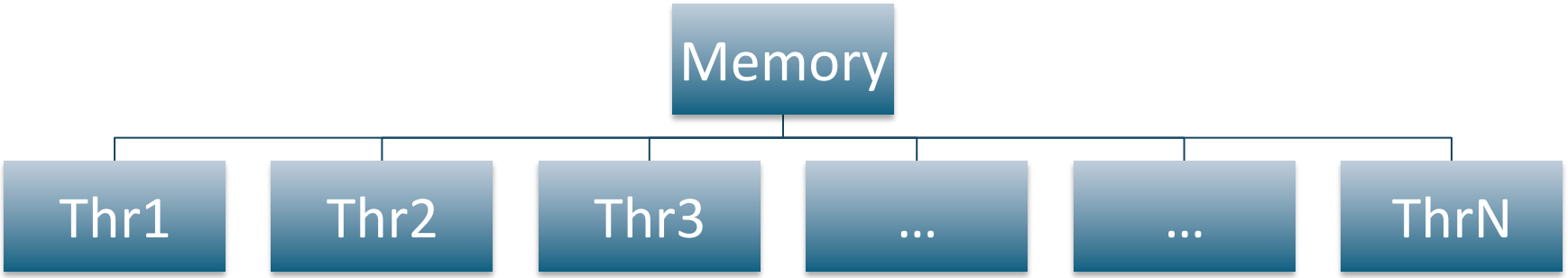


# OpenMP

A process, such as an MPI task, owns a lot of state information about the process, including the memory, file handles, etc. Threads, launched by the process, share the state information, including memory, of the launching process and so are considered *light weight processes*.

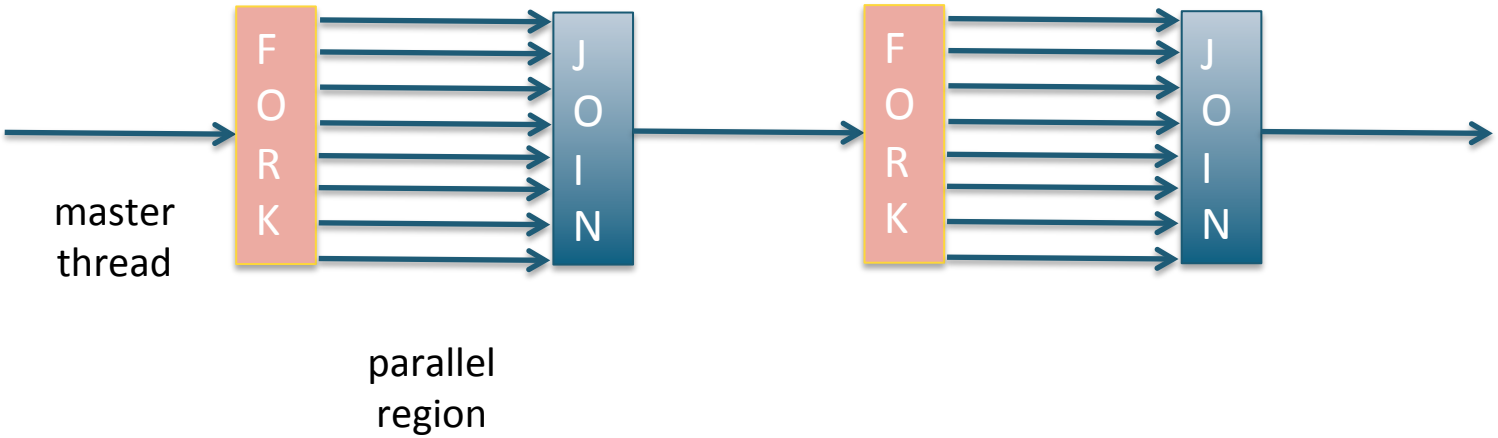
Since memory references amongst a team of threads are shared: *OpenMP requires that the programmer ensures that memory references are handled correctly*.

It is possible, for both paradigms to be used in one application to improve either speed, or scaling, or both. This is the so called *hybrid* parallel programming model.



# Fork-and-join model

- **OpenMP programs begin as a single process, the master thread, until they reach a parallel region, which then spawns a team of threads.**



# Creating parallelism

# Parallel regions

- **Directives (or *sentinels*) are comments (in Fortran) or pragmas (in C/C++). Thus, you can create portable code that works with or without OpenMP depending on the architecture or your available compilers.**
  - !$OMP directive  Fortran
  - #pragma omp directive  C/C++
- **Thread groups are created with the `parallel` directive**

# Fortran example

```
double precision :: x(1000)
integer id,n
!$omp parallel private(id)

   id = omp_get_thread_num()
   n  = omp_get_num_threads()
   call foo( id, x )

!$omp end parallel
```

- **Outside of parallel region, there is only 1 thread (master).**
- **Inside of parallel region there are N threads, N set by OMP_NUM_THREADS env var and aprun.**
- **All threads share X and call foo(), id is private to each thread.**
- **There is an implicit barrier at the end of the parallel region**

# Fortran example

- **In the previous example, we also saw two functions from the run time library**
  - `omp_get_thread_num()`
    - Returns unique thread id number for each thread in the team.
  - `omp_get_num_threads()`
    - Returns the number of threads in the team.
- **There are more (over 20) but these are the two most common, if they are used at all.**

# C example

```
double x[1000];

#pragma omp parallel
{
    int id = omp_get_thread_num();
    int n = omp_get_num_threads();
    foo( id, x );
}
```

# num_threads

- **Can also set the number of threads to execute a parallel section**

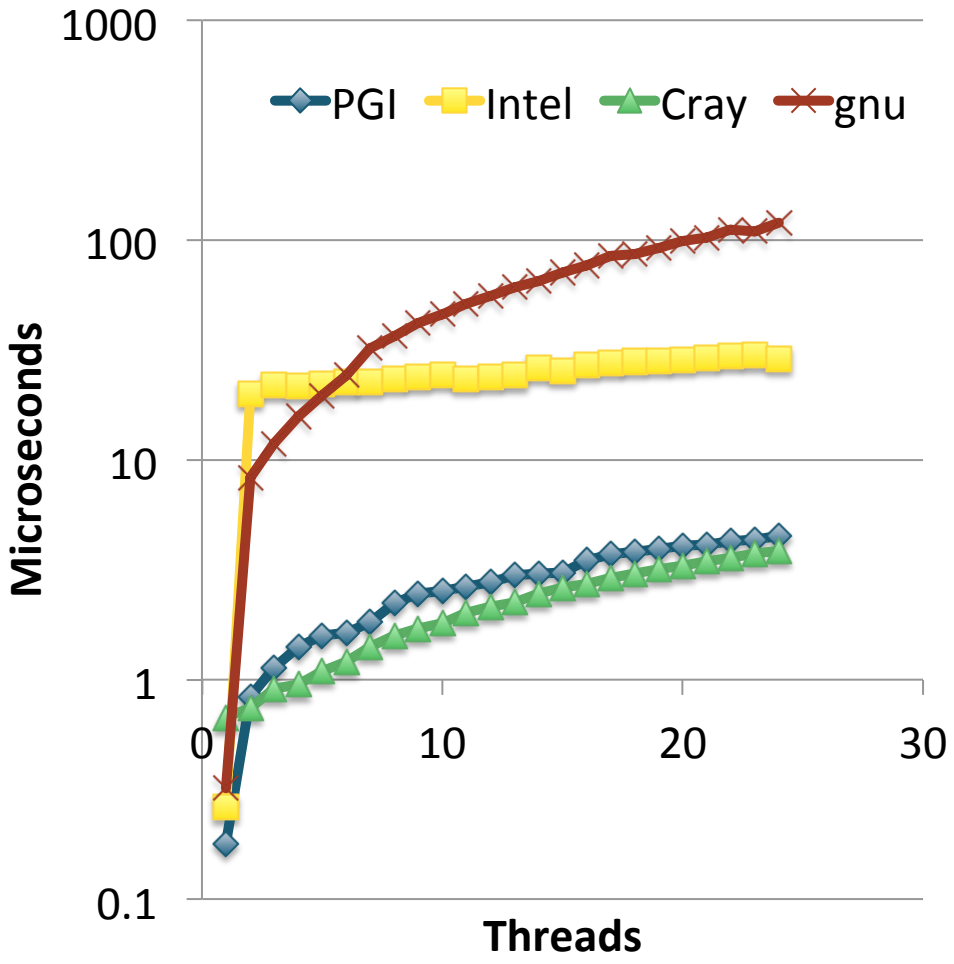
```
#omp parallel num_threads(N)
```

or

```
omp_set_num_threads(N);
#omp parallel
```

# Optimization Hint: Parallel Regions

- **Creating threads takes time.**
  - Reduce number of parallel regions by either
    - Encapsulating several parallel regions in a routine into one
    - Hoisting OpenMP parallel regions to a higher level



# Synchronization

- **Synchronization is used to impose order constraints and to protect shared data.**
  - Master
  - Single
  - Critical
  - Barrier

# master directive

```
!$omp parallel private(id)

   id = omp_get_thread_num()

!$omp master
   print *, 'myid = ', id
!$omp end master

!$omp end parallel
```

- **In this example, all threads are assigned a thread ID number (0-23, say).**
- **Because of the `master` directive, only the master thread (id=0) prints out a message.**
- **No implied barrier at end of the master region**

# single directive

```
!$omp parallel private(id)

   id = omp_get_thread_num()

!$omp single
   print *, 'myid = ', id
!$omp end single [nowait]

!$omp end parallel
```

- Again, all threads are assigned a thread ID number.
- Because of the **single** directive, only one thread prints out a message.
- Which thread executes the **single** section may change from one execution to the next.
- Implied barrier at end of single region => all threads wait!
- The optional **nowait** clause overrides the implicit barrier.

# critical directive

```
!$omp parallel private(id)

   id = omp_get_thread_num()

!$omp critical
   print *, 'myid = ', id
!$omp end critical

!$omp end parallel
```

- All threads will print their id number.
- Within the `critical` section, only one thread out of the team will be executing at any time.
- Thus, for six threads, there will be six print statements but they will not necessarily be ordered by id number.

# barrier directive

```
!$omp parallel
   call foo1()
!$omp barrier
   call foo2()
!$omp end parallel
```

- The `barrier` directive requires that all threads in the team arrive at the barrier before execution continues.
- In this example, the function foo1 may perform some action, e.g. on shared data, that may affect other threads in the function foo2. Thus, all threads execute foo1, stop at the barrier and then continue on to foo2.

# atomic directive

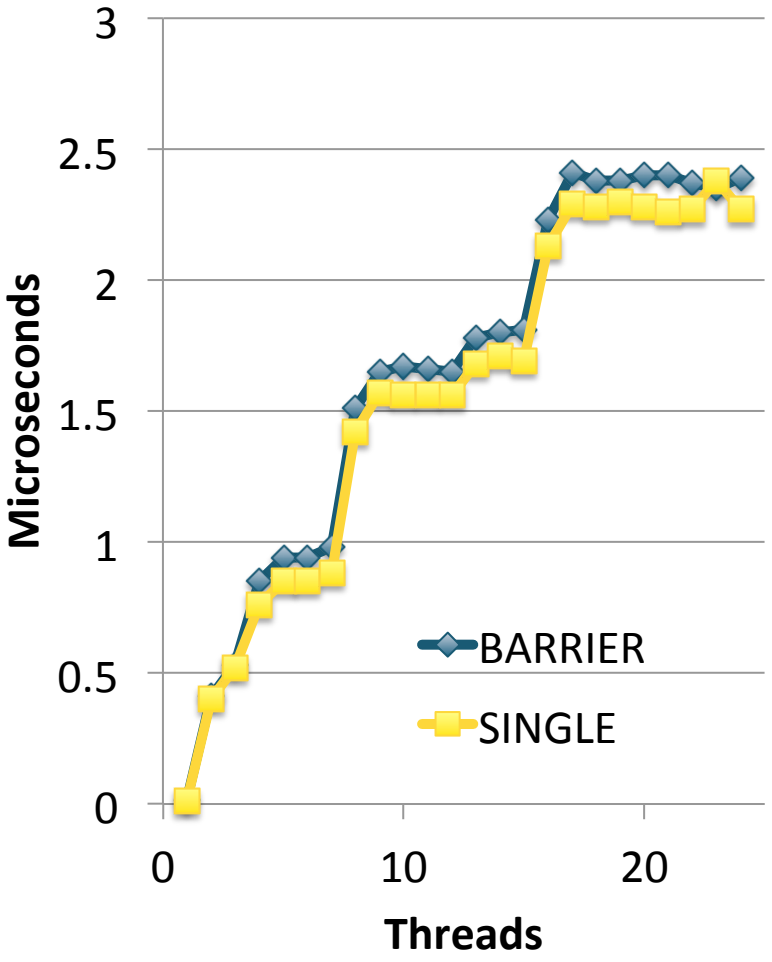
- The `atomic` protects memory locations from being updated by more than one thread.

```
n = 0
!$omp parallel

!$omp atomic
   n = n + 1

!$omp end parallel
```

OpenMP 3 implements several new `atomic` clauses, specifically: `read, write, update, capture`

# Optimization Hint: Barriers

- In general, try to avoid the use of sync/barrier directives, as they may cause significant performance degradation.
- If possible, try to re-factor your algorithm to avoid using them. Consider using temporary variables in to accomplish this.



# Data sharing

# Private/Shared Data

- **In parallel regions, several types of data attributes can exist**
  - `shared (default)`
    - Accessible by all threads
  - `private`
    - Accessible only by the current thread
    - NB: Loop counters are automatically private
- **Also**
  - `None`
  - `firstprivate/lastprivate/threadprivate`
- **The default can be changed using the `default` directive**

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

# Private/Shared data

- **Individual variables in parallel regions can be declared `private` or `shared`**

```
!$omp parallel private(x0,y0)
   x0 = xarray(…)
   y0 = yarray(…)
   f(…) = foo1(x0,y0)
!$omp end parallel
```

- **Here, `x0`, and `y0` are private variables, taken from the shared arrays `xarray()`, and `yarray()` that are used to compute some variable that is stored in the shared array `f()`.**
- **It is also possible to directly specify that variables be `shared`.**

```
!$omp parallel private(x0,y0) shared(xarray,yarray,f)
   x0 = xarray(…)
   y0 = yarray(…)
   f(…) = foo1(x0,y0)
!$omp end parallel
```

# firstprivate

- **The `firstprivate` directive allows you to set `private` variables to the master thread value upon entry into the parallel region.**

```
A = 1
B = 2
!$omp parallel private(A) firstprivate(B)
   ….
!$omp end parallel
```

- **In this example, A has an undefined value on entry into the parallel region while B has the value specified in the previous serial region.**
- **This can be costly for large data structures.**

# lastprivate

- **Specifies that the variable in the serial section of the code is set equal to the private version of whichever thread executes the final iteration(for/do loop) or last section (sections).**

```
A = 1
B = 2
!$omp parallel firstprivate(B)
!$omp do lastprivate(A)
do i = 1, 1000
   A = i
end do
!$omp end do
!$omp end parallel
```

- **In this example, upon exiting the do loop, A=1000.**

# threadprivate

- **Makes a private version of a global variable or common block for each thread.**
  - Outside of parallel regions, master thread version is referenced
  - Each thread gets its own copy so threads don't interfere with each other
  - Assume values are undefined unless a `copyin` clause is specified on `parallel` directive
  - Persist through multiple parallel regions, subject to restrictions

# threadprivate example

```
int a;
float x;

#pragma omp threadprivate(a, x)

main() {

#pragma omp parallel copyin(a,x)
{
….

}
```

# Optimization Hint: Scoping

- **If a function is called from a parallel region, local variables declared in that function are automatically private to the calling thread.**
  - Life might be easier if you moved your parallel region to a higher level.
    - Thread persistence, even if you don't use them
    - Software engineering is easier:
      - no need to declare private variables (very easy to get wrong and debug if you have a lot)

# Loop Worksharing (do/for)

- **The OpenMP worksharing construct do (in Fortran) or for (in C/C++) enables the programmer to distribute the work of loops across threads.**

```
!$omp parallel
!$omp do
DO I = 1, N
    a(i) = b(i) + c(i)
END DO
!$omp end do [nowait]
!$omp end parallel
```

  - **In this example, OpenMP determines, by default, the amount of work to give to each thread by dividing N by the number of threads. We will see later how to change this behavior.**
  - **Implicit thread synchronization point at end of DO section. Can change this with the `nowait` clause.**

# Loop worksharing

- **For convenience, the two statements can be combined**

```
!$omp parallel do
DO I = 1, N
    a(i) = b(i) + c(i)
END DO
!$omp end parallel do
```

# Reductions

- **Very often, a programmer needs to compute a variable that is the sum of other data, e.g.**

```
Real :: x(N), avg
Avg = 0.0
DO I = 1, N
    avg = avg + x(i)
END DO
Avg = avg / FLOAT(N)
```

- **This operation is called a reduction and there is support in OpenMP for parallelizing this sort of thing rather trivially.**

# reduction directive

```
real :: x(N), avg
!$omp parallel do reduction(+:avg)
DO I = 1, N
    avg = avg + x(i)
END DO
!$omp end parallel do
```

- **In this example, the `avg` variable is automatically declared `private` and initialized to zero.**
- **The general form of the reduction directive is**

```
reduction(operator:variable)
```

# Reductions

- **Some of the most common reduction operators and initial values are as follows**

| Operator | Initial value |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |

C/C++ Only

| Operator | Initial value |
|---|---|
| & | ~0 |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |

Fortran Only

| Operator | Initial value |
|---|---|
| MIN | Largest pos. number |
| MAX | Most negative number |
| .AND. | .TRUE. |
| .OR. | .FALSE. |
| .NEQV. | .FALSE. |
| .IEOR. | 0 |
| .IOR. | 0 |
| .IAND. | All bits on |
| .EQV. | .TRUE. |

# `ordered` directive

- **Some expressions in do/for loops need to be executed sequentially because the results are order dependent, e.g.**

```
DO I = 1, N
    a(i) = 2 * a(i-1)
END DO
```

- **In order to parallelize this loop, it is mandatory to use the `ordered` directive**

```
!$omp do ordered
DO I = 1, N
!$omp ordered
    a(i) = 2 * a(i-1)
!$omp end ordered
END DO
!$omp end do
```

Let OpenMP know an ordered statement is coming later

# `ordered` restrictions

- **Can only be used in a do/for loop**
  - If you have an `ordered` directive, you have to have an `ordered` clause on the do loop
- **Only one thread at a time in an ordered section**
- **Illegal to branch into/out of it**
- **Only one ordered section in a loop**

# Scheduling

- **When a do-loop is parallelized and its iterations distributed over the different threads, the most simple way of doing this is by giving to each thread the same number of iterations.**
  - not always the best choice, since the computational cost of the iterations may not be equal for all of them.
  - different ways of distributing the iterations exist, this is called scheduling.

# `schedule` directive

- **The `schedule` directive allows you to specify the *chunking* method for parallelization of `do` or `parallel do` loops. Work is assigned to threads in a different manner depending on the scheduling type or chunk size used.**
  - `static (default)`
  - `dynamic`
  - `guided`
  - `runtime`

# schedule directive

```
!$omp parallel do schedule(type[, chunk])
DO I = 1, N
   a(i) = b(i) + c(i)
END DO
!$omp end parallel do
```

- **The `schedule` clause accepts two parameters.**
  - The first one, `type`, specifies the way in which the work is distributed over the threads.
  - The second one,`chunk`, is an optional parameter specifying the size of the work given to each thread: its precise meaning depends on the type of scheduling used.

# `schedule` directive

- `static (default)`
  - work is distributed in equal sized blocks. If the chunk size is specified, that is the unit of work and blocks are assigned to threads in a round-robin fashion.
- `dynamic`
  - work is assigned to threads one at a time. If the chunk size is not specified, the chunk size is one.
  - Faster threads get more work, slower threads less.

# Sections

# `schedule` directive

- `guided`
  - Similar to dynamic but each block of work is a fixed fraction of the preceding amount, decreasing to `chunk_size` (1, if not set)
  - Fewer chunks = less synchronization = faster?
- `runtime`
  - Allows scheduling to be determined at run time.
  - Method and chunk size specified by the environment variable `OMP_SCHEDULE`, e.g.
    - `setenv OMP_SCHEDULE "guided, 25"`

# Sections

- Sections are a means of distributing independent blocks of work to different threads.
- For example, you may have three functions that do not update any common data

```
…
call foo1(…)
call foo2(…)
call foo3(…)
…
```

# Section directive

- **Using sections, each of these functions can be executed by different threads**

```
!$omp parallel
!$omp sections [options]
!$omp section
call foo1(…)            !thread 1
!$omp section
call foo2(…)            !thread 2
!$omp section
call foo3(…)            !thread 3
!$omp end sections[nowait]
!$omp end parallel
```

# Sections

- **May be the only way to parallelize a region.**
- **If you don't have enough sections, some threads my be idle.**
  - Still may be useful and provide a performance boost if you can't thread your blocks or functions.
- **Can also use `!$omp parallel sections` shortcut.**

# Workshare

# Workshare (Fortran only)

- **In Fortran, the following can be parallelized using the `workshare` directive**
  - `forall`
  - `where`
  - Array notation expression
    - e.g. `A = B + C`, where `A`, `B`, and `C` are arrays.
  - Transformational array functions
    - e.g. `matmul`, `dot_product`, `sum`, `maxval`, `minval`, etc.

# workshare example

```
real(8) :: a(1000), b(1000)
!$omp parallel
!$omp workshare

A(:) = a(:) + b(:)

!$omp end workshare[nowait]
!$omp end parallel
```

**Each thread gets a chunk of the iteration space of the arrays.**

# Useful links

- **OpenMP Consortium (www.openmp.org)**
  - http://www.openmp.org/mp-documents/OpenMP3.0-FortranCard.pdf
  - http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf

# Lab 1

- **You can use your personal or course account**
- **`ssh` to hopper/edison**
- **`module load training`**
- **Copy homework problems from**
  - $EXAMPLES/NUG/OpenMP/openmp_lab.tar
- **Load your compiler if necessary**
- **Edit ENV file to pick C or Fortran compiler and OpenMP flag**
- **Each exercise has a build and run script**
  - ./runit N
    - N = number of threads.
    - Script compiles code, creates a batch script, and launches a batch job.

# Lab 1

- **Exercise 1: Parallel loop with reduction.**
  - Program integrates a function to determine pi.
  - Parallelize the loop and use a reduction.
  - Determine speedup for several thread counts.
- **Exercise 2: Worksharing and sections**
  - Use worksharing to parallelize array constructs
  - Use Sections to parallelize functional calls
  - Determine speedup for several thread counts.

# Lab 1

- **Exercise 3: Simple matrix-matrix multiply.**
  - Parallelize the initializations using sections.
  - Parallelize the multiply
  - Introduces use of omp_get_thread_num()
    - Must reside in the $omp parallel section
  - Determine speedup for various thread counts.

# OpenMP 3.0 features

- OMP_STACKSIZE
- Loop collapsing
- Nested parallelism

# OMP_STACKSIZE

- **`omp_stacksize` *`size`***
- **Environment variable that controls the stack size for threads.**
  - Valid sizes are *size*, *sizeB*, *sizeK, sizeM*, *sizeG* bytes.
  - If B, K, M, G not specified, size is in kilobytes(K).

# Collapse(n)

- **Clause for do/for constructs**
- **Specifies how many loops in a nested loop should be collapsed into one large iteration space.**

```
!$omp parallel do collapse(2)
DO K = 1, N1
   DO J = 1, N2
      DO I = 1, N3
         a(i,j,k) = b(i,j,k) + c0(c(i,j,k)
      END DO
   END DO
END DO
!$omp end parallel do
```

**NB: Collapsing down to the innermost loop might inhibit compiler optimizations.**

# Nested Parallelism

- **It is possible to nest parallel sections within other parallel sections**

```
!$omp parallel
   print *, 'hello'
!$omp parallel
   print *, 'hi'
!$omp end parallel
!$omp end parallel
```

- Can be useful, say, if individual loops have small counts which would make them inefficient to process in parallel.

# Nested parallelism

- **Nested parallelism needs to be enabled by either**
  - Setting an environment variable
    - `setenv OMP_NESTED TRUE`
    - `export OMP_NESTED=TRUE`
  - Using the OpenMP run-time library function
    - `call omp_set_nested(.true.)`
- **Can query to see if nesting is enabled**
  - `omp_get_nested()`
- **Set/Get Number of maximum active levels**
  - omp_set_max_active_levels (int max_levels)
  - OMP_MAX_ACTIVE_LEVELS=N

# Nested Parallelism

- **Warnings**
  - Remember overhead from creating parallel regions.
  - Easy to oversubscribe a node if you don't pay attention.
  - May cause load imbalance

# Caveat Emptor

- **Section 1.1 of OpenMP spec**
  - *OpenMP-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs.*
  - *… compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming.*
  - *The user is responsible for using OpenMP in his application to produce a conforming program.*

# Caveat Emptor

- **Be careful of the use of 'orphaned' directives.**
  - Good example
    - Parallel region in foo1() that calls foo2() that has an OpenMP for/do construct.
  - Bad example
    - Parallel ordered do loop with an ordered section that calls another function that has an orphaned ordered directive.
    - Violates having more than one ordered section in a loop but compiler doesn't see it and so the behavior is 'undefined'.

# Exercises left to the reader:

- **Environment variables: 9**
- **Run time functions: 32 (incl locks!)**
- **Tasks!**
- **Optimization**
  - Thread affinity
  - Memory: NUMA effects/false sharing
- **Hazards**
  - Race conditions
  - Dead/Live locking
- **OpenMP 4**
  - Processor binding
  - SIMD directives (loops, function declarations)

# Lab 2

- **Exercise 4: Collapse directive**
  - Parallelize the two loops structures. Get some timings for various thread counts.
  - Insert collapse(2) directive for both loops. Note effect for different thread counts.