# Introduction to hybrid programming

Carlo Cavazzoni, HPC department, CINECA

# MPI
## Message Passing Interface

CINECA
Consorzio Interuniversitario

# A First Program: Hello World!

Fortran

```
PROGRAM hello

  INCLUDE 'mpif.h'
  INTEGER err

  CALL MPI_INIT(err)
  PRINT *, "hello world!"
  CALL MPI_FINALIZE(err)

END
```

C

```
#include <stdio.h>
#include <mpi.h>

void main (int argc, char * argv[])
{
  int err;

  err = MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  err = MPI_Finalize();
}
```

# A little more than Hello World!

```
PROGRAM hello
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER:: myPE, totPEs, i, ierr

CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK( MPI_COMM_WORLD, myPE, ierr )
CALL MPI_COMM_SIZE( MPI_COMM_WORLD, totPEs, ierr )
PRINT *, "myPE is ", myPE, "of total ", totPEs, " PEs"
CALL MPI_FINALIZE(ierr)
END PROGRAM hello
```

Output (4 Procs)

```
MyPE is 1 of total 4 PEs
MyPE is 0 of total 4 PEs
MyPE is 3 of total 4 PEs
MyPE is 2 of total 4 PEs
```

CINECA Consorzio Interuniversitario

# Send and Receive

```
PROGRAM send_recv

INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

IF( myid .EQ. 0 ) THEN
  A(1) = 3.0
  A(2) = 5.0
  CALL MPI_SEND(A, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  CALL MPI_RECV(A, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
  WRITE(6,*) myid,': a(1)=',a(1),' a(2)=',a(2)
END IF

CALL MPI_FINALIZE(ierr)
END
```

# Send and Receive, the easy way.

```
PROGRAM send_recv
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), B(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
IF( myid .EQ. 0 ) THEN
  a(1) = 2.0
  a(2) = 4.0
  CALL MPI_SENDRECV(a, 2, MPI_REAL, 1, 10, b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  a(1) = 3.0
  a(2) = 5.0
  CALL MPI_SENDRECV(a, 2, MPI_REAL, 0, 11, b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
END IF
WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
CALL MPI_FINALIZE(ierr)
END
```

# Collective Communications

-Communications involving a group of process
-Called by all processes in a communicator

Barrier Synchronization

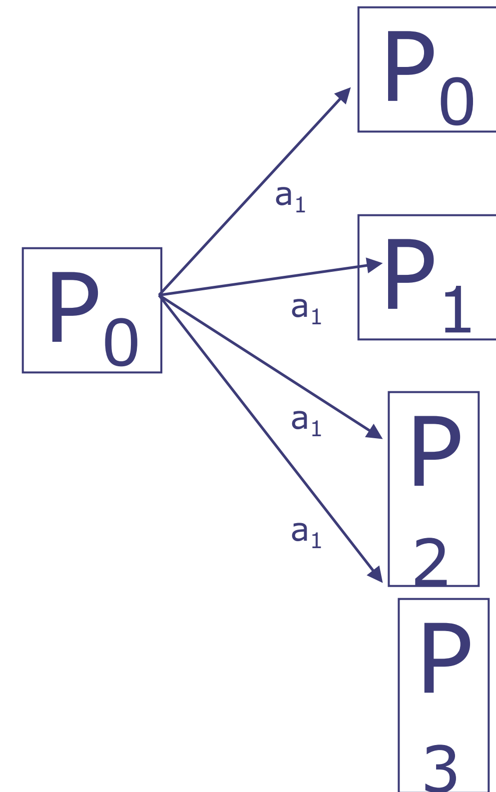
Broadcast

Gather/Scatter

Reduction (sum, max, prod, … )

# Broadcast

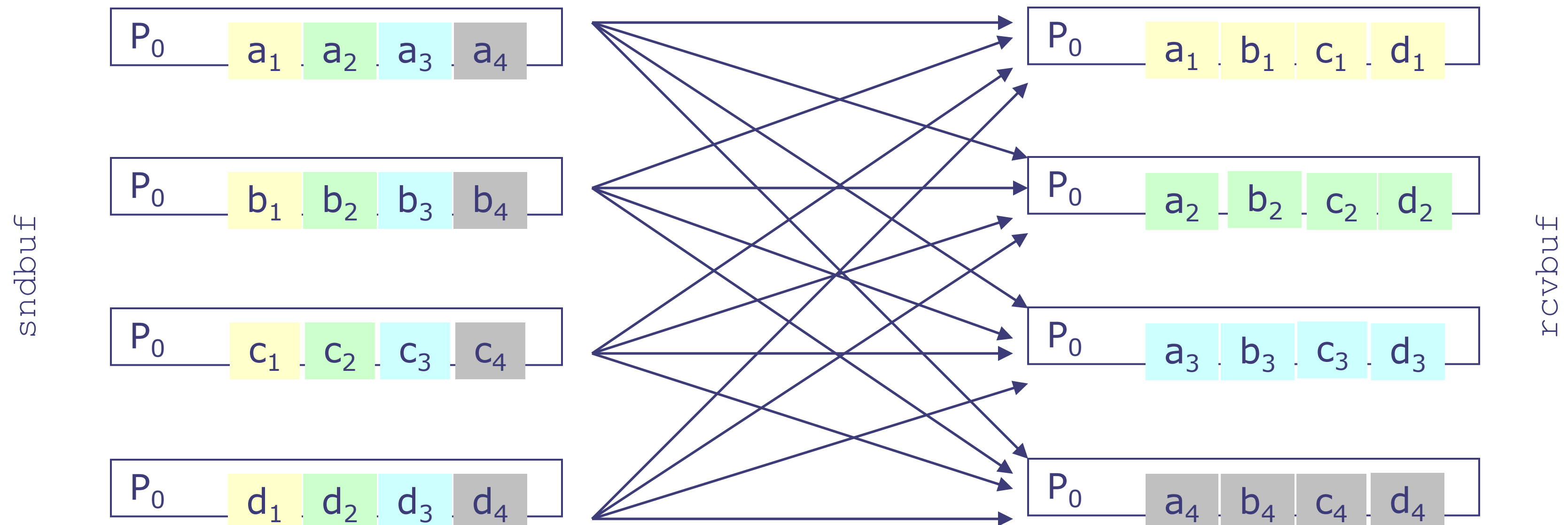
```
PROGRAM broad_cast
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 IF( myid .EQ. 0 ) THEN
   a(1) = 2.0
   a(2) = 4.0
 END IF
 CALL MPI_BCAST(a, 2, MPI_REAL, 0,
                MPI_COMM_WORLD, ierr)
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 CALL MPI_FINALIZE(ierr)
 END
```



# MPI_Alltoall

Fortran:

```
CALL MPI_ALLTOALL(sndbuf, sndcount, sndtype, rcvbuf, rcvcount,
rcvtype, comm, ierr)
```




Very useful to implement data transposition

# Reduce, example

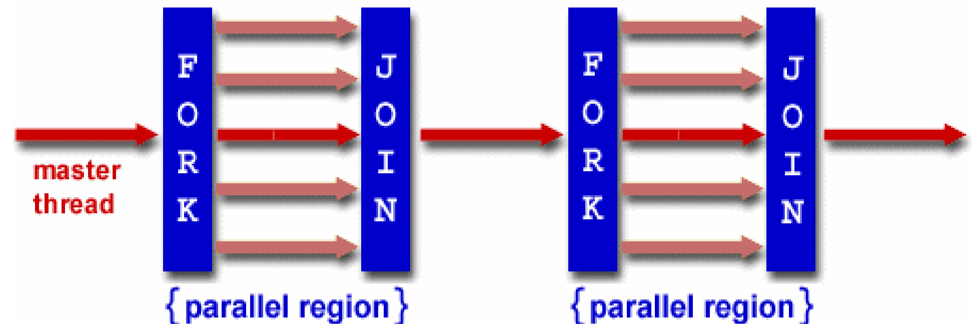
```
 PROGRAM reduce
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2), res(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 a(1) = 2.0*myid
 a(2) = 4.0+myid
 CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root,
&  MPI_COMM_WORLD, ierr)
 IF( myid .EQ. 0 ) THEN
   WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
 END IF
 CALL MPI_FINALIZE(ierr)
 END
```

OpenMP

- The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.
- OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

# OpenMP

- The OpenMP API uses the fork-join model of parallel execution.
- An OpenMP program begins as a single thread of execution, called the initial thread. The initial thread executes sequentially until encounters a parallel construct.
- The initial thread creates a team of threads and becomes the master of the new team. Beyond the end of the parallel construct, only the master thread resume execution.



OpenMP directives for C/C++ are specified with the pragma preprocessing directive.

The syntax of an OpenMP directive is formally specified as follows:

- C/C++:

#pragma omp directive-name [clause[[,]clause]...]

- Fortran:

!$omp directive-name [clause[[,]clause]...]

A first program in Fortran:

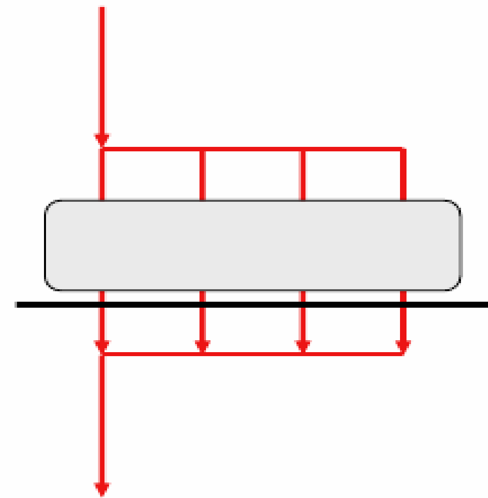
```
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3
!Serial code
!Beginning of parallel region.
!Fork a team of threads.
!Specify variable scoping.
!$OMP PARALLEL
  Print *, "Hello World!!!"
!$OMP END PARALLEL
!Resume serial code

END
```

Export OMP_NUM_THREADS=4
Setenv OMP_NUM_THREADS 4

# Loop construct (DO/for)

Fortran:

```
integer :: i,n=200
real :: a(n),b(n),c(n)
!$OMP PARALLEL
!$OMP DO
do i=1, n
  a(i) = b(i) + c(i)
enddo
!$OMP END DO
!$OMP END PARALLEL
```



# Master construct

The master construct specifies a structured block that is executed by the master thread of the team.

There is no implied barrier either on entry to, or exit from, the master construct.

Fortran:

```
!$OMP PARALLEL
…
!$OMP MASTER
read *, a
!$OMP END MASTER
…
!$OMP END PARALLEL
```

# Critical construct

The critical construct restricts execution of the associated structured block to a single thread at a time. An optional name may be used to identify the critical construct.

Fortran:

```
!$OMP PARALLEL
…
!$OMP CRITICAL [NAME]
X=FUNC_A(X)
!$OMP END CRITICAL
…
!$OMP END PARALLEL
```

# Barrier construct

The barrier construct specifies an explicit barrier at the point at which the construct appears.

Fortran:

```
!$OMP PARALLEL
…
X=FUNC_A(X)
!$OMP BARRIER
…
!$OMP END PARALLEL
```

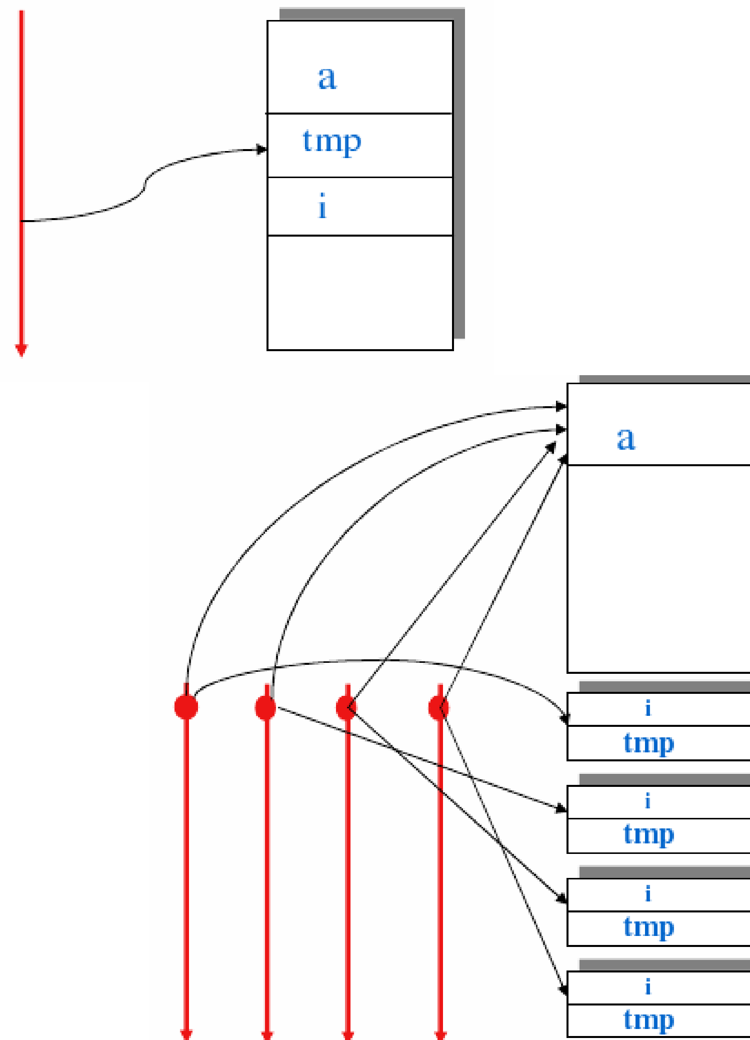
# Atomic construct

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

Fortran:

```
!$OMP PARALLEL
…
!$OMP ATOMIC
X=X+1
…
!$OMP END PARALLEL
```

# OpenMP Memory Model




Fortran:

```
integer :: i=5,n=200
real :: tmp=7
!$OMP PARALLEL
!$OMP DO PRIVATE(tmp)
do i=1, n
  tmp = func(b(i))
  a(i) = b(i) + tmp
enddo
!$OMP END DO
!$OMP END PARALLEL
```

# Data-Sharing Attribute Clauses

**Reduction**: The reduction clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

```
!$omp do reduction (+:x)
do i = 1,n
  x = x + a(i)
enddo
!$omp end do
```

Support for most arithmetic and logical operators

+, *, -, .MIN., .MAX., . AND., .OR., ...

# Environment Variables

- **OMP_NUM_THREADS**: sets the number of threads to use for parallel regions;
- **OMP_SCHEDULE:** controls the schedule type and chunk size of all loop directives that have the schedule type runtime.
- **OMP_STACKSIZE:** specifies the size of the stack for threads created by the OpenMP implementation.

***csh:***

```
% setenv OMP_NUM_THREADS 8
% setenv OMP_SCHEDULE "guided,4"
```

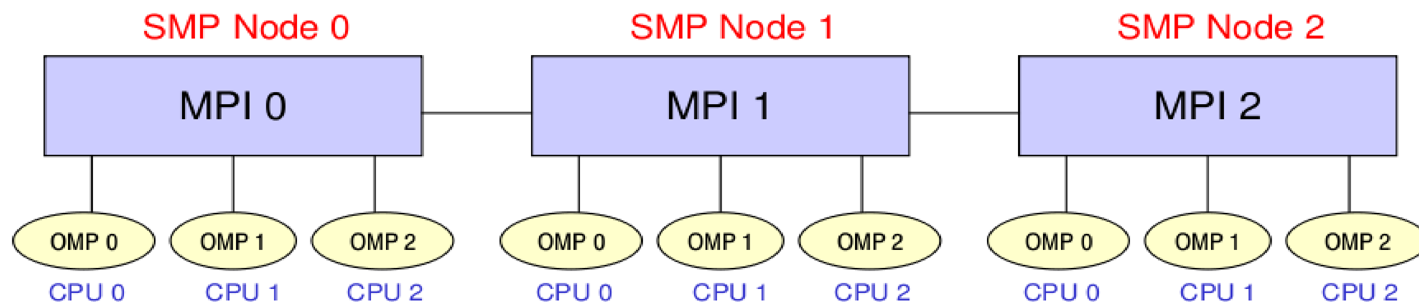
***sh:***

```
$ export OMP_NUM_THREADS=8
$ export OMP_SCHEDULE=”guided,4”
```

# Hybrid programming MPI+OpenMP

# The hybrid model

- Multi-node SMP (Symmetric Multiprocessor) connected by and interconnection network.
- Each node is mapped (at least) one process MPI and OpenMP threads more.



# MPI *vs.* OpenMP

- **Pure MPI Pro:**
  - High scalability
  - High portability
  - No false sharing
  - Scalability out-of-node
- **Pure MPI Con:**
  - Hard to develop and debug.
  - Explicit communications
  - Coarse granularity
  - Hard to ensure load balancing

**Pure OpenMP Pro:**

Easy do deploy (often)

Low latency

Implicit communications

Coarse and fine granularity

Dynamic Load balancing

**Pure OpenMP Con:**

Only on shared memory machines

Intranode scalability

Possible long waits for unlocking data

No order specific thread

# Why hybrid?

- MPI+OpenMP hybrid paradigm is the trend for clusters with SMP architecture.

Elegant in concept: use OpenMP within the node and MPI between nodes, in order to have a good use of shared resources.

- Avoid additional communication within the MPI node.
- OpenMP introduces fine-granularity.
- Two-level parallelism introduces other problems
- Some problems can be reduced by lowering MPI procs number
- If the problem is suitable, the hybrid approach can have better performance than pure MPI or OpenMP codes.

# Why mixing MPI and OpenMP code can be slower?

- OpenMP has lower scalability because of locking resources while MPI has not potential scalability limits.
- All threads are idle except ones during an MPI communication
  - Need overlap computation and communication to improve performance
  - Critical section for shared variables
- Overhead of thread creation
- Cache coherency and false sharing.
- Pure OpenMP code is generally slower than pure MPI code
- Few optimizations by OpenMP compilers compared to MPI

# Pseudo hybrid code

```
call MPI_INIT (ierr)
call MPI_COMM_RANK (…)
call MPI_COMM_SIZE (…)
 …  some computation and MPI communication
 call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL
!$OMP DO
   do i=1,n
       … computation
   enddo
 !$OMP END  DO
 !$OMP END  PARALLEL
 …  some computation and MPI communication
call MPI_FINALIZE (ierr)
```

# MPI_INIT_Thread support ( MPI-2)

MPI_INIT_THREAD (required, provided, ierr)

- IN: required, desider level of thread support (integer).
- OUT: provided, provided level (integer).
- provided may be less than required.

Four levels are supported:

- MPI_THREAD_SINGLE: Only one thread will runs. Equals to MPI_INIT.
- MPI_THREAD_FUNNELED: processes may be multithreaded, but only the main thread can make MPI calls (MPI calls are delegated to main thread)
- MPI_THREAD_SERIALIZED: processes could be multithreaded. More than one thread can make MPI calls, but only one at a time.
- MPI_THREAD_MULTIPLE: multiple threads can make MPI calls, with no restrictions.

# MPI_THREAD_SINGLE

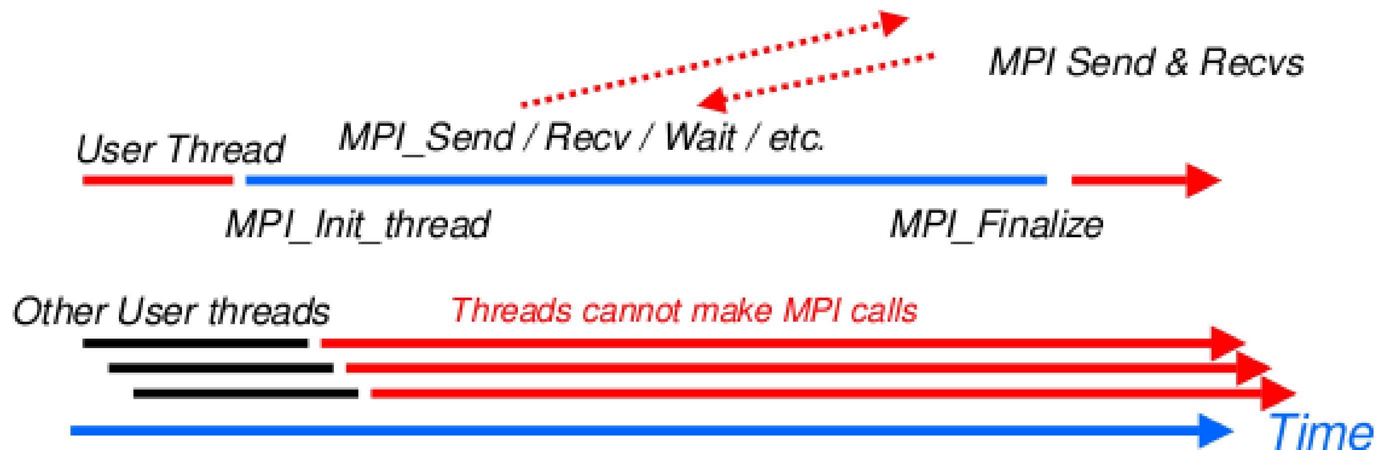
Hot to implement:

```
!$OMP PARALLEL DO
  do i=1,10000
     a(i)=b(i)+f*d(i)
  enddo
!$OMP END PARALLEL DO
  call MPI_Xxx(...)
!$OMP PARALLEL DO
  do i=1,10000
     x(i)=a(i)+f*b(i)
  enddo
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for
     for (i=0; i<10000; i++)
     { a[i]=b[i]+f*d[i];
     }
/* end omp parallel for */
     MPI_Xxx(...);
#pragma omp parallel for
     for (i=0; i<10000; i++)
     { x[i]=a[i]+f*b[i];
     }
/* end omp parallel for */
```

# MPI_THREAD_FUNNELED

Only the main thread can do MPI communications. Obviously, there is a main thread for each node



# MPI_THREAD_FUNNELED

MPI calls outside the parallel region.

Inside the parallel region with “omp master”.

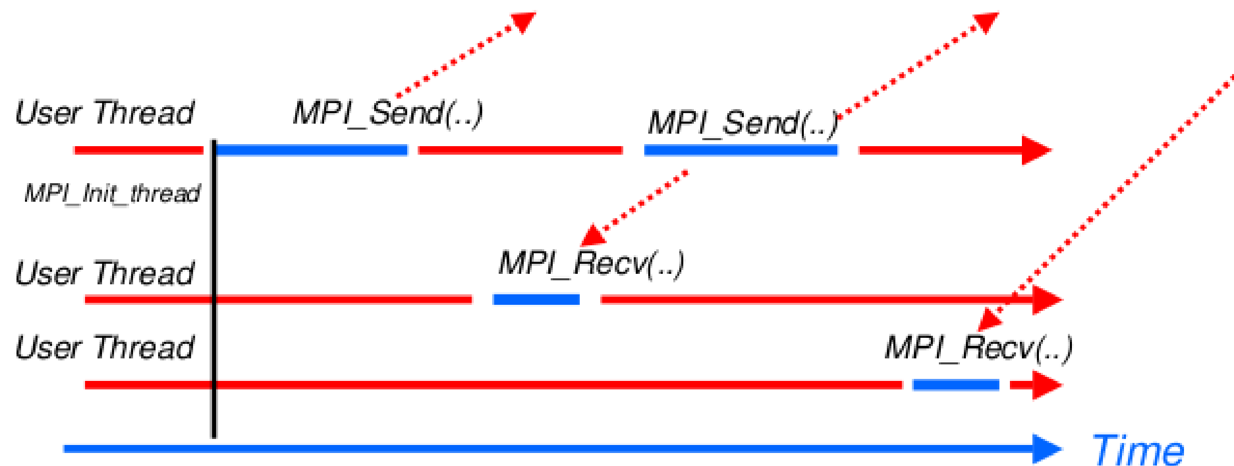
```
!$OMP BARRIER
!$OMP MASTER
      call MPI_Xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

```
#pragma omp barrier
#pragma omp master
     MPI_Xxx(...);
#pragma omp barrier
```

There are no synchronizations with “omp master”, thus needs a barrier before and after, to ensure that data and buffers are availabe before and/or after MPI calls

# MPI_THREAD_SERIALIZED

MPI calls are made concurrently by two (or more) different threads
(all MPI calls are serialized)



# MPI_THREAD_SERIALIZED

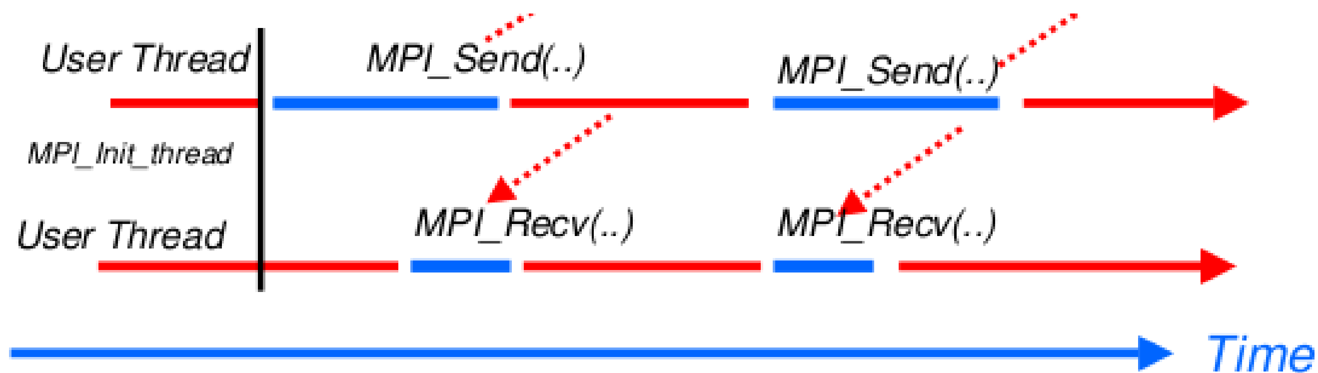
- Outside the parallel region
- Inside the parallel region with ”omp master”
- Inside the parallel region with “omp single”

```
!$OMP BARRIER
!$OMP SINGLE
      call MPI_Xxx(...)
!$OMP END SINGLE
```

```
#pragma omp barrier
#pragma omp single
    MPI_Xxx(...);
```

# MPI_THREAD_MULTIPLE

Any thread can make communications at all times. Less restrictive and very flexible, but the application becomes very hard to manage



# A little example

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
 int rank,omp_rank,mpisupport;
 MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED, &mpisupport);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 omp_set_num_threads(atoi(argv[1]));
 #pragma omp parallel private(omp_rank)
  {
    omp_rank=omp_get_thread_num();
    printf("%d %d \n",rank,omp_rank);
  }
  MPI_Finalize();
}
```

Output-->

```
0 0
0 2
0 1
0 3
1 0
1 2
1 1
1 3
```

# Overlap communications and computation

- Need at least MPI_THREAD_FUNNELED.
- While the master or the single thread is making MPI calls, other threads are doing computations.
- It's difficult to separate code that can run before or after the exchanged data are available

```
!$OMP PARALLEL
    if (thread_id==0) then
        call MPI_xxx(…)
    else
        do some computation
    endif
!$OMP END PARALLEL
```