Introduction to Parallel Programming with MPI

Mikhail Sekachev
Outline

- Message Passing Interface (MPI)
- Point to Point Communications
- Collective Communications
- Derived Datatypes
- Communicators and Groups
- MPI Tips and Hints
Collective Communications
Overview

• Generally speaking, collective calls are substitutes for a more complex sequence of point-to-point calls
• Involve all the processes in a process group
• Called by all processes in a communicator
• All routines block until they are locally complete
  – With MPI-3, collective operations can be blocking or non-blocking.
• Restrictions
  – Receive buffers must be exactly the right size
  – No message tags are needed
  – Can only be used with MPI predefined datatypes
Three Types of Collective Operations

• Synchronization
  – processes wait until all members of the group have reached the synchronization point.

• Data movement
  – broadcast, scatter/gather, all to all

• Global computation (reduction)
  – one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data
Synchronization Routine – MPI_Barrier

• To synchronize all processes within a communicator
• No processes in the communicator can pass the barrier until all of them call the function.

• C:
  ```
  ierr = MPI_Barrier(comm)
  ```

• Fortran:
  ```
  call MPI_Barrier(comm, ierr)
  ```
Data Movement Routine: MPI Broadcast

- One process broadcasts (sends) a message to all other processes in the group.
- The `MPI_Bcast` must be called by each node in a group, specifying the same communicator and root.

C:

```c
ierr = MPI_Bcast(buffer, count, datatype, root, comm)
```

Fortran:

```fortran
call MPI_Bcast(buffer, count, datatype, root, comm, ierr)
```
Data Movement Routine: MPI Scatter

- Distributes distinct messages from one process to all other processes in the group
- Data are distributed into $n$ equal segments, where the $i^{th}$ segment is sent to the $i^{th}$ process in the group, which contains all $n$ processes.

C:

```c
ierr = MPI_Scatter(&sbuff, scount, sdatatype, &rbuf, rcount, rdatatype, root, comm)
```

Fortran:

```fortran
call MPI_Scatter(sbuff, scount, sdatatype, rbuf, rcount, rdatatype, root, comm, ierr)
```
Example: \texttt{MPI\_Scatter}

\begin{verbatim}
real sbuf(12), rbuf(2)
call MPI\_Scatter(sbuf, 2, MPI\_INT, rbuf, 2, MPI\_INT, 3, MPI\_COMM\_WORLD, ierr)
\end{verbatim}
Scatter and Gather

DATA

PE 0
A0 A1 A2 A3 A4 A5

PE 1

PE 2

PE 3

PE 4

PE 5

scatter

DATA

PE 0
A0

A1

A2

A3

A4

A5

gather

Page 10
Data Movement Routine: MPI Gather

• Gathers distinct messages from each processes in the group to a single process in the order of process ranks
• The reverse operation of MPI_Scatter

• C :
  ierr = MPI_Gather(&sbuf, scount, sdatatype, &rbuf, rcount, rdatatype, root, comm)

• Fortran :
  call MPI_Gather(sbuff, scount, sdatatype, rbuff, rcount, rdatatype, root, comm, ierr)
Example: \texttt{MPI\_Gather}

\begin{verbatim}
real sbuf(2), rbuf(12)
call MPI\_Gather(sbuf, 2, MPI\_INT, rbuf, 2, MPI\_INT, 3, MPI\_COMM\_WORLD, ierr)
\end{verbatim}
**MPI_Scatterv and MPI_Gatherv**

- Allows varying count of data and flexibility for data placement

- C:

  ```c
  ierr = MPI_Scatterv( &sbuf, &scount, &displace, sdatatype, &rbuf, rcount, rdatatype, root, comm)
  ```

- Fortran:

  ```fortran
  call MPI_Scatterv(sbuf,scount,displace,sdatatype, rbuf, rcount, rdatatype, root, comm, ierr)
  ```
Data Movement Routine: **MPI_Allgater**

- Concatenation of data to all tasks in a group.
- Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.

**C:**

\[
ierr = \text{MPI\_Allgater}(\&\text{sbuf}, \&\text{scount}, \text{stype}, \&\text{rbuf}, \text{rcount}, \text{rtype}, \text{comm})
\]
Data Movement Routine: MPI_Alltoall

- Sends data from all to all processes

\[
\text{MPI\texttt{\_Alltoall}}(\texttt{sbuf}, \texttt{scount}, \texttt{stype}, \texttt{rbuff}, \texttt{rcount}, \texttt{rtype}, \texttt{comm})
\]

- \texttt{sbuf}: starting address of send buffer (*)
- \texttt{scount}: number of elements sent to each process
- \texttt{stype}: data type of send buffer
- \texttt{rbuff}: address of receive buffer (*)
- \texttt{rcount}: number of elements received from any process
- \texttt{rtype}: data type of receive buffer elements
- \texttt{comm}: communicator
**Example: MPI_Alltoall**

<table>
<thead>
<tr>
<th>PE 0</th>
<th>DATA</th>
<th>PE 0</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A1</td>
<td>B0</td>
<td>B1</td>
</tr>
<tr>
<td>A2</td>
<td>A3</td>
<td>B2</td>
<td>B3</td>
</tr>
<tr>
<td>A4</td>
<td>A5</td>
<td>B4</td>
<td>B5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B0</td>
<td>B1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>B3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>B5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>C1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>C3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>C5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td>D3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>D5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0</td>
<td>E1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>E3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E4</td>
<td>E5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>F1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td>F5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**alltoall**

<table>
<thead>
<tr>
<th>PE 0</th>
<th>DATA</th>
<th>PE 0</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>B0</td>
<td>C0</td>
<td>D0</td>
</tr>
<tr>
<td>B1</td>
<td>C1</td>
<td>D1</td>
<td>E0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F0</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
</tr>
<tr>
<td>A4</td>
<td>B4</td>
<td>C4</td>
<td>D4</td>
</tr>
<tr>
<td>A5</td>
<td>B5</td>
<td>C5</td>
<td>D5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F5</td>
</tr>
</tbody>
</table>

**COMM 0**

**COMM 1**

**COMM 2**

**COMM 3**

**COMM 4**

**COMM 5**
Global Computation Routines

- One process of the group collects data from the other processes and performs an operation (min, max, etc.) on that data.
- Basic MPI reduction operations are predefined.
- Users can also define their own reduction functions by using the MPI_Op_create routine.
- Examples:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation
## Predefined Reduction Operations

<table>
<thead>
<tr>
<th>MPI NAME</th>
<th>FUNCTION</th>
<th>MPI NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>MPI_MAXLOC</td>
<td>Maximum and location</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Bitwise AND</td>
<td>MPI_MINLOC</td>
<td>Minimum and location</td>
</tr>
</tbody>
</table>
Global Computation Routine: MPI_Reduce and MPI_Allreduce

**MPI_Reduce**

(sbuf, rbuf, count, stype, op, root, comm)

- Applies a reduction operation on all tasks in the group and **returns the result to one task**

**MPI_Allreduce**

(sbuf, rbuf, count, stype, op, comm)

- Applies a reduction operation on all tasks in the group and **returns the result to all tasks**

**Parameters**

- **sbuf**: address of send buffer
- **rbuf**: address of receive buffer
- **count**: the number of elements in the send buffer
- **stype**: the datatype of elements of send buffer
- **op**: the reduce operation function, predefined or user-defined
- **root**: the rank of the root process
- **comm**: communicator
**Example: MPI_Reduce**

**MPI_Reduce**

**Root process: 2**

<table>
<thead>
<tr>
<th>Proc: 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbuf:</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>rbuf:</td>
<td></td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

**MPI_Allreduce**

<table>
<thead>
<tr>
<th>Proc: 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbuf:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>rbuf:</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

**MPI_Reduce**(sbuf, rbuf, 1, MPI_INT, MPI_SUM, 2, MPI_COMM_WORLD)

**MPI_Allreduce**(sbuf, rbuf, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
Derived Datatypes
## Predefined (Basic) MPI Datatypes for C

<table>
<thead>
<tr>
<th>MPI Datatypes</th>
<th>C Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>--------</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>--------</td>
</tr>
</tbody>
</table>
MPI Derived Datatypes: Overview

- MPI allows you to define (derive) your own data structures based upon sequences of the MPI basic datatypes.
- Derived datatypes provide an efficient way of communicating mixed types or non-contiguous types in a single message
  - MPI-IO uses derived datatypes extensively
- MPI datatypes are created at run-time through calls to MPI library
- MPI provides several methods for constructing derived datatypes:
  - Contiguous
  - Vector
  - Indexed
  - Struct
Four MPI Datatype Constructors

**MPI Type contiguous** (count, oldtype, &newtype)
Produces a new data type by making count copies of an existing datatype oldtype.

**MPI Type vector** (count, blen, stride, oldtype, &newtype)
Similar to contiguous, but allows for regular gaps. ‘count’ blocks with ‘blen’ elements of ‘oldtype’ spaced by ‘stride’.

**MPI Type indexed** (count, blens[], strides[], oldtype, &newtype)
Extension of vector with varying ‘blens’ and ‘strides’.

**MPI Type struct** (count, blens[], strides[], oldtypes, &newtype)
Extension of indexed with varying oldtypes.
Example: MPI_Type_vector

- Replicates basic datatypes by placing blocks at fixed offsets.
  MPI_Type_vector(count, blocklen, stride, oldtype, &newtype)
- The new datatype consists of...
  - count number of blocks (nonnegative integer)
  - blocklen number of elements in each block (nonnegative integer)
  - stride number of elements between start of each block (integer)
  - oldtype old datatype (handle)
- Example
  - count = 4, blocklen = 1, stride = 4, oldtype = {MPI_FLOAT}

```c
float A[4][4];
int dest, tag;
MPI_Datatype newtype;

MPI_Type_vector( 4, /* number column elements */
                 1, /* 1 column only */
                 4, /* skip 4 elements */
                 MPI_FLOAT, /* elements are float */
                 &newtype); /* new MPI derived datatype */

MPI_Type_commit(&newtype);

MPI_Send(&A[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```
Communicators and Groups
MPI Communicators and Groups: MPI_COMM_WORLD

- MPI uses objects called *communicators* and *groups* to define which collection of processes may communicate with each other.

- All MPI communication calls require a *communicator argument* and MPI processes can only communicate if they share a communicator.

- **MPI_Init()** initializes a **default communicator**: MPI_COMM_WORLD

- The *base group* of MPI_COMM_WORLD contains all processes

- Process grouping capability allows the programmer to:
  - Organize tasks based upon application nature into task groups.
  - Enable Collective Communications operations across a subset of related tasks.
  - Provide basis for implementing virtual communication topologies.
MPI Communicators and Groups: Overview

• MPI Groups
  – A group is an ordered set of processes.
  – Each process in a group is associated with a unique integer rank
  – Rank values start at zero and go to N-1, where N is the number of processes in the group.
  – One process can belong to two or more groups.

• MPI Communicators
  – The communicator determines the scope and the "communication universe"
  – Each communicator contains a group of valid participants.

• Groups and communicators are dynamic objects in MPI and can be created and destroyed during program execution.
Every process has three communicating groups and a distinct rank associated to it.
MPI Communicators and Groups: Usage

- MPI provides over 40 routines related to groups, communicators, and virtual topologies.

- Typical usage:
  - Extract handle of global group from MPI_COMM_WORLD using `MPI_Comm_group`
  - Form new group as a subset of global group using `MPI_Group_incl`
  - Create new communicator for new group using `MPI_Comm_create`
  - Determine new rank in new communicator using `MPI_Comm_rank`
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using `MPI_Comm_free` and `MPI_Group_free`

- A Note on Virtual Topologies
  - Describes a mapping of MPI processes into a geometric "shape"
  - The two main types of topologies are Cartesian (grid) and Graph
  - Virtual topologies are built upon MPI communicators and groups.
Cray MPI Environment Variables

• Why use MPI environment variables?
  – Allow users to tweak optimizations for specific application behavior
  – Flexibility to choose cutoff values for collective optimizations
  – Determine maximum size of internal MPI resources - buffers/queues, etc.

• MPI Display Variables
  – `export MPICH_VERSION_DISPLAY=1`
    • Displays version of Cray MPI being used

  – `export MPICH_ENV_DISPLAY=1`
    • Displays all MPI environment variables and their current values
    • Helpful to determine what defaults are set to
# MPI Environment Variables: Default Values

The default values of MPI environment variables:

<table>
<thead>
<tr>
<th>Environment Variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI VERSION : CRAY MPICH2 XT version 3.1.2-pre (ANL base 1.0.6)</td>
<td></td>
</tr>
<tr>
<td>BUILD INFO : Built Thu Feb 26 3:58:36 2009 (svn rev 7308)</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH environment settings:</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ENV_DISPLAY = 1</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_VERSION_DISPLAY = 1</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ABORT_ON_ERROR = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_CPU_YIELD = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_RANK_REORDER_METHOD = 1</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_RANK_REORDER_DISPLAY = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MAX_THREAD_SAFETY = single</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MSGS_PER_PROC = 16384</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH/SMP environment settings:</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_SMP_OFF = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_SMPDEV_BUFS_PER_PROC = 32</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_SMP_SINGLE_COPY_SIZE = 131072</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_SMP_SINGLE_COPY_OFF = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH/PORTALS environment settings:</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MAX_SHORT_MSG_SIZE = 128000</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_UNEX_BUFFER_SIZE = 62914560</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_PTL_UNEX_EVENTS = 20480</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_PTL_OTHER_EVENTS = 2048</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH/VSHORT_OFF = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MAX_VSHORT_MSG_SIZE = 1024</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_VSHORT_BUFFERS = 32</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_PTL_EAGER_LONG = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_PTL_MATCH_OFF = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_PTL_SEND CREDITS = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH/COLLECTIVE environment settings:</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_FAST_MEMCPY = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_COLL_OPT_OFF = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_COLL_SYNC = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_BCAST_ONLY_TREE = 1</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ALLTOALL_SHORT_MSG = 1024</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_REDUCE_SHORT_MSG = 65536</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ALLREDUCE_LARGE_MSG = 262144</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ALLGATHER_VSHORT_MSG = 2048</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ALLTOALLVW_FCSIZE = 32</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ALLTOALLVW_SENDWIN = 20</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_ALLTOALLVW_RECVWIN = 20</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MPIIO environment settings:</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MPIIO_HINTS_DISPLAY = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MPIIO_CB_ALIGN = 0</td>
<td></td>
</tr>
<tr>
<td>PE 0: MPICH_MPIIO_HINTS = NULL</td>
<td></td>
</tr>
</tbody>
</table>
Dealing with errors

- **If you see this error message:**
  
  internal ABORT - process 0: Other MPI error, error stack:
  MPIDI_PortsU_Request_PUPE(317): exhausted unexpected receive queue
  buffering increase via env. var. MPICH_UNEX_BUFFER_SIZE

- **It means:**
  The application is sending too many short, unexpected messages to a particular receiver.

- **Try doing this to work around the problem:**
  Increase the amount of memory for MPI buffering using the MPICH_UNEX_BUFFER_SIZE variable (default is 60 MB) and/or decrease the short message threshold using the MPICH_MAX_SHORT_MSG_SIZE (default is 128000 bytes) variable.
Pre-posting receives

- If possible, pre-post receives before sender posts the matching send
  - typically useful technique for all MPICH installations

- But be careful with excessive pre-posting of the receives though, as it will hit Portals internal resource limitations eventually

  Error message
  [0] MPIDI_Portals_Progress: dropped event on "other" queue, increase
  [0] queue size by setting the environment variable MPICH_PTL_OTHER_EVENTS
  aborting job: Dropped Portals event

  Try doing this to work around the problem:
  You can increase the size of this queue by setting the environment variable
  MPICH_PTL_OTHER_EVENTS to some value higher than the 2048 default.
Aggregating data

• For very small buffers, aggregate data into fewer MPI calls (especially for collectives)
  – Example: alltoall with an array of 3 reals is clearly better than 3 alltoalls with 1 real
  – Do not aggregate too much. The MPI protocol switches from an short (eager) protocol to a long message protocol using a receiver pull method once the message is larger than the eager limit. This limit can be changed with the MPICH_MAX_SHORT_MSG_SIZE environment variable.
MPI Tips on Cray XT5

http://www.nics.tennessee.edu/computing-resources/kraken/mpi-tips-for-cray-xt5
Resources for Users: man pages and MPI web-sites

- There are man pages available for MPI which should be installed in your MANPATH. The following man pages have some introductory information about MPI.
  - `man MPI`
  - `man cc`
  - `man ftn`
  - `man qsub`
  - `man MPI_Init`
  - `man MPI_Finalize`

- MPI man pages are also available online.

- Main MPI web page at Argonne National Laboratory

- Set of guided exercises

- MPI tutorial at Lawrence Livermore National Laboratory
  [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)

- MPI Forum home page contains the official copies of the MPI standard.