



Introduction to OpenMP

Le Yan

HPC Consultant
User Services





Goals

- Acquaint users with the concept of shared memory parallelism
- Acquaint users with the basics of programming with OpenMP
- Discuss briefly the topic of MPI+OpenMP hybrid programming





Memory System: Shared Memory

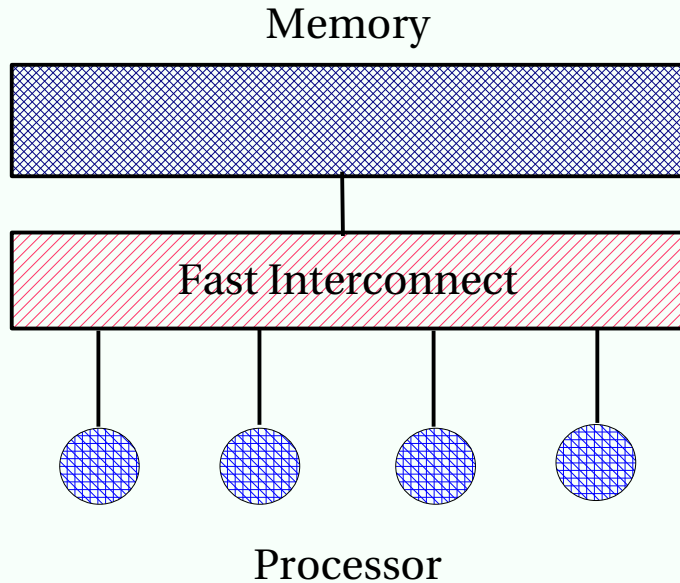
- Global memory space, accessible by all processors
- Data sharing among processors achieved by reading/writing to the same memory location
- Most commonly represented by Symmetric Multi-Processing (SMP) systems
 - Identical processors
 - Equal access time to memory
- Large shared memory systems are rare, clusters of SMP nodes are popular



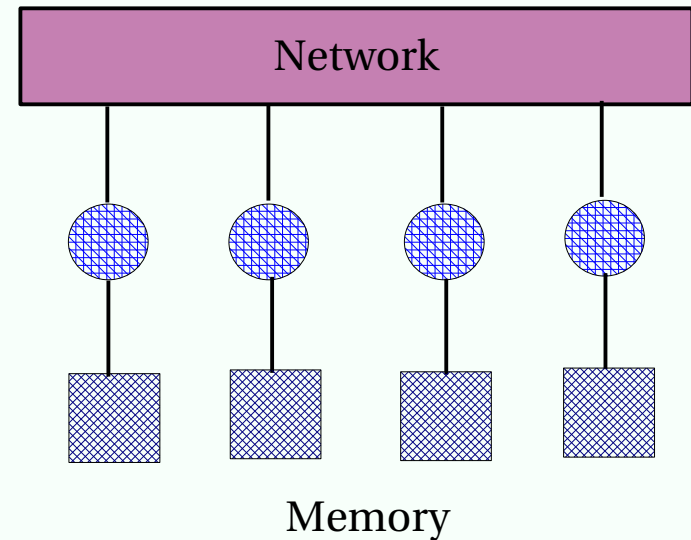


Shared vs. Distributed

Shared memory



Distributed memory





Shared vs Distributed

Distributed Memory

- Pros
 - Memory scalable with number of number of processors
 - Easier and cheaper to build
- Cons
 - Difficult load balancing
 - Data sharing is slow

Shared Memory

- Pros
 - Global address space is user-friendly
 - Data sharing is fast
- Cons
 - Lack of scalability
 - Data conflict issues





OpenMP

- OpenMP has been the industry standard for shared memory programming over the last decade
 - Permanent members of the OpenMP Architecture Review Board: AMD, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, PGI, SGI, Sun
- OpenMP 3.0 was released in May 2008
- OpenMP is an Application Program Interface (API) for thread based parallelism; Supports Fortran, C and C++
- Uses a fork-join execution model
- OpenMP structures are built with program directives, runtime libraries and environment variables





Advantages of OpenMP

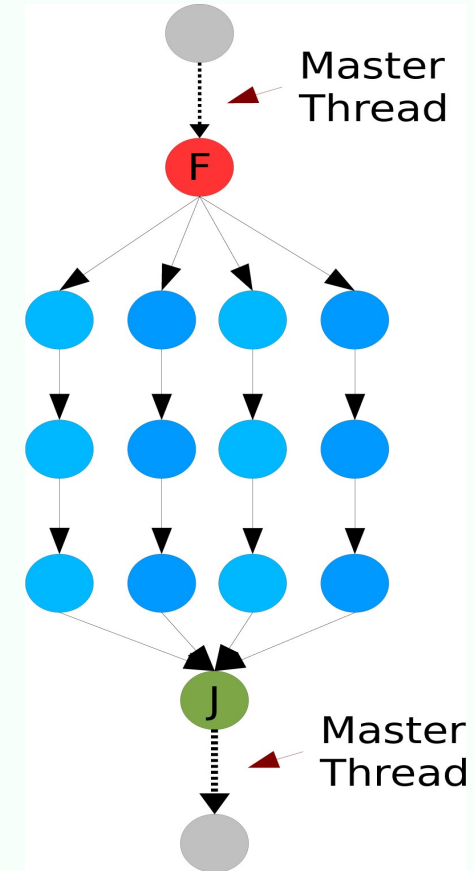
- Portability
 - Standard among many shared memory platforms
 - Implemented in major compiler suites
- Ease to use
 - Serial programs can be parallelized by adding compiler directives
 - Allows for incremental parallelization – a serial program evolves into a parallel program by parallelizing different sections incrementally





Fork-join Execution Model

- Parallelism is achieved by generating multiple threads that run in parallel
- A fork is when a single thread is made into multiple, concurrently executing threads
- A join is when the concurrently executing threads synchronize back into a single thread
- OpenMP programs essentially consist of a series of forks and joins.





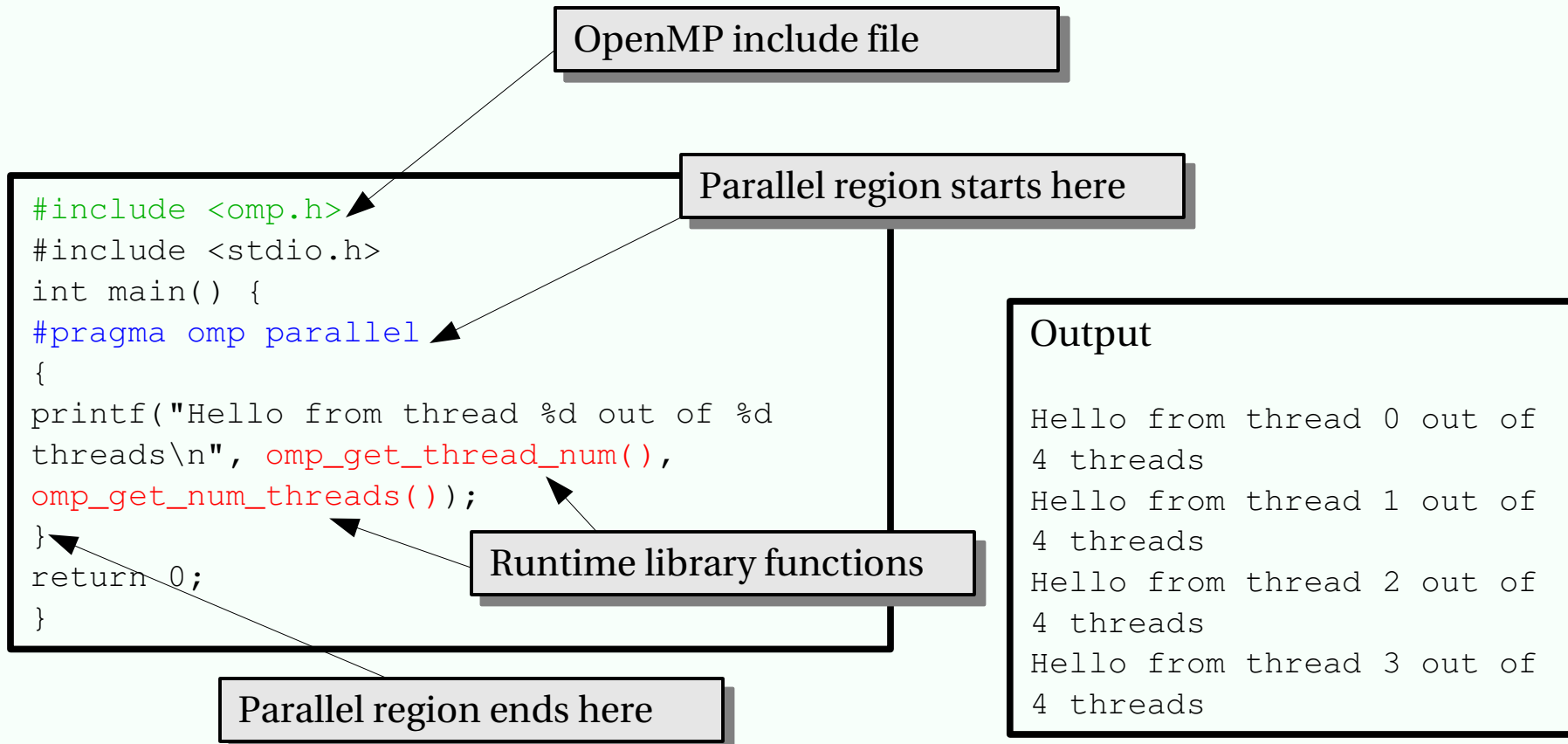
Building Blocks of OpenMP

- Program directives
 - Syntax
 - C/C++: `#pragma omp <directive> [clause]`
 - Fortran: `!$omp <directive> [clause]`
 - Parallel regions
 - Parallel loops
 - Synchronization
 - Data structure
 - ...
- Runtime library routines
- Environment variables





Hello World: C





Hello World: Fortran

Parallel region starts here

```

program hello
  implicit none
  integer omp_get_thread_num, omp_get_num_threads
  !$omp parallel
  print *, 'Hello from thread', omp_get_thread_num(), 'out
of', omp_get_num_threads(), 'threads'
  !$omp end parallel
end program hello
    
```

Runtime library functions

Parallel region ends here





Compilation and Execution

- IBM p575 clusters
 - Use the thread-safe compilers (with “_r”)
 - Use '-qsmp=omp' option

```
%xlc_r -qsmp=omp test.c && OMP_NUM_THREADS=4 ./a.out
```

- Dell Linux clusters
 - Use '-openmp' option (Intel compiler)

```
%icc -openmp test.c && OMP_NUM_THREADS=4 ./a.out
```





Exercise 1: Hello World

- Write a “hello world” program with OpenMP where
 - If the thread id is odd, then print a message “Hello world from thread x, I'm odd!”
 - If the thread id is even, then print a message “Hello world from thread x, I am even!”





Solution

C/C++

```
#include <omp.h>
#include <stdio.h>
int main() {
    int id;
#pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        if (id%2==1)
            printf("Hello world from
thread %d, I am odd\n", id);
        else
            printf("Hello world from
thread %d, I am even\n", id);
    }
}
```

Fortran

```
program hello
    implicit none
    integer i,omp_get_thread_num
    !$omp parallel private(i)
    i = omp_get_thread_num()
    if (mod(i,2).eq.1) then
        print *,'Hello world from
thread',i,', I am odd!'
    else
        print *,'Hello world from
thread',i,', I am even!'
    endif
    !$omp end parallel
end program hello
```





Work Sharing: Parallel Loops

- We need to share work among threads to achieve parallelism
- Loops are the most likely targets when parallelizing a serial program
- Syntax
 - Fortran: `!$omp parallel do`
 - C/C++: `#pragma omp parallel for`
- Other working sharing directives available
 - Sections (discussed later)
 - Tasks (new feature in OpenMP 3.0)





Example: Parallel Loops

C/C++

```
#include <omp.h>
int main() {
    int i=0,N=100,a[100];
    #pragma omp parallel for
    for (i=0;i<N;i++){
        a[i]=user_function(i);
    }
}
```

Fortran

```
program paralleldo
    implicit none
    integer i,n,a(100)
    i=0
    n=100
    !$omp parallel do
    do i=1,n
        a(i)=user_function(i)
    enddo
end program paralleldo
```





Load Balancing (1)

- OpenMP provides different methods to divide iterations among threads, indicated by the `schedule` clause
 - Syntax: `schedule(<method>, [chunk size])`
- Methods include
 - `Static`: the default schedule; divide iterations into chunks according to `size`, then distribute chunks to each thread in a round-robin manner;
 - `Dynamic`: each thread grabs a chunk of iterations, then requests another chunk upon the completion of the current one, until all iterations executed
 - `Guided`: similar to `dynamic`; the only difference is that the chunk size starts large and shrinks to `size` eventually





Load Balancing (2)

4 threads, 100 iterations

Schedule	Iterations mapped onto thread			
	0	1	2	3
Static	1-25	26-50	51-75	76-100
Static, 20	1-20, 81-100	21-40	41-60	61-80
Dynamic	1...	2...	3...	4...
Dynamic, 10	1-10...	11-20...	21-30...	31-40...





Load Balancing (3)

Schedule	When to use
Static	Even and predictable workload per iteration; scheduling may be done at compilation time, least work at runtime;
Dynamic	Highly variable and unpredictable workload per iteration; most work at runtime
Guided	Special case of <code>dynamic</code> scheduling; compromise between load balancing and scheduling overhead at runtime





Working Sharing: Sections

- Gives a different block to each thread

C/C++

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        some_calculation();
        #pragma omp section
        more_calculation();
        #pragma omp section
        yet_more_calculation();
    }
}
```

Fortran

```
!$omp parallel
!$omp sections
!$omp section
    call some_calculation
!$omp section
    call more_calculation
!$omp section
    call yet_more_calculation
!$omp end sections
!$omp end parallel
```





Scope of Variables

- `Shared(list)`
 - Specifies the variables that are shared among all the threads
- `Private(list)`
 - Creates a local copy of the specified variables for each thread
 - the value is uninitialized!
- `Default(shared|private|none)`
 - Defines the default scope of variables
 - **C/C++ API does not have `default(private)`**
- Most variables are shared by default
 - A few exceptions: iteration variables; stack variables in subroutines; automatic variables within a statement block





Private Variables

- Not initialized at the beginning of parallel region
- After the parallel region
 - Not defined in OpenMP 2.5
 - 0 in OpenMP 3.0

```
void wrong()
{
    int tmp=0;
    #pragma omp for private(tmp)
    for (int j=0; j<100; ++j)
        tmp += j
    printf("%d\n",tmp)
}
```

tmp not initialized here

OpenMP 2.5: tmp undefined
OpenMP 3.0: tmp is 0





Special Cases of Private

- Firstprivate
 - Initialize each private copy with the corresponding value from the master thread
- Lastprivate
 - Allows the value of a private variable to be passed to the shared variable outside the parallel region

```
void correct()
{
    int tmp=0;
    #pragma omp for firstprivate(tmp) \
    lastprivate(tmp)
    for (int j=0; j<100; ++j)
        tmp += j
    printf("%d\n", tmp)
```

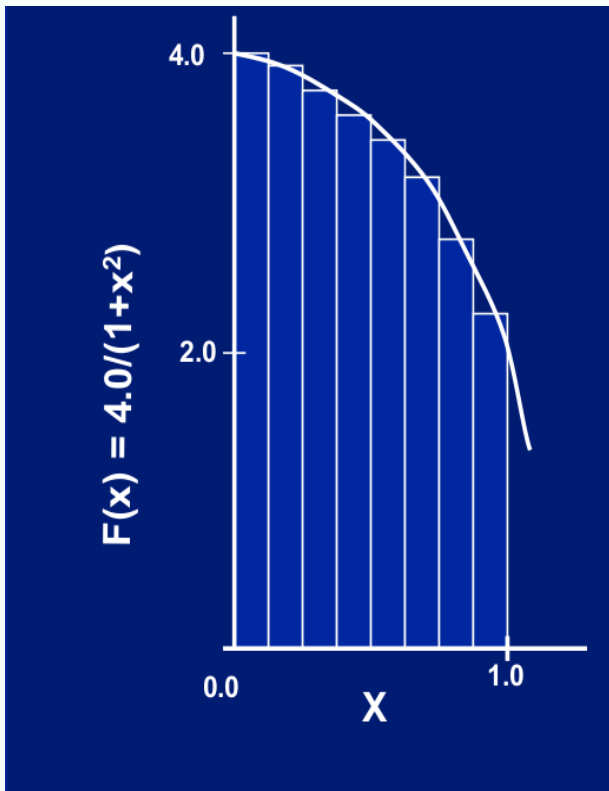
tmp initialized as 0

The value of tmp is the value when j=99





Exercise 2: Calculate pi by Numerical Integration



- We know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

- So numerically we can approximate pi as the sum of the area of a number of rectangles

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$





Serial Version

C/C++

```
double x,deltax,pi,sum=0.0
int i,nstep=<a large number>

deltax=1./((double)nstep

for (i=0; i<nstep; i++)
{
    x=(i+0.5)*deltax
    sum=sum+4./(1.+x*x)
}

pi=deltax*sum
```

Fortran

```
Real*8 :: x,deltax,pi,sum
integer :: i,nstep

nstep=<a large number>
sum=0

deltax=1./float(nstep)

do i=1,nstep
    x=(i+0.5)*deltax
    sum=sum+4./(1.+x*x)
enddo

pi=deltax*sum
```





Pitfalls (1): False Sharing

- Array elements that are in the same cache line can lead to false sharing
 - The system handles cache coherence on a cache line basis, not on a byte or word basis
 - Each update of a single element could invalidate the entire cache line

```
!$omp parallel
myid=omp_get_thread_num()
nthreads=omp_get_num_threads()
do i=myid+1,n,nthreads
    a(i)=some_function(i)
enddo
```





Pitfalls (2): Race Condition

- Multiple threads try to write to the same memory location at the same time
 - Indeterministic results
- Inappropriate scope of variable can cause indeterministic results too
- When having indeterministic results, set the number of threads to 1 to check
 - If problem persists: scope problem
 - If problem is solved: race condition

```
!$omp parallel do
do i=1,n
    if (a(i).gt.max) then
        max=a(i)
    endif
enddo
```





Reduction

- The `reduction` clause allows accumulative operations on the value of variables
- **Syntax:** `reduction(operator:variable list)`
- A private copy of each variable appears in `reduction` is created as if the `private` clause is specified
- Operators
 - Arithmetic
 - Bitwise
 - Logical





Example: Reduction

C/C++

```
#include <omp.h>
int main() {
    int i,N=100,sum,a[100],b[100];
    for (i=0;i<N;i++){
        a[i]=i;
        b[i]=1;
    }
    sum=0;
    #pragma omp parallel for
    reduction(+:sum)
    for (i=0;i<N;i++){
        sum=sum+a[i]*b[i];
    }
}
```

Fortran

```
program reduction
    implicit none
    integer i,n,sum,a(100),b(100)
    n=100
    do i=1,n
        a(i)=i
    enddo
    b=1
    sum=0
    !$omp parallel do reduction(+:sum)
    do i=1,n
        sum=sum+a(i)*b(i)
    enddo
end
```





Exercise 3: Calculate pi by Numerical Integration

- Redo the pi calculation using reduction





Synchronization: Barrier

- “Stop sign” where every thread waits until all threads arrive
- Purpose: protect access to shared data
- Syntax
 - Fortran: `!$omp barrier`
 - C/C++: `#pragma omp barrier`
- A barrier is implied at the end of every parallel region
 - Use the `nowait` clause to turn it off
- Synchronizations are costly so their usage should be minimized





Synchronization: Critical and Atomic

- Critical
 - Only one thread at a time can enter a critical region
- Atomic
 - Only one thread at a time can update a memory location

```
double x;  
#pragma omp parallel for  
for (i=0;i<N;i++)  
{  
    a = some_calculation(i)  
#pragma omp critical  
    some_function(a,x);  
}
```

```
double a;  
#pragma omp parallel  
{ double b;  
    b = some_calculation();  
#pragma omp atomic  
    a += b;  
}
```





Runtime Library Functions

- Modify/query the number of threads
 - `omp_set_num_threads()`, `omp_get_num_threads()`,
`omp_get_thread_num()`, `omp_get_max_threads()`
- Query the number of processes
 - `omp_num_procs()`
- Query whether or not in an active parallel region
 - `omp_in_parallel()`
- Control the behavior of dynamic threads
 - `omp_set_dynamic()`, `omp_get_dynamic()`





Environment Variables

- `OMP_NUM_THREADS`: set default number of threads to use
- `OMP_SCHEDULE`: control how iterations are scheduled for parallel loops





More Exercises

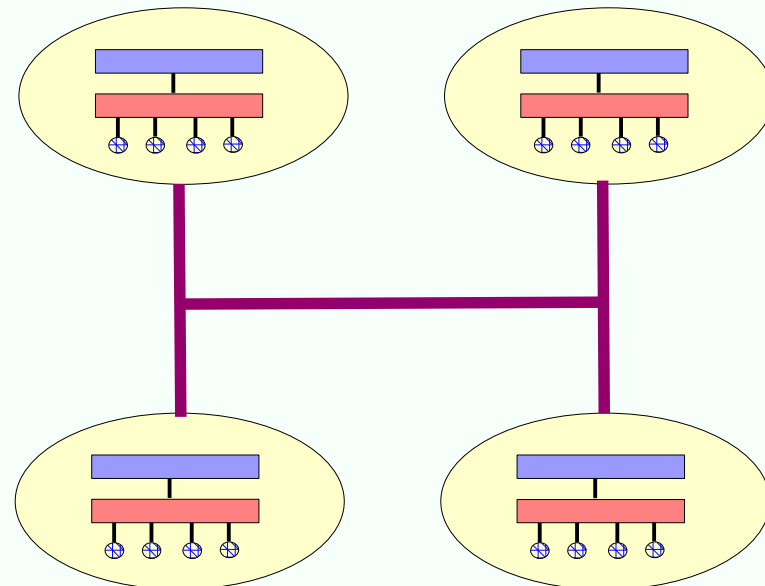
- Cell problem
 - <http://www.cct.lsu.edu/~lyan1/totalview/lsu.php>
 - We will revisit it tomorrow during the “Debugging and Profiling” session
- Matt's molecular dynamic code
 - Details will be provided tomorrow
- Or your own sequential code





MPI+OpenMP Hybrid Programming

- Use MPI and OpenMP in the same program
- Natural programming model for clusters of SMP nodes
 - The system is not flat, why use a flat programming model (pure MPI)?
- MPI between the SMP nodes
- OpenMP within each node





A Sample Hybrid Program

```
call mpi_initialize(ierr) ! MPI initialization
call domain_decomposition ! Divide domain among MPI processes
...
do timestep=1,n
  do subdomain=1,ndomain
    if (this is my subdomain) then
!$omp parallel ! OpenMP threads handle calculate in subdomain
      some calculation
!$omp end parallel
    endif
  enddo
  call mpi_send() ! Exchange data with other MPI processes
  call mpi_recv()
enddo
...
call mpi_finalize(ierr) ! MPI finalization
```





Different Ways of Writing A Hybrid Code

Only the master thread makes MPI calls
(the one that calls MPI_Init)

```
#pragma omp parallel
{
    some calculation
}
/* on mast thread only */
calls to MPI functions
```

Multiple threads make MPI calls, but one
at a time

```
#pragma omp parallel
{ some calculation
#pragma omp critical
    calls to MPI functions
    some other calculation
}
```

Multiple threads make MPI calls at any
time

```
#pragma omp parallel
{ some calculation
  calls to MPI functions
  some other calculation
}
```



Different Ways of Writing A Hybrid Code

Only the master thread makes MPI calls

(Supported by most MPI implementations (but not required by the standard))

```
#pragma omp parallel
{
    some calculation
}
/* on mast thread only */
calls to MPI functions
```

Multiple threads make MPI calls but one at a time

Might not be supported; users are responsible to avoid race condition

```
#pragma omp parallel
{ some calculation
  #pragma omp critical
    calls to MPI functions
  some other calculation
}
```

Multiple threads make MPI calls at any time

Might not be supported; users are responsible to avoid race condition; user need to make sure threads are properly ordered for collective communications

```
#pragma omp parallel
{ some calculation
  calls to MPI functions
  some other calculation
}
```





Hybrid Is Good, But...

- Hybrid programs are often not much faster (if not slower) than pure MPI programs
 - MPI implementation may already use shared memory to communicate within a node
 - Process/thread placement issue (thread affinity is very important)
 - SMP nodes are not flat either: multi-socket, shared vs. separate caches
 - OpenMP loop overhead depends on position of threads
- Therefore, adding OpenMP directives to a MPI program just for the sake of being hybrid may not be a good idea
- Hybrid programs perform best when
 - Limited parallel opportunity for MPI on outer level
 - Severe load balancing issue with pure MPI





Compiling and Running on LONI Clusters

- Compile using MPI compiler with the OpenMP flag
 - Linux: `mpicc -openmp hybrid.c`
 - AIX: `mpcc_r -qsmp=omp hybrid.c`
- Run on Linux clusters
 - Create a new machine file that only contains unique nodes
 - Set the following environment variable
 - MVAICH: `VIADEV_USE_AFFINITY=0`
 - MVAICH2: `MV2_ENABLE_AFFINITY=0`
- Run on AIX clusters
 - Set `MP_SHARED_MEMORY=yes`





References

- https://docs.loni.org/wiki/Using_OpenMP
- <http://en.wikipedia.org/wiki/OpenMP>
- <http://www.nersc.gov/nusers/help/tutorials/openmp/>
- <http://www.llnl.gov/computing/tutorials/openMP/>

