MPI Environment
(Cray XT Systems)
MPI Message Protocols

• Message consists of envelope and data
  ▪ Envelope contains tag, communicator, length, source information, plus implementation private data
    – vshort
      ▪ Message data is sent with the envelope
      ▪ Default is 1024 bytes, max is 16,384 (user tunable)
    – short (Eager)
      ▪ Message is sent, based on the expectation that the destination can store; if no matching receive exists, the receiver must buffer or drop
      ▪ Default is 128,000 bytes (user tunable)
    – Long (Rendezvous)
      ▪ Only the envelope is sent (and buffered) immediately
      ▪ Message is not sent until the destination posts a receive
      ▪ Any message longer than short
All short Cray XT messages are eager
- `MPICH_MAX_SHORT_MSG_SIZE` defines the maximum size of a short message (the default size is 128,000 bytes)

For long messages, a small 8-byte message is sent to the receiver, which contains sufficient information for the receiver to pull the message data when a matching receive is posted

However, if the `MPICH_PTL_EAGER_LONG` environment variable is set, the sender sends long messages via the eager (short) protocol
- This is good if application logic ensures that matching receives are pre-posted
Where Do Unexpected Messages Go?

- There are three buffers for unexpected eager messages (20M each by default). Portals delivers unexpected messages (< 128KB) to these buffers.
- Both long and short unexpected messages generate entries in the unexpected event queue (EQ).
- When the process posts a receive, the MPI library checks against unexpected messages and, if it finds a short match, copies data from buffer. If it matches an unexpected long message, it pulls data from the sender.
- Therefore, it is important to prepost receives.
Match Entries created by application pre-posting of
Receives

Match Entries posted by MPI to handle unexpected short and long messages

Incoming message

Portals matches incoming messages with pre-posted messages and delivers message data directly into user buffer.

Pre-posted ME msgX

Pre-posted ME msgY

App buffer for msgY

App buffer for msgX

Eager short message ME

Eager short message ME

Eager short message ME

Long message ME

Expected long message buffers

Portals Event Queue events only

Unexpected long message buffers

An unexpected message generates two entries on the unexpected event Queue

Other EQ

Unexpected EQ

10/18/2010

Cray Private
Cray XT MPI Tunables

**MPICH_UNEX_BUFFER_SIZE**
- Overrides the size of the buffers that are allocated to the MPI unexpected receive queue; default is 60 MB
- If you increase MPICH_MAX_SHORT_MSG_SIZE, increase this one as well; it is the total size of the buffers that hold unexpected short messages

**MPICH_PTL_UNEX_EVENTS**
- The number of event queue entries for unexpected MPI point-to-point messages. Defaults to 20480

**MPICH_PTL_OTHER_EVENTS**
- The number of entries in the event queue that is used to receive all other (not unexpected point-to-point) MPI-related Portals events
MPICH_ALLTOALLVW_FCSIZE

- The algorithm for flow-controlled versions of the MPICH_ALLTOALLV and MPICH_ALLTOALLW is enabled when the size of the communicator is greater than this variable; default is 120

MPICH_ALLTOALLVW_SENDWIN,
MPICH_ALLTOALLVW_RECVWIN

- When flow control is enabled, send and receive windows are established that can allow maximums of 80 \texttt{Isend} operations and 100 \texttt{Irecv} operations; use these variables to change these numbers

- Also applies to medium-size (256<n<32768 bytes) MPI_ALLTOALL operations
Cray XT MPI Tunables

**MPI_COLL_OPT_ON**
- Enables collective optimizations that use non default architecture-specific algorithms for some MPI collective operations

**MPICH_FAST_MEMCPY**
- Enables an optimized `memcpy` routine in MPI

**MPICH_MAX_VSHORT_MSG_SIZE**
- Specifies in bytes the maximum size of a message to be considered for the `vshort` path; default is 1024

**MPICH_VSHORT_BUFFERS**
- Specifies the number of 16,384 byte buffers to be preallocated for the sending side buffering of messages for the `vshort` protocol; default is 32
MPI Rank Reordering

- The default ordering for multi-core nodes is SMP
- `MPICH_RANK_REORDER_METHOD` is an environment variable which allows users to select an alternative ordering.
- To display the MPI rank placement and launching information, set `PMI_DEBUG` to 1.
MPI Rank Reordering

- **MPICH_RANK_REORDER_METHOD** accepts the following values:
  
  1. **Round-robin**
  
  2. Specifies SMP-style placement. For a multi-core node, this places sequential MPI ranks on the same node. For example, for an 8-process MPI job on dual-core nodes, the placement would be:

     ```
     NODE  0  1  2  3  
     RANK  0&1 2&3 4&5 6&7
     ```

  3. Specifies folded-rank placement. Instead of rank placement starting over on the first node when half of the MPI processes have been placed, this option places the N/2 process on the last node, going back to the initial node. For example, for an 8-process job on dual-core nodes, the placement would be:

     ```
     NODE  0  1  2  3  
     RANK  0&7 1&6 2&5 3&4
     ```

  4. Specifies a custom rank placement defined in the file named **MPICH_RANK_ORDER**.

10/18/2010 Cray Private
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

int main(int ac, char**av) {
    int i, me ,np, nameSize;
    char myProcName[MPI_MAX_PROCESSOR_NAME];
    MPI_Init( &ac, &av );
    MPI_Comm_rank( MPI_COMM_WORLD, &me );
    MPI_Comm_size( MPI_COMM_WORLD, &np );
    MPI_Get_processor_name(myProcName, &nameSize);
    for ( i=0; i<np; ++i ) {
        if ( i==me ) {
            printf("rank = %d processor = %s\n",me,myProcName);
            fflush(stdout);
        }
        MPI_Barrier( MPI_COMM_WORLD );
    }
    MPI_Finalize();
    exit(0);
}
% export MPICH_RANK_REORDER_METHOD=1
% export PMI_DEBUG=1
aprun -n 8 ./MPI_where
rank = 0 processor = nid00346
rank = 1 processor = nid00346
rank = 2 processor = nid00347
rank = 3 processor = nid00347
rank = 4 processor = nid00348
rank = 5 processor = nid00348
rank = 6 processor = nid00349
rank = 7 processor = nid00349
% export MPICH_RANK_REORDER_METHOD=2
  % aprun -n 8 ./MPI_where
  rank = 0 processor = nid00346
  rank = 1 processor = nid00347
  rank = 2 processor = nid00348
  rank = 3 processor = nid00349
  rank = 4 processor = nid00349
  rank = 5 processor = nid00348
  rank = 6 processor = nid00347
  rank = 7 processor = nid00346
% cat MPICH_RANK_ORDER
   0,2,1,3,4,6,5,7
% export MPICH_RANK_REORDER_METHOD=3
% aprun -n 8 ./MPI_where
rank = 0 processor = nid00346
rank = 1 processor = nid00347
rank = 2 processor = nid00346
rank = 3 processor = nid00347
rank = 4 processor = nid00348
rank = 5 processor = nid00349
rank = 6 processor = nid00348
rank = 7 processor = nid00349
Timing With MPI_Wtime

- **Using MPI_WTIME**
  - You can compute the elapsed time between two points in an MPI program by using MPI_Wtime.
  - MPI_Wtime granularity is 0.000001 sec. (see MPI_Wtick). You cannot time any period that is smaller than a microsecond with it.
  - The clock in each node is independent of the clocks in other nodes.
  - MPI_WTIME_IS_GLOBAL has value=1 if MPI_WTIME is globally synchronized.
    - Default is 0.