COMP 364 / 464
High Performance Computing

OpenMP:

Multi-threaded and task parallelism

Computing Terminology

Terms	Definition
• NUMA	• Non Uniform Memory Access. In SMP systems with multiple CPUs, access time to different parts of memory may vary
• Affinity	• Propensity to maintain a process or thread on a hardware execution unit
• SMP	• Symmetric Multi-Process(ing/or). Single OS system with shared memory
• OpenMP	
– Directive	• Comment statement (F90) or #pragma (C/C++) that specifies parallel operations and control
– Construct	• The lexical extent that a directive controls
– Region	• All code controlled by a directive– lexical extent + content of called routines
• Runtime	• Code or a library within an executable that interacts with the operating system and can control code execution

What is OpenMP (*Open Multi-Processing*)

- De-facto standard for high-performance parallel programming on *Symmetric Multi-Processor* (SMP) Systems
- It is an API (*Application Program Interface*) for designing and executing parallel Fortran, C, and C++ programs
    - Based on threads, but
    - Higher level than POSIX threads (*Pthreads*) http://www.llnl.gov/computing/tutorials/pthreads/#Abstract
- Implemented by:
    - Compiler Directives (or #pragma’s)
    - Runtime Library (interface to OS and Program Environment)
    - Environment Variables
- Compiler option required to interpret/activate directives
- http://www.openmp.org/ has tutorials and description
- Directed by OpenMP ARB (*Architecture Review Board*)

OpenMP -- Overview

- Standard developed in the late 1990s
- The “language” is easily grasp. You can start simple and expand.
- Lightweight from system perspective
- Very portable – GNU GCC and vendor (Intel) compilers.
- OMP compilers generate the parallel code but developers must define the parallelism and communication mechanisms.
  - Time spent finding parallelism can be the most difficult part. The parallelism may be hidden.
  - Writing parallel OpenMP code examples is relatively easy.
  - Developing parallel algorithms and/or parallelizing existing serial code is much harder.
  - Expert level requires awareness of scoping and synchronization.

OpenMP Executable Runs on an SMP*

- Shared Memory Systems:
  - One Operating System
  - Instantiation of ONE process
  - Threads are forked (created) from within your program
  - Multiple threads on multiple cores

* SMP = Symmetric Multi-Processor: The execution of the operating system has equal access to any of the “processors”

OpenMP History

- Primary OpenMP participants

  AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, PGI, Oracle, MS, TI, CAPS, NVIDIA
  ANL, LLNL, cOMPunity, EPCC, LANL, NASA, ORNL, RWTH, TACC

• OpenMP Fortran API, Version 1.0	Published October 1997
• OpenMP C API, Version 1.0	Published October 1998
• OpenMP 2.0 API for Fortran	Published 2000
• OpenMP 2.0 API for C/C++	Published 2002
• OpenMP 2.5 API for C/C++ & F90	Published 2005
• OpenMP 3.0 Tasks	Published May 2008
• OpenMP 3.1	Published July 2011
• OpenMP 4.0 Affinity, Offload, SIMD	Published July 2013
• OpenMP 4.5	Published Nov 2015

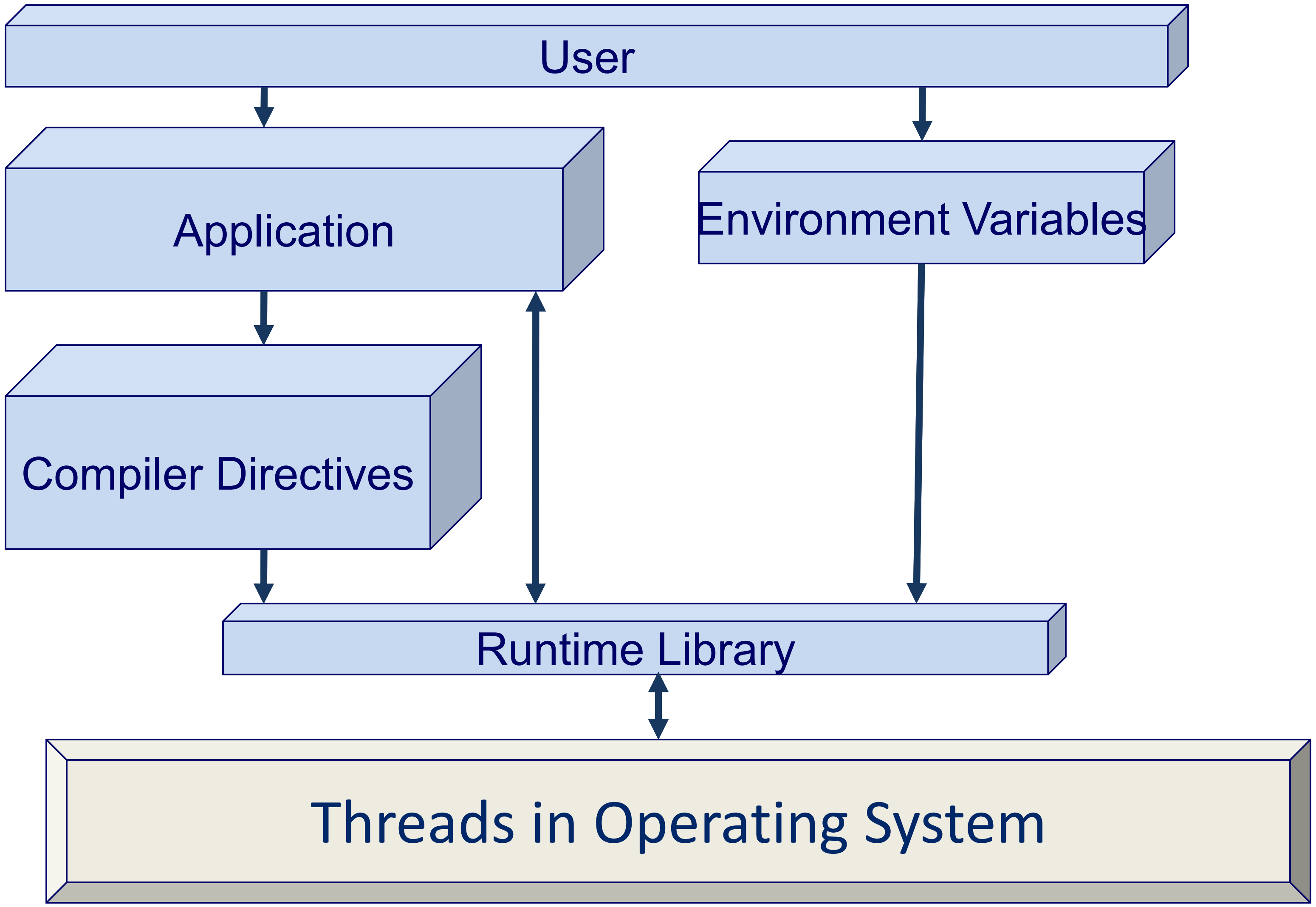
Advantages/Disadvantages of OpenMP

- Pros
  - Shared Memory Parallelism is easier to learn than distributed MP
  - Coarse-grained or fine-grained parallelism
  - Parallelization can be incremental
  - Widely available, portable
  - Converting serial code to OpenMP parallel
    can be easier than converting to multi-process, distributed parallel
  - SMP hardware is ubiquitous now: Supercomputers **and** your desktop/laptop
- Cons
  - Scalability limited by memory architecture:
    - Non-uniform memory access (NUMA) can be devastating.
    - Thread synchronization dependent upon fast access to global variables – in main memory.
  - Available on SMP systems “only”
  - Beware: “Upgrading” large serial code may be time-consuming … but it always will be!
  - Not directly part of the language.

OpenMP Parallel Directive

- Supports parallelism by Directives (Fortran) and Pragmas (C/C++)
- Unlike others that require base language changes and constructs (Pthreads)
- Unlike MPI which supports parallelism through communications library between processes.

OpenMP Architecture



OpenMP Syntax

- OpenMP Directives: **sentinel**, **construct** and **clauses**

```
C      #pragma omp construct [clause [[,]clause]…]
F90    !$omp       construct [clause [[,]clause]…]
```

- Example

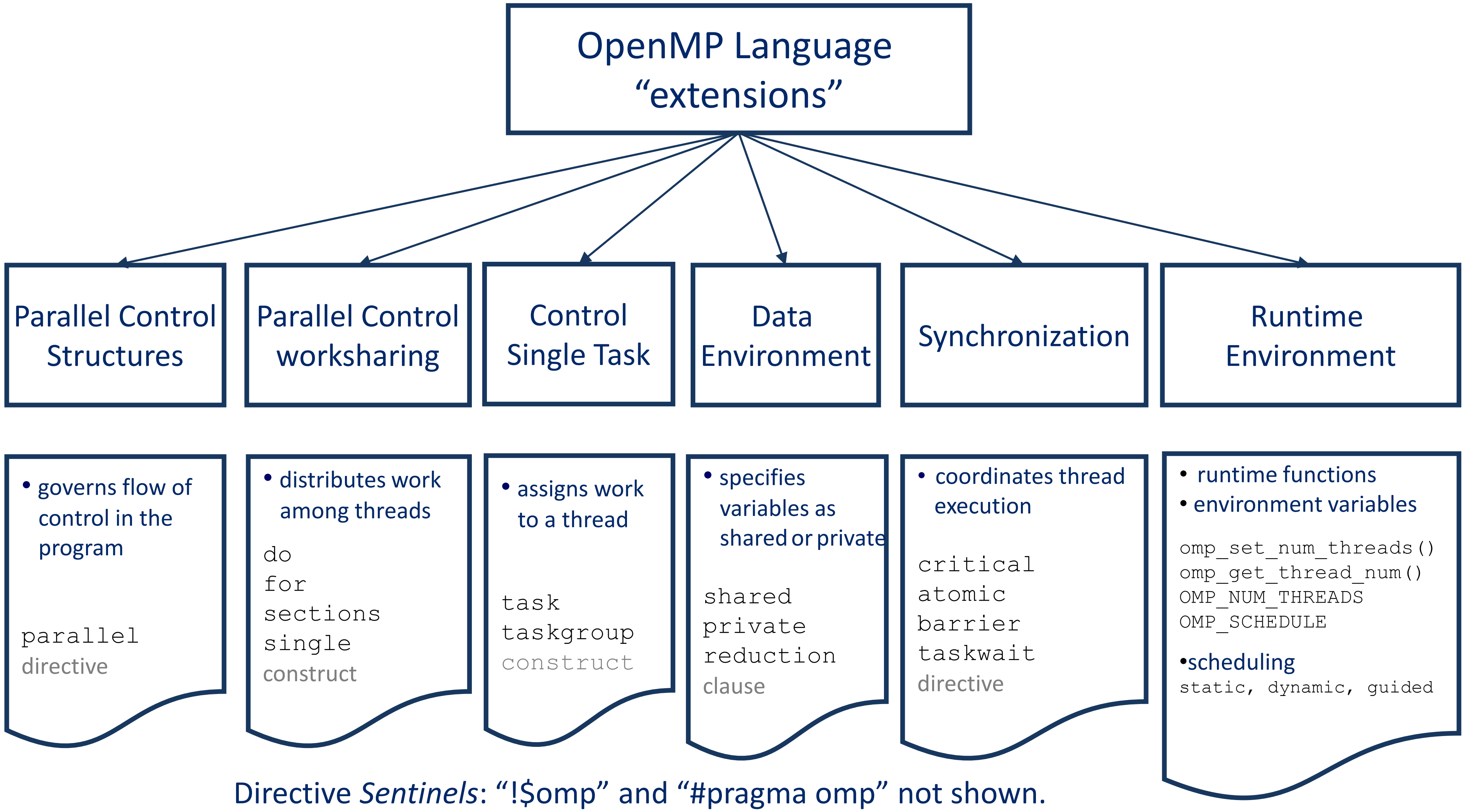
```
C      #pragma omp  parallel num_threads(4)
F90    !$omp        parallel num_threads(4)
```

- Function prototypes and types are in the file:

```
C      #include <omp.h>
F90    use omp_lib
```

- Most OpenMP constructs apply to a “structured block”, that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom

OpenMP Constructs



Directive *Sentinels*: “!$omp” and “#pragma omp” not shown.

OpenMP Directives

- OpenMP directives begin with special comments/pragmas that a OpenMP-aware compiler can interpret. Directive sentinels are:

F90	`!$OMP`
C/C++	`# pragma omp`

Syntax: *sentinel* parallel *clauses* *uses defaults when clauses not present*

Fortran

```
!$OMP parallel
  ...
!$OMP end parallel
```

C/C++

```
# pragma omp parallel
   {...}
```

- Fortran parallel regions are enclosed by enclosing directives.
- C/C++ parallel regions are enclosed by curly brackets.

Parallel Region & Thread Number

```
1 #include <omp.h>
2 …
3 int ierr = 0;
4 #pragma omp parallel
5 {
6     int numThreads = omp_get_num_threads();
7     int threadId = omp_get_thread_num();
8     ierr = do_work( threadId, numThreads );
9 }
```

Every thread can inquire the total number of threads (**`numThreads`** in line 6) and get a unique value for the thread number [0,K-1)

Parallel Region & Thread Number

```c
#if defined(_OPENMP)
# include <omp.h>
#endif
…
int ierr = 0;
#pragma omp parallel
{
    int numThreads = 1;
#if defined( _OPENMP)
    numThreads = omp_get_num_threads();
#endif
    ierr = work(numThreads);
}
```

OpenMP code can be disabled and ignored by the compiler. All sentinels and directives ignored when OMP isn't enabled by compiler. Pre-processor (macro) flag useful to remove OMP API function calls and header. _OPENMP automatically defined when OMP enabled.

Parallel Region & Worksharing

Use OpenMP directives to specify Parallel Region, *worksharing* constructs, and mutual exclusion

```
parallel


end parallel
```

Use parallel … end parallel for F90
Use parallel {…} for C

Code block	Each Thread Executes
`for`	Worksharing: splice loop
`sections`	Worksharing: splice calls
`single`	Only one thread (first there)
`master`	Only master thread (0)
`critical`	Execute one thread at a time
`atomic`	Update one thread at a time

```
parallel for
parallel sections
```

A single worksharing construct (e.g. **`for`**) may be combined on a parallel directive line.

Parallel Region

```
...
#pragma omp parallel
{
  #pragma omp for
  for ( i = 0; i < n; i++){
     work(i);
  } // <- implicit barrier
} // <- implicit barrier
```

- In above example the **`for`** loop iterations are split among the threads via the **`for`** worksharing construct.
- Which iteration(s) are executed by which threads? Controlled by **`schedule`** construct. Default is to split the iterations into K=#threads chunks, each n/K long.

OpenMP Combined Directives

- Combined directives
  - **`parallel for`** and **`parallel sections`**
  - Same as parallel region + one **`for`** worksharing construct

```
#pragma omp parallel for
for (i= 0; i< 100; i++){
   a[i] = b[i];
} // <- implicit barrier
```

- Fixed (known) trip count and increment required
- no break
- limited C++ throw
- continue ok (since it doesn't impact other iterations)
- Basic rule of thumb: if it can be done with SIMD, it can be threaded with OpenMP-for

Parallel Region

worksharing (WS) constructs: **`for`**, **`sections`**, and **`single`**

- WS Threads execution their “share” of statements in a PARALLEL region.
- **`for`** worksharing may require run-time work distribution and scheduling

```
1 #pragma omp parallel for
2 for (i= 0; i< n; i++){
3     a[i] = b[i] + c[i];
4 } // <- implicit barrier
```

Line 1: Team of threads formed (parallel region).
Line 2-4: Loop iterations are split among threads.
Implied barrier at “**}**”.

Each loop iteration must be independent of other iterations!

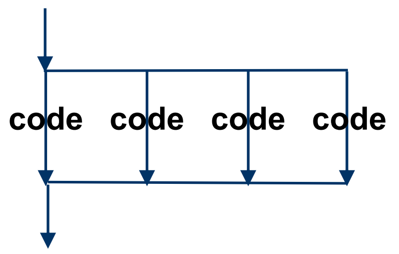
Replicated vs. Workshare Constructs

- Replicated: Work (code) blocks are executed by all threads.
- Worksharing: Work is divided among threads … no overlap.

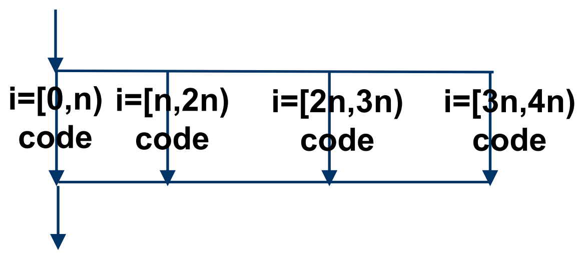
```
#pragma omp parallel
   {code}
```

```
#pragma omp parallel for
for (i=0; i<4*n; i++)
     {code}
```

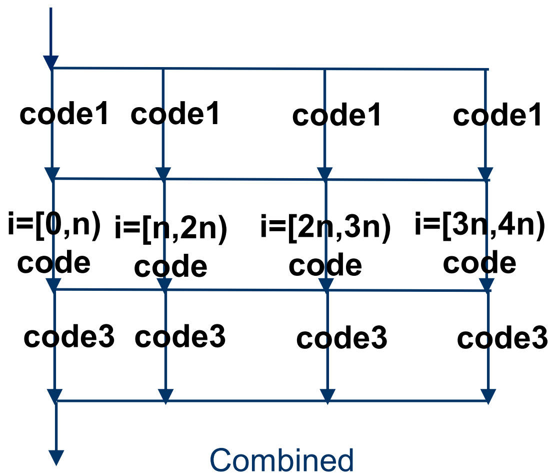
```
#pragma omp parallel
{
   code1();
   #pragma omp for
   for (i=0; i<4*n; i++)
      code();
   code3();
} // end parallel
```



Replicated



worksharing



Combined

OpenMP Worksharing Scheduling

Clause Syntax: **`parallel for schedule(schedule-type[,chunk-size])`**

Schedule Type

Schedule (**`static, chunk`**)

- Threads receive chunks of iterations in thread order, round-robin. (Divided "equally" if no chunk size … N / p)
- Good if every iteration contains same amount of work … uniform workload.
- May help keep parts of an array in a particular processor's cache– good between **`parallel for`**'s.

Schedule (**`dynamic, chunk`**)

- Thread receives chunks as it (the thread) becomes available for more work … a queue.
- Default chunk size may be 1
- Good for load-balancing non-uniform workloads.

OpenMP Worksharing Scheduling

Schedule (**`guided, chunk`**)

- Thread receives chunks as the thread becomes available for work.
- Chunk size decreases exponentially, until it reaches the chunk size specified (default is 1).
- Balances load and reduces number of requests for more work.
- (*I have never found this useful*.)

Schedule (**`runtime`**)

- Schedule is determined at run-time by the OMP_SCHEDULE environment value.
- Useful for experimentation

OpenMP Worksharing Scheduling

For example, loop with 100 iterations and 4 threads

- schedule (`static`)

Thread	0	1	2	3
Iteration	1-25	26-50	51-75	76-100

- schedule (`dynamic, 15`) *(one possible outcome)*

Thread	0	1	3	2	1	3	2
Iteration	1-15	16-30	31-45	46-60	61-75	76-90	90-100

  – Dynamic scheduling has overhead since assigning the "next" chunk requires thread communication (access to shared data object)

OpenMP worksharing -- Sections

- `sections`
  - Blocks of code are split among threads - task parallel style
  - A thread might execute more than one block or no blocks
  - Implied barrier

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      TASK1(…);
    #pragma omp section
      TASK2(…);
    #pragma omp section
      TASK3(…);
  } // <- implicit barrier
} // <- another barrier at the end of the parallel region
```

OpenMP worksharing - Single

- **`single`** (or master)
  - Block of code is executed only by the 1st thread (single) or rank-0 (master)
  - Implied barrier at '}' after single … no barrier after master!

```
int global_count = 0; // Defined globally.
…
#pragma omp parallel shared( global_count )
{
    foo1();
    #pragma single
    {
        global_count++;
        printf("%d\n", global_count);
    } // <- barrier
    foo2();
}
```

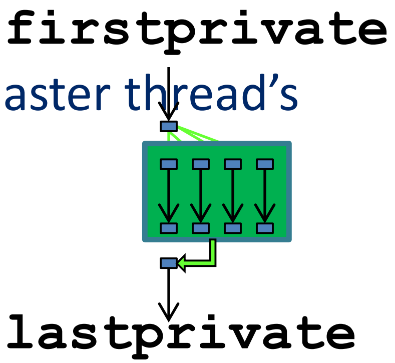
OpenMP Data scoping

`#pragma omp directive-name [clause [ [,]clause]…]`

- **`private`** (variable list)
  - Each thread has its own copy of the specified variable
  - Variables are undefined after worksharing region
- **`shared`** (variable list)
  - Threads share a single copy of the specified variable
- **`default`** (type)
  - A default of **`private`**, **`shared`**, or **`none`** can be specified
  - Note that loop counter(s) of worksharing constructs are always **`private`** by default; everything else is **`shared`** by default

OpenMP Data Scoping

- **firstprivate** (variable list)
  - Like **private**, but copies are initialized with the master thread's value
- **lastprivate**(variable list)
  - Like **private**, but final value is copied out to master thread's copy
  - **for**: last iteration; **sections**: last section
- **reduction** (operator : variable)
  - Each thread has its own copy of the specified variable
  - Can appear only in reduction operation
  - All copies are "reduced" back into the original master thread's variable.
  - Example **reduction(+:sum)** adds all values together.



OpenMP Data Scoping

- `for` and `parallel for` constructs
  - index variable is automatically private
- Automatic storage variables
  - private, if declared in a scope inside the construct (e.g. ordinary local variables declared inside functions)

OpenMP worksharing - Single

- **`shared`** - Variable is shared (seen) by all processors.
- **`private`** - Each thread has a private instance of the variable.
- Defaults: All loop indices are private, all other variables are shared.

```
double t1, t2;
int i;
#pragma parallel for shared(a) private(t1,t2)
for (i = 0; i < 1000; i++){
   t1 = f[i];
   t2 = g[i];
   a[i] = sqrt( t1*t1 + t2*t2 );
}
```

OpenMP worksharing - Single

- **`shared`** - Variable is shared (seen) by all processors.
- **`private`** - Each thread has a private instance of the variable.
- Use local scoping whenever possible! It's a lot safer and unambiguous.

```
#pragma parallel for shared(a)
for (int i = 0; i < 1000; i++){
    double t1 = f[i];
    double t2 = g[i];
    a[i] = sqrt( t1*t1 + t2*t2 );
}
```

OpenMP Data Scoping

```
int sum = 0;
#pragma omp parallel for reduction( + : sum )
for (int i = 0; i < N; i++){
   sum = sum + a[i];
}
// Each thread's copy of sum is added
// to original sum at end of loop
printf("sum= %f\n",sum);

#pragma omp parallel for lastprivate( temp )
for (int i = 0; i < N; i++){
   temp = f[i];
}
printf("f[N-1] == %f\n", temp);
// temp is equal to f[N-1] at end of loop
```

OpenMP Synchronization

- `nowait` clause
  - Threads encounter a barrier synchronization at end of worksharing constructs.
  - Specifies that threads can proceed to the next task without waiting.

```
#pragma omp    ...    nowait
```

OpenMP Synchronization

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i=0; i<N; i++){
     b[i] = foo(a[i]);
  }
  // Threads can start foo2() before others finish foo()

  #pragma omp for
  for (i=0; i<N; i++){
     d[i] = foo2(c[i]);
  }
}
```

– Why? This avoids unnecessary synchronization … which is overhead.

– Serial overhead increases the serial ratio in Amdahl’s limit.

OpenMP Synchronization

- **`barrier`** directive explicitly adds synchronization point
  - All threads must reach the barrier and wait until the last gets there.

```
#pragma omp barrier
```

OpenMP Synchronization

```
#pragma omp parallel
{
   const int thread_id   = omp_get_thread_num(),
             num_threads = omp_get_num_threads();
   const int N_per_thread = N / num_threads;
   int istart = thread_id * N_per_thread,
         iend = istart + N_per_thread;

   for (i = istart; i < iend; i++)
      b[i] = foo( a[i] ); // Fixed cost per iteration

   // Don't let any threads past this point yet to avoid race
   // scenario on b[].
   #pragma omp barrier

   #pragma omp for schedule(dynamic) nowait
   for (int i = 0; i < N; i++){
      b[i] += foo2( c[i] );
}
// All threads block (or sync) here implicitly.
```

OpenMP Synchronization

- `critical` section permits access to only one thread at a time
  - All threads execute the critical section but they access the section with mutual exclusion … only one can be inside the code at a given time.
  - This avoids race conditions but is very expensive.

```
#pragma omp critical
```

OpenMP Synchronization

```
InitializeQueue( allWorkItems ); // Initialize all work items serially.
#pragma omp parallel
{
   bool notEmpty = true; // This is a private variable for each thread.
   while ( notEmpty )
   {
      workItemType workItem;

      // Pop a task off the queue one thread at a time. This avoids
      // race condition on queue state.
      #pragma omp critical
      {
         notEmpty = PopQueue( &workItem );
      }

      // Do some work (if necessary)
      if ( notEmpty )
         foo(workItem);
   }
} // <= All threads block (or sync) here.
```

OpenMP Synchronization

- **`atomic`** statements insure that only one threads reads / writes / updates a shared variable memory location at a time.
  - These are generally faster than critical sections since they can be supported in hardware. And, if there isn't a conflict with another thread, this has little overhead.
  - Therefore, this is a cheap (scalable) strategy to avoid race conditions but is limited to simple statements.

```
#pragma omp atomic [read,write,capture]
```

OpenMP Synchronization

```
int nWorkItems = 4000, WorkItemCounter = 0;
WorkItemType workItem[] = InitializeItems(nWorkItems);
#pragma omp parallel shared( nWorkItems, workItemCounter )
{
   int myWorkItem = omp_get_thread_num();

   while (myWorkItem < nWorkItems)
   {
      foo( workItem[myWorkItem] );

      // copy the shared counter and then increment.
      #pragma omp atomic capture
      myWorkItem = WorkItemCounter++;
   }
} // <= All threads block (or sync) here.
```

– This is (usually) faster than a queue with a critical section since atomics are pretty quick.
– This is a way of manually coding a dynamic parallel for loop.