

OMP Functions

- See the text in the online NCSA course

OpenMP Correctness Rules

- A correct OpenMP program...
 - should not depend on the number of threads
 - should not depend on a particular schedule
 - should not have BARRIER in serialization or work sharing construct (critical, omp do/for, section, single)
 - should not have work sharing constructs inside serialization or other work sharing constructs
 - all threads should reach same work sharing constructs

OpenMP Efficiency Rules

- Optimization for
- scalability and performance:
 - maximize independent data
 - minimize synchronization

OpenMP “Danger Zones”

- 3 major SMP programming errors:
 - Race Conditions
 - the outcome of the program depends on the detailed timing of the threads in the team
 - Deadlock
 - threads lock up waiting on a locked resource that will never come free
 - Livelock
 - multiple threads working individual tasks which the ensemble can not finish
 - Death traps:
 - thread safe libraries?
 - Simultaneous access to shared data
 - I/O inside parallel region
 - shared memory not coherent (FLUSH)
 - implied barriers removed (NOWAIT)

Deadlock/1

The following code shows a race condition with deadlock:

```
        call omp_init_lock(lcka)
        call omp_init_lock(lckb)
C$omp parallel sections
c$omp section
        call omp_set_lock(lcka)
        call omp_set_lock(lckb)
            call use_A_and_B(res)
        call omp_unset_lock(lckb)
        call omp_unset_lock(lcka)
c$omp section
        call omp_set_lock(lckb)
        call omp_set_lock(lcka)
            call use_B_and_A(res)
        call omp_unset_lock(lcka)
        call omp_unset_lock(lckb)
c$omp end parallel sections
```

- if A is locked by one thread and B by another - there is a deadlock
- if the same thread gets both locks, you get a race condition:
 - different behaviour depending on detailed timing of the threads
- **Avoid nesting different locks**

Deadlock/2

Often deadlock condition arises from error processing:

```
    call omp_init_lock(lcka)
C$omp parallel sections
c$omp section
    call omp_set_lock(lcka)
        ival = dowork()
        if(ival.EQ.TOL) then
            call omp_unset_lock(lcka)
        else
            call error(ival)
        endif
c$omp section
    call omp_set_lock(lcka)
        call use_B_and_A(res)
    call omp_unset_lock(lcka)
c$omp end parallel sections
```

- if lcka is set in the first section and the if statement branches around the unset lock, threads running the other sections will deadlock waiting forever for the lock to be released
- **make sure locks are always released**

Livelock

The following code shows a race condition and a livelock:

```
C$omp parallel private(id)
    id = omp_get_thread_num()
    n = omp_get_num_threads()
10    continue
    phases(id) = update(u, id)
c$omp single
    res = match(phases, n)
c$omp end single
    if(res.lt.TOL) goto 20
    goto 10
20    continue
c$omp end parallel
```

```
C$omp barrier
c$omp single
    res = match(phases, n)
    icount = icount + 1
c$omp end single
    if(res.lt.TOL.or
        icount.gt.MAX) goto 20
    goto 10
20    continue
c$omp end parallel
```

- if res is never smaller than TOL, program spins endlessly in Livelock
- the race may be fixed with a barrier before the single, such that the match function will work better
- also, decide on the maximum number of iterations, rather than an infinite loop

Types of MPI Routines

- **Point-to-point communication**
- **Collective communication**
- Process groups
- Process topologies
- Environment management and inquiry
- *main focus will be point-to-point and collective communications.*

Point-to-Point Communications and Messages

- elementary communication operation in MPI is point-to-point
 - direct communication between two processors,
 - one **sends** data and the other **receives** same data.
- P-to-p is two-sided
 - need an explicit send and an explicit receive
 - Data not transferred without participation of both processors.
- In a generic send or receive a **message** with some block of data is transferred between processors.
- A message consists of
 - an **envelope** that indicates the source and destination processors,
 - and a **body** that contains the actual data to be sent.

P-to-P Message Body

- Contains three pieces of information
- **Buffer** — starting location in memory of data
 - where outgoing data is to be found (for a send)
 - incoming data is to be stored (for a receive).
 - C -> buffer is the actual address of the array element
 - Fortran -> name of array element
- **Datatype** — the type of data to be sent.
 - elementary types such as float (REAL), int (INTEGER).
 - Or a user-defined datatype built from the basic types.
 - roughly analogous to C structures, or Fortran modules
 - not necessarily in contiguous memory locations.
- **Count** — number of datatype items to be sent/recvd

Communication modes

- **asynchronous send** is defined to be complete when receipt of the message at its destination has been acknowledged.
- **buffered** send is complete when the outgoing data has been copied to a local buffer, awaiting transfer
- completion of a send makes it safe to overwrite original data
- Four communication modes available for sends:
 - Standard
 - Synchronous
 - Buffered
 - Ready
- For receives, there is only a single communication mode.
- Receive is complete when incoming data has actually arrived and is available

Blocking/Non-Blocking

- Blocking send or receive does not return from the subroutine call until the operation has actually completed.
 - Safe and ensures that the process completes before continuing
 - Blocking send ensures that variables sent can safely be overwritten on the sending processor.
 - Blocking receive ensures data has actually arrived and is ready for use.
- Nonblocking send or receive returns immediately with no information about whether operation has completed
 - Advantage that the processor is free to do something else while communication proceeds in the background
 - Remember communication is slow
- Polling functions to check completion
 - E.g., nonblocking synchronous send returns immediately,
 - Send is only complete when message is received which sets a flag to a value
 - sending processor can then do other useful work,
 - Will test variable to see if the send is complete.
 - However, until complete the variables sent may not be safely overwritten.

MPI Communications

- Buffering
 - no guarantee that there are buffers
 - possible that send will block until receive is called
- Delivery Order
 - two sends from same process to same dest. will arrive in order
 - no guarantee of fairness between processes on recv.

MPI Communicators

All processes within a communicator can be named

- numbered from 0...n-1
- Allows libraries to be constructed
 - application creates communicators
 - library uses it
 - prevents problems with posting wildcard receives
 - adds a communicator scope to each receive
- All programs start with
MPI_COMM_WORLD

Non-Blocking Functions

- Two Parts
 - post the operation
 - wait for results
- Also includes a poll option
 - checks if the operation has finished
- Semantics
 - must not alter buffer while operation is pending