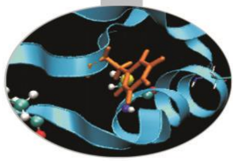
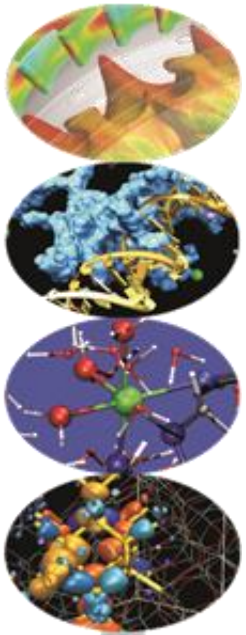
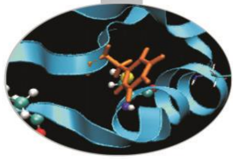
Advanced MPI

Andrew Emerson (a.emerson@cineca.it)

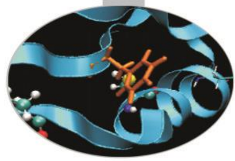





Agenda

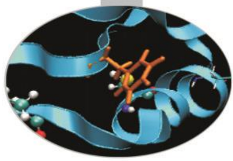
1. One sided Communications (MPI-2)
2. Dynamic processes (MPI-2)
3. Profiling MPI and tracing
4. MPI-I/O
5. MPI-3

One sided communications



- In two-sided (point-to-point) communications there can be a delay if the sender has to wait to send the data because the receiver is not ready.
- The MPI-2 standard added Remote Memory Access (RMA), also called one-sided communication, to decouple data transfer from system synchronisation.
- In RMA only one process carries out the data transfer. The MPI_Get and MPI_Put calls are non-blocking and don't require intervention of the remote process.
- MPI-3 further extended RMA to improve functionality and performance.
- In this course we only describe the simple MPI RMA functionality with MPI Get/Put and Fence synchronisation.

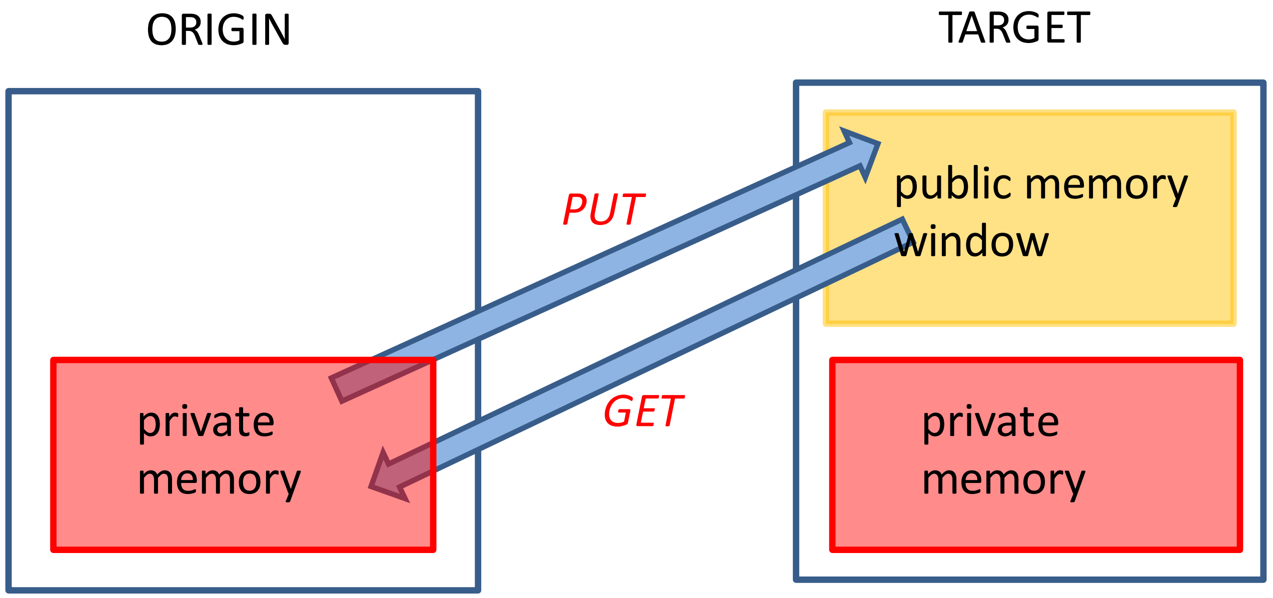
One sided communications



- Advantages of RMA:
  - With only one process taking part performance should be greater (no implicit synchronization, all data movement routines are non-blocking)
  - Some programs are more easily written with RMA

Using one sided communications

1. Define an area of memory to be used for the RMA (“window”).
2. Specify the data to be moved and where to move them.
3. Specify a way to know when the data are available.



Using one sided communications – MPI_Win_Create

```
int MPI_Win_create( void *base, MPI_Aint size, int
disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
MPI_Win_Create(base, integer(KIND=MPI_ADDRESS_KIND) size,
Integer disp_unit, integer info, integer comm, integer
win, integer ierr)
```

base – initial address of the window (IN)

size – size of the window in bytes (IN)

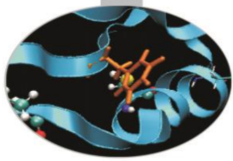
info – info argument (IN)

comm – communicator

win – window object handle (OUT)

ierr – error code for Fortran

Using one sided communications – MPI_Get/MPI_Put



```
int MPI_Get( void *origin_addr, int origin_count,
MPI_Datatype origin_datatype, int target_rank, MPI_Aint
target_disp, int target_count, MPI_Datatype
target_datatype, MPI_Win *win)
```

origin_addr – address of the buffer in which to receive data

origin_count –no. of entries in origin buffer

origin_datatype – datatype of each entry in origin buffer

target_rank – rank of target

target_disp- displacement from window start to beginning of target data

target_count- number of entries to transfer

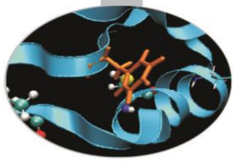
target_datatype – datatype of entries

win – window object handle

ierr – error code for Fortran

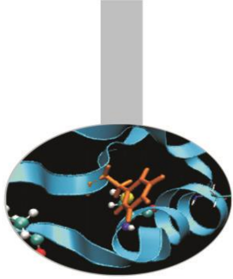
Similarly for MPI_Put

Using one sided communications –synchronisation



- The MPI_Get and MPI_Put calls are non-blocking.
- Need to synchronize the data transfer so that one process knows when it is safe to read the data of another.
- MPI provides various synchronization models, but we will consider only MPI_Win_Fence.
- This is used to *start* and *end* the PUT/GET operations. All operations complete at the second fence synchronization.

Using one sided communications - template

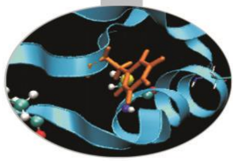


```
...
MPI_Win_Create(shared_buffer,….,win);
MPI_Win_Fence(0,win); // start RMA Get/Put

MPI_Get() or MPI_Put();

MPI_Win_Fence(0,win); // end RMA
// Use transferred data
MPI_Win_Free(&win); //
..
MPI_Finalize();
```

Using one sided communications -Example



```
MPI_Win win;
MPI_Win_create(sharedbuffer, NUM_ELEMENT, sizeof(int), MPI_INFO_NULL,
    MPI_COMM_WORLD, &win);
.....
MPI_Win_fence(0, win);

if (id != 0)
    MPI_Get(&localbuffer[0], NUM_ELEMENT, MPI_INT, id-1, 0, NUM_ELEMENT, MPI_INT,
    win);
else
    MPI_Get(&localbuffer[0], NUM_ELEMENT, MPI_INT, num_procs-1, 0, NUM_ELEMENT,
    MPI_INT, win);

MPI_Win_fence(0, win);
if (id < num_procs-1)
   MPI_Put(&localbuffer[0], NUM_ELEMENT, MPI_INT, id+1, 0, NUM_ELEMENT, MPI_INT,
    win);
 else
   MPI_Put(&localbuffer[0], NUM_ELEMENT, MPI_INT, 0, 0, NUM_ELEMENT, MPI_INT,
    win);

MPI_Win_fence(0, win);
MPI_Win_free(&win);
MPI_Finalize();
```

target rank ↓ (id-1)

← create shared buffer (window)

← synchronize

← get data from target

← synchronize

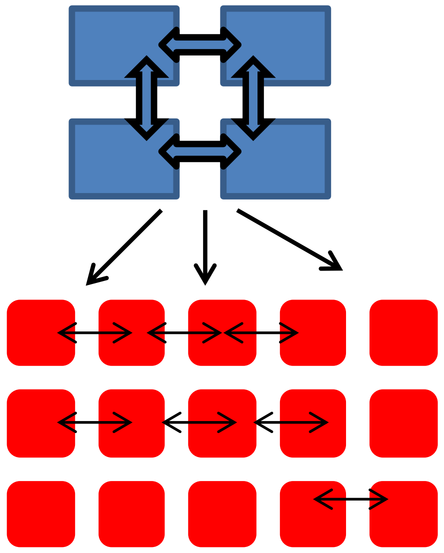
← put data into target

← synchronize

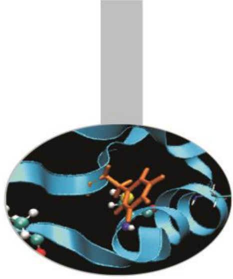
← free window object

Dynamic processes in MPI

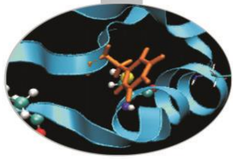
- Normally MPI tasks are fixed (e.g. by mpirun) at the start of execution.
- But can be useful to add or create tasks “on the fly”:
  - Master – slave type codes, or on heterogenous architectures (normal nodes + accelerators).
  - client-server or peer-to-peer
- Handling faults failures



MPI_COMM_SPAWN



- MPI-2 provides "spawn functionality"
  - MPI_COMM_SPAWN
    - starts a new set of processes with the same command lines (SPMD model)
  - MPI_COMM_SPAWN_MULTIPLE
    - starts a new set of processes with potentially different command lines (i.e. different executables and arguments = MPMD)



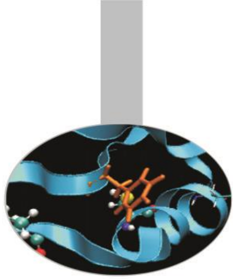
Spawn semantics

- Group of parents collectively call spawn
  - Launches a new set of child processes
  - Child processes become an MPI job
  - An intercommunicator is created between parents and children.
- Parents and children can then use MPI functions to communicate.

MPI_Comm_Spawn example

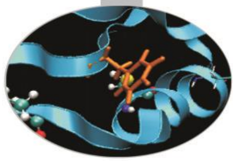
```
#define NUM_SPAWNS 2
int main(int argc, char* argv[])
{
   int np=NUM_SPAWNS;
   MPI_Comm parentcomm, intercomm;
   int errcodes[NUM_SPAWNS];
   MPI_Init( &argc, &argv );
   MPI_Comm_get_parent( &parentcomm );
    if (parentcomm == MPI_COMM_NULL)
    {
    // Create 2 more processes- example must be called spawn_example.exe for this to work
         MPI_Comm_spawn( "./spawnexample", MPI_ARGV_NULL, np, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
    &intercomm, errcodes);
        printf("I'm the parent.\n");
    }
    else
    {
         printf("I'm the spawned.\n");
    }
  MPI_Finalize();
    return 0;
}
```

MPI_COMM_SPAWN



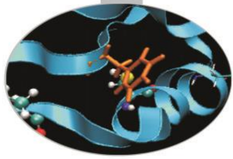
- Not all MPI implementations support MPI spawning (e.g. IBM BG/Q).
- The MPI implementation may require particular runtime options.
- Remember that if working in a batch environment you should allocate resources to cover the spawned processes as well.
  - MPI_UNIVERSE_SIZE is often used to set the total number of processes available (i.e. including spawned processes)
- Not commonly used in HPC environments. May be used in heterogenous (i.e. with accelerators), although OpenMP task creation is more likely.

Debugging and profiling MPI with PMPI



- MPI implementations also provide a profiling interface called PMPI.
- In PMPI each standard MPI function (MPI_) has an equivalent function with prefix PMPI_ (e.g. PMPI_Send, PMI_RECV, etc).
- With PMPI it is possible to customize normal MPI commands to provide extra information useful for profiling or debugging.
- Not necessary to modify source code since the customized MPI commands can be linked as a separate library during debugging. For production the extra library is not linked and the standard MPI behaviour is used.

PMPI Examples



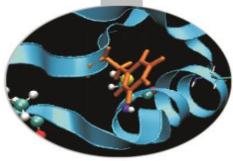
Profiling

```
// profiling example
static int send_count=0;
int MPI_Send(void*start,int count, MPI_Datatype datatype,
   int dest, int tag, MPI_Comm comm)
{
send_count++;
return PMPI_Send(start, count, datatype, dest, tag, comm);
}
```

Debugging

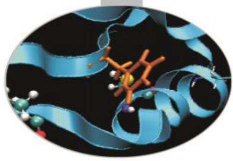
```
! Unsafe uses of MPI_Send
! MPI_Send can be implemented as MPI_Ssend (synchronous send)
subroutine MPI_Send( start, count, datatype, dest,
 tag, comm, ierr )
 integer start(*), count, datatype, dest, tag, comm
 call PMPI_Ssend( start, count, datatype,
 dest, tag, comm, ierr )
end
```

MPI-3



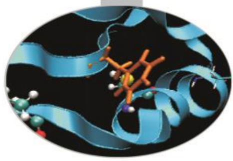
- MPI 3.0 was approved in 2012. MPI 3.1 was approved in 2015.
- Features include
  – Non-blocking collectives
  – Neighbourhood collectives
  – New one sided communications
  – Fortran 2008 bindings
  – plus enhancements for many other features of MPI-2 (e.g. Remote Memory Access).

New collective calls in MPI-3



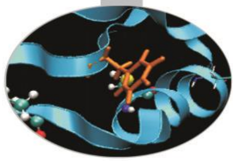
- Collective calls (MPI_Bcast, MPI_Reduce, etc) are very often performance bottlenecks in MPI codes. For Exascale, with potentially millions of process, their impact could be serious.
- MPI-3 has introduced several enhancements to minimise performance loss due to collectives. These include:
  1. Non-blocking collectives
  2. Neighbourhood collectives.

Non-blocking collectives



- Work in the same way to the usual blocking collectives, except that they return almost immediately after being called, i.e. a task does not wait for other tasks to make the call.
- Naming convention just like non-blocking point-to-point calls: MPI_Iallreduce, MPI_Ibarrier, MPI_Ibcast ..
- Used with MPI_Test or MPI_Wait to increase overlap of calculation and computation.

Neighbourhood collectives

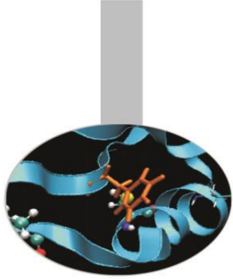


- A special type of collective call for *sparse* communication patterns, i.e where communications occur between a few processes in a communicator.
- In a neighbourhood call each process makes the call but communication *only occurs between nearest neighbours*.

- Example:

```
MPI_Neighbor_allgather(void* sendbuf, int sendcount,
   MPI_Datatype sendtype, void* recvbuf, int recvcount,
   MPI_Datatype recvtype, MPI_Comm comm)
```

This sends the same data element to all neighbor processes and receives a distinct data element from each of the neighbors.



MPI -4 ?

- Under discussion but *resiliency and fault tolerance* likely to be important.
- Current MPI implementations kill all other processes if one process fails.
- Future implementations may allow the program to continue in case of failure of one or more processes.
- Other subjects under discussion include more support for MPI+X (where X is OpenMP, CUDA, OpenACC, OpenCL, etc) and persistent collectives.