Advanced MPI tutorial
George Bosilca
June 12, 2013
Outline

1. MPI
2. Datatypes
3. Hybrid Programming
   - Matching Probes
   - New Communicator Creation Functions
   - Shared Memory Windows
4. Nonblocking and Collective Communications
   - Nonblocking Communications
   - Nonblocking Collectives Communications
   - Neighborhood Collectives
Message Passing Interface

- MPI is not a programming model, it is an programming interface allowing data movement following different communication patterns.
  - However it imposes a very strict constraint *a rank is a process*.
- MPI is more like an assembly language providing concepts about moving data between entities.
  - a very complex assembly language
  - more than 300 functions so far without counting the profiling interface.
- The design is driven forward following 3 principles
  - Performance Portability
  - Composability (prevent interferences by isolation)
  - Tools Support (PMPI and now MPI-T)
Message Passing Interface

- MPI is an open evolving standard
  - 1.0 in 1994, 1.1 in 1995, 1.2 in 1997, 1.3 in 2008
  - 2.0 in 1997, 2.1 in 2008, 2.2 in 2009
  - 3.0 in 2012

- MPI continues to evolve in the context of the MPI Forum
  - Regular face-to-face meetings to discuss new concepts (once every 4 months)
  - Working groups focused on particular topics (collective, fault tolerance, ...)
  - Votes the new additions to the standard and release the official document

- More info at http://www.mpi-forum.org

- You are welcome to join on a regular basis or during the public comment period
MPI Concepts

- Groups and communicators
- Datatypes
- Topologies
- Communication concepts: point-to-point communications, collective communications, one-sided communications, Collective I/O Operations
- Process Management: dynamic processes
- Tools support
Type Constructors

- Heterogeneity and interoperability were primary goals (less important today)
- Allow the MPI implementation to know what data it handles.
- Cleaner code, provides opportunities to the MPI library to optimize the transfers.

- Implicit addresses: all displacements are computed in terms of number of derived datatypes
- Provide heterogeneous support for data transfers

- Explicit addresses: addresses computed in bytes
- The H* functions:
  - MPI_TYPE_CREATE_HVECTOR,
  - MPI_TYPE_CREATE_HINDEXED,
  - MPI_TYPE_CREATE_HINDEXED_BLOCK,
  - MPI_TYPE_CREATE_STRUCT

- MPI_GET_ADDRESS is used to retrieve the pointer (in Fortran)
- The predefined type MPI_Aint and INTEGER(KIND=MPI_ADDRESS_KIND) are used to store addresses.
### Predefined MPI datatypes

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char (treated as printable character)</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</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_LONG_LONG_INT</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_LONG (as a synonym)</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char (treated as integral value)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char (treated as integral value)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long 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_WCHAR</td>
<td>wchar_t (defined in <code>&lt;stddef.h&gt;</code>)</td>
</tr>
<tr>
<td>MPI_C_BOOL</td>
<td>_Bool (treated as printable character)</td>
</tr>
<tr>
<td>MPI_INT8_T</td>
<td>int8_t</td>
</tr>
<tr>
<td>MPI_INT16_T</td>
<td>int16_t</td>
</tr>
<tr>
<td>MPI_INT32_T</td>
<td>int32_t</td>
</tr>
<tr>
<td>MPI_INT64_T</td>
<td>int64_t</td>
</tr>
<tr>
<td>MPI_UINT8_T</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MPI_UINT16_T</td>
<td>uint16_t</td>
</tr>
<tr>
<td>MPI_UINT32_T</td>
<td>uint32_t</td>
</tr>
<tr>
<td>MPI_UINT64_T</td>
<td>uint64_t</td>
</tr>
<tr>
<td>MPI_C_FLOAT_COMPLEX</td>
<td>float _Complex</td>
</tr>
<tr>
<td>MPI_C_DOUBLE_COMPLEX</td>
<td>double _Complex</td>
</tr>
<tr>
<td>MPI_C_LONG_DOUBLE_COMPLEX</td>
<td>long double _Complex</td>
</tr>
</tbody>
</table>

- **Similar types matching** exists for Fortran and C++
- **Support for Posix types** (uint32_t) added in MPI-2.2
Derived Datatypes

A general datatype is an object that specifies a sequence of basic (predefined) datatypes, and a sequence of displacements.

- Typesig = \{ type_0, ..., type_{n-1} \}
- Typemap = \{ (type_0, disp_0), ..., (type_{n-1}, disp_{n-1}) \}

```c
struct {
    int a;
    char b;
    float c;
}
```

- Typesig = \{ int, char, double \}
- Typemap = \{ (int, 0), (char, 4), (float, 8) \}
Contiguous Types

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

- IN count  replication count (≥ 0)
- IN oldtype original type
- OUT newtype new type

- Allow replication of a datatype into contiguous locations.
- MPI_TYPE_CONTIGUOUS(3, type, newtype)
Vector Types

MPI_TYPE_VECTOR(count, blocklen, stride, oldtype, newtype)

- **IN** count: number of blocks ($\geq 0$)
- **IN** blocklen: number of elements in each block ($\geq 0$)
- **IN** stride: number of elements between start of each block
- **IN** oldtype: original type
- **OUT** newtype: new type

- Allow replication of a datatype into locations that consists of equally spaced blocks
- MPI_TYPE_VECTOR(3, 2, 3, type, newtype)
## Vector Types

**MPI_TYPE_HVECTOR(count, blocklen, stride, oldtype, newtype)**

<table>
<thead>
<tr>
<th>IN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>number of blocks ($\geq 0$)</td>
</tr>
<tr>
<td>blocklen</td>
<td>number of elements in each block ($\geq 0$)</td>
</tr>
<tr>
<td>stride</td>
<td>number of <strong>bytes</strong> between start of each block</td>
</tr>
<tr>
<td>oldtype</td>
<td>original type</td>
</tr>
<tr>
<td>OUT</td>
<td>new type</td>
</tr>
</tbody>
</table>

- Similar to MPI_TYPE_VECTOR except for the stride which is now in bytes instead of number of datatypes.
Indexed Types

MPI_TYPE_INDEXED(count, array_of_blocklen, array_of_disp, oldtype, newtype)

- **IN** count: number of blocks ($\geq 0$)
- **IN** array_of_blocklen: number of elements in each block ($[\ast] \geq 0$)
- **IN** array_of_disp: displacement for each block in number of elements
- **IN** oldtype: original type
- **OUT** newtype: new type

- Allow replication of a datatype into a sequence of blocks, where each block can contain a different number of copies and have a different displacement

```plaintext
MPI_TYPE_INDEXED(3, {1, 2, 1}, {1, 1, 2}, type, newtype)
```
Indexed Types

MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklen, array_of_disp, oldtype, newtype)

- **IN** count: number of blocks ($\geq 0$)
- **IN** array_of_blocklen: number of elements in each block ([*] $\geq 0$)
- **IN** array_of_disp: displacement for each block in number of bytes
- **IN** oldtype: original type
- **OUT** newtype: new type

- Similar to MPI_TYPE_INDEXED except for the displacements which are now in bytes instead of number of datatypes.
Indexed Types

MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklen, array_of_disp, oldtype, newtype)

- IN count number of blocks (\geq 0)
- IN blocklen number of elements per block (\geq 0)
- IN array_of_disp displacement for each block in number of elements
- IN oldtype original type
- OUT newtype new type

- Allow replication of a datatype into a sequence of blocks, where each block contain the same number of copies but have a different displacement

| MPI_TYPE_INDEXED( 3, 1, \{1, 1, 3\}, type, newtype) |
Indexed Types

MPI_TYPE_CREATE_HINDEXED_BLOCK(count, array_of_blocklen, array_of_disp, oldtype, newtype)

IN count number of blocks ($\geq 0$)
IN blocklen number of elements per block ($\geq 0$)
IN array_of_disp displacement of each block in number of bytes
IN oldtype original type
OUT newtype new type

Well, you got the idea ...
Structured Types

The most general datatype constructor, everything is variable including the datatypes used. There is no notion of multiple of number of elements here, all displacements are in bytes.

There is no H* version

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_TYPE_CREATE_STRUCT</td>
<td><code>count</code>, <code>array_of_blocklen</code>, <code>array_of_disp</code>,</td>
</tr>
<tr>
<td></td>
<td><code>array_of_types</code>, <code>newtype</code></td>
</tr>
<tr>
<td>IN count</td>
<td>number of blocks ($\geq 0$)</td>
</tr>
<tr>
<td>IN blocklen</td>
<td>number of elements per block ($\geq 0$)</td>
</tr>
<tr>
<td>IN array_of_disp</td>
<td>displacement of each block in number of bytes</td>
</tr>
<tr>
<td>IN array_of_types</td>
<td>type of elements in each block</td>
</tr>
<tr>
<td>OUT newtype</td>
<td>new type</td>
</tr>
</tbody>
</table>
Subarray Type

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

- **IN** ndims: number of array dimensions (≥ 0)
- **IN** array_of_sizes: number of elements of type `oldtype` in each dimension of the full array
- **IN** array_of_subsizes: number of elements of type `oldtype` in each dimension of the subarray
- **IN** array_of_starts: starting coordinates of the subarray in each dimension
- **IN** order: array storage order flag (column or row major)
- **IN** oldtype: original type
- **OUT** newtype: new type

- Create an MPI datatype describing an $n$-dimensional subarray of an $n$-dimensional array
- Facilitates creation of datatypes for accessing arrays distributed in blocks among processes to a single file
## Distributed Array Type

The `MPI_TYPE_CREATE_DARRAY` function is used to create the datatype corresponding to the distribution of an `ndims`-dimensional array of `oldtype` elements onto an `ndims`-dimensional array of logical processes. The function call is:

```c
MPI_TYPE_CREATE_DARRAY(array_of_distrib, array_of_dargs, array_of_psizes, order, oldtype, newtype)
```

- **IN** `size` - size of process group (> 0)
- **IN** `rank` - rank in the process group (>= 0)
- **IN** `ndims` - number of array and the process grid dimension (> 0)
- **IN** `array_of_gsizes` - number of elements of type `oldtype` in each dimension of the global array
- **IN** `array_of_distrib` - distribution of array in each dimension
- **IN** `array_of_dargs` - distribution argument in each dimension (block, cyclic, none)
- **IN** `array_of_psizes` - size of process grid in each dimension
- **IN** `order` - array storage order flag (column or row major)
- **IN** `oldtype` - original type
- **OUT** `newtype` - new type

Create the datatype corresponding to the distribution of an `ndims`-dimensional array of `oldtype` elements onto an `ndims`-dimensional array of logical processes.
Resized Datatypes

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN  oldtype    input datatype
IN   lb        new lower bound of datatype
IN   extent    new extent of datatype in bytes
OUT  newtype   new type

- Create a datatype having the same type signature as the input datatype, but a different type map (the lower bound and the extent have been altered)

- How to create the following type?

\[ \begin{array}{ccc} 
  & & \\
  & & \\
  & & 
\end{array} \]
Exercice 1

Create a datatype corresponding to the diagonal of a 8x8 matrix (as depicted by the red elements in the picture).
Exercice 1

Create a datatype corresponding to the diagonal of a 8x8 matrix (as depicted by the red elements in the picture).

- Can be done with an MPI_TYPE_CREATE_STRUCT
- Can be done with an MPI_TYPE_CREATE_INDEXED_BLOCK
- Can be done with an MPI_TYPE_CREATE_INDEXED_INDEXED
- Can be done with an MPI_TYPE_VECTOR
- Can be done with an MPI_TYPE_CONTIGUOUS?
Type Accessors

- MPI_TYPE_SIZE(MPI_Datatype type, int* size): retrieve the size of the datatype (bytes)
- MPI_TYPE_GET_EXTENT(MPI_Datatype type, MPI_Aint* lb, MPI_Aint* extent): get the extent of the datatype
- MPI_TYPE_GET_TRUE_EXTENT(MPI_Datatype type, MPI_Aint* true_lb, MPI_Aint* true_extent): get the true extent of the datatype
- MPI_TYPE_COMMIT(MPI_Datatype* type): validate the datatype for communications (potential optimization stage).
- MPI_TYPE_FREE(MPI_Datatype* type): Deallocation of the datatype. Does not affect other datatypes build using the freed type.
How about very large data?

- What is happening if the application needs to exchange more than $2^{31} - 1$ bytes?
  - One can divide the data in chunks of a usable size ($< 2^{31} - 1$) and then construct datatypes using these datatypes.
  - It might be a workable solution but ...
    - How do you know how much data have been received?
    - Isn’t MPI_GET_ELEMENTS(MPI_STATUS status, MPI_Datatype datatype, int* count) supposed to help?
    - Not as long as count argument is an int!

- A new set of accessors MPI_TYPE_*_X allowing the count to be of a configurable type MPI_Count
How about very large data?

- What is happening if the application needs to exchange more than $2^{31} - 1$ bytes?
  - One can divide the data in chunks of a usable size ($< 2^{31} - 1$) and then construct datatypes using these datatypes.
  - It might be a workable solution but ...
  - How do you know how much data have been received?
    - Isn’t MPI_GET_ELEMENTS(MPI_STATUS status, MPI_Datatype datatype, int* count) supposed to help?
    - Not as long as `count` argument is an int!

- A new set of accessors MPI_TYPE_*_X allowing the count to be of a configurable type MPI_Count
How about very large data?

- What is happening if the application needs to exchange more than $2^{31} - 1$ bytes?
  - One can divide the data in chunks of a usable size ($< 2^{31} - 1$) and then construct datatypes using these datatypes.
  - It might be a workable solution but ...
  - How do you know how much data have been received?
  - Isn’t MPI_GET_ELEMENTS(MPI_STATUS status, MPI_Datatype datatype, int* count) supposed to help?
  - Not as long as count argument is an int!

- A new set of accessors MPI_TYPE_\_*_X allowing the count to be of a configurable type MPI_Count
How about very large data?

What is happening if the application needs to exchange more than $2^{31} - 1$ bytes?
- One can divide the data in chunks of a usable size ($< 2^{31} - 1$) and then construct datatypes using these datatypes.
- It might be a workable solution but ...
- How do you know how much data have been received?
- Isn’t MPI_GET_ELEMENTS(MPI_STATUS status, MPI_Datatype datatype, int* count) supposed to help?
- Not as long as \texttt{count} argument is an int!

A new set of accessors MPI\_TYPE\_*\_X allowing the count to be of a configurable type MPI\_Count.
Augmented Type Accessors

- MPI_TYPE_SIZE_X(MPI_Datatype type, MPI_Count* size): retrieve the size of the datatype (bytes)
- MPI_TYPE_GET_EXTENT_X(MPI_Datatype type, MPI_Count* lb, MPI_Count* extent): get the extent of the datatype
- MPI_TYPE_GET_TRUE_EXTENT_X(MPI_Datatype type, MPI_Count* true_lb, MPI_Count* true_extent): get the true extent of the datatype
- MPI_GET_ELEMENTS_X(MPI_Status* status, MPI_Datatype datatype, MPI_Count* count)
Pack and Unpack

- Two function allowing the application to pack and unpack the data in contiguous buffers.
- The wrong way of using datatypes ...
Decoding datatypes

- Sometimes is important to know how a datatype was created (e.g. Libraries developers)
- Given a datatype can I determine how it was created?
- Given a datatype can I determine what memory layout it describe?
MPI_Type_get_enveloppe

MPI_Type_get_enveloppe( MPI_Datatype datatype, int *num_integers, int *num_addresses, int *num_datatypes, int *combiner );

The combiner field returns how the datatype was created, e.g.

- MPI_COMBINER_NAMED: basic datatype
- MPI_COMBINER_CONTIGUOUS: MPI_Type_contiguous
- MPI_COMBINER_VECTOR: MPI_Type_vector
- MPI_COMBINER_INDEXED: MPI_Type_indexed
- MPI_COMBINER_STRUCT: MPI_Type_struct

The other fields indicate how large the integer-array, the datatype-array, and the address-array has to be for the following call to MPI_Type_get_contents
**MPI_Type_get_contents**

- Call is erroneous for a predefined datatypes
- If returned data types are derived datatypes, then objects are duplicates of the original derived datatypes. User has to free them using MPI_Type_free
- The values in the integers, addresses and datatype arrays are depending on the original datatype constructor
Why? What? and How?

- Hardware become more and more complex (multi-cores, specialized cores, accelerators), with increasing level of hierarchies.
- Is there a place for MPI in this context? Should the MPI break the portability or the generality rule?
- Many solution proposed from threaded MPI to shared memory windows. So far no convergence in the MPI Forum toward a specific approach.
- However, MPI can offers helpers to programming models to integrate and blend together to improve the efficiency of parallel applications.
Threading in MPI 2.2

Some level of support exists MPI_Init_Thread(int* argc, char*** argv, int required, int* provided)

- **SINGLE** only one thread will execute
- **FUNNELED** the process may be multi-threaded, but the application will ensure only the main thread makes MPI calls
- **SERIALIZED** the process may be multi-threaded, and multiple threads can make MPI calls, but only one at a time
- **MULTIPLE** no restrictions, any number of threads can make MPI calls simultaneously.

MPI_QUERY_THREAD can help to retrieve the current level of thread support.
What’s new about threads in MPI-3.0

- A lot of clarification in the text, and one goodie
  - Safe multi-threading probe matching

- What didn’t make it?
  - Thread are MPI managed entities and have a common shared space (mpirun -np 2 -t 2 ./hello_world) (à la AMPI)
  - Thread as entities in the MPI world (not one rank per process but one endpoint per execution flow)
  - Sharing threads between programming approaches (helper threads provided by the upper level)
  - And few others ideas ...

- But few others things related to hybrid programming did succeed in entering in the MPI-3.0
Outline

1. MPI
2. Datatypes
3. Hybrid Programming
   - Matching Probes
     - New Communicator Creation Functions
     - Shared Memory Windows
4. Nonblocking and Collective Communications
   - Nonblocking Communications
   - Nonblocking Collectives Communications
   - Neighborhood Collectives
Probes (MPI_PROBE and MPI_IPROBE) are unusable in a threaded environment, if all threads query the same communicator with the same tag.

As the probe operation peek in the matching queues without retiring the matched element a race condition between the two threads is possible as the receive operation retires the message.

The user would have to protect the matching per communicator between the probe and the corresponding receive, decreasing the potential parallelism.
**Matching Probes**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_IMPROBE</code></td>
<td>Source, Tag, Communicator, Logical Operation State, Returned Message, Status</td>
</tr>
</tbody>
</table>

- **MPI_MPROBE** and **MPI_IMPROBE** provide a mechanism to receive the matched message regardless of other operations (the message is retired from the matching queues).
- **MPI_Message** is the placeholder of all the information regarding the message, and is manipulated by these routines.
- **Beware**: it is not because the matching was done that the data transfer is going on, as there is not yet a local buffer attached to the operation. This is the major difference with a receive operation.
Matched Receives

MPI_MRECV(buf, count, datatype, message, status)

- **OUT** buf: receive buffer
- **IN** count: number of elements in receive buffer
- **IN** datatype: datatype of each receive element
- **INOUT** message: message (MPI_Message)
- **OUT** status: status object

- Similar to the receive operation if the message is shorter than the receive buffer only the data corresponding to the shorter message are received.
- The message is destroyed upon return
- MPI_IMRECV is the nonblocking version of MPI_MRECV
Outline

1. MPI
2. Datatypes
3. Hybrid Programming
   - Matching Probes
   - New Communicator Creation Functions
   - Shared Memory Windows
4. Nonblocking and Collective Communications
   - Nonblocking Communications
   - Nonblocking Collectives Communications
   - Neighborhood Collectives
Non-collective communicator creation

MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)

- **IN** comm: communicator
- **IN** group: participating processes (subgroup of comm)
- **IN** tag: operation identification
- **OUT** newcomm: new communicator

- Allow communicator creation without involving all processes in the parent communicator
  - A drastic change from all MPI-2.x

- If the group is not empty then all processes in that group must call the function and provide the same arguments (congruent groups)

- Useful for dynamic programming (PGAS) or some type of fault tolerance
Communicator Duplication

- MPI_COMM_DUP_WITH_INFO(comm, info, newcomm): Same as MPI_COMM_DUP except that the info hints associated with `comm` are not duplicated, but instead replaced with the ones provided as argument.
- MPI_COMM_IDUP(comm, newcomm, request): the nonblocking version of the MPI_COMM_DUP
  - `newcomm` is not valid before the successful completion of the associated request.
  - attributes of `comm` will be copied as they were at the moment of the MPI_COMM_IDUP call.
  - crucial to development of nonblocking libraries.
Shared Memory Communicator Creation

- **MPI_COMM_SPLIT_TYPE**(comm, split_type, key, info, newcomm)
  - **IN** comm communicator
  - **IN** split_type criterion of grouping
  - **IN** key rank assignment control
  - **IN** info info argument
  - **OUT** newcomm new communicator

- Collective call partitioning the group associated with *comm* into disjoint subgroups based on the type of *split_type*.
- Two supported *split_type* values
  - **MPI_COMM_TYPE_SHARED** each subgroup can create a shared memory region (or they are on the same shared memory node).
  - **MPI_UNDEFINED** a process disqualify itself from the partitioning
- Conceptually similar to **MPI_COMM_SPLIT**(key, color)
Outline

1. MPI
2. Datatypes
3. Hybrid Programming
   - Matching Probes
   - New Communicator Creation Functions
   - Shared Memory Windows
4. Nonblocking and Collective Communications
   - Nonblocking Communications
   - Nonblocking Collectives Communications
   - Neighborhood Collectives
Shared Memory Window

MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win)

- **IN size**: size of the local window in bytes
- **IN disp_unit**: local unit size for displacements
- **IN info**: info argument (contiguous memory or not)
- **IN comm**: intra-communicator
- **OUT baseptr**: address of local allocated window segment
- **OUT win**: window returned by the call

- collective call that on each process i allocated at least size bytes that is shared among all processes in the communicator
- the locally allocated memory can be target to remote RMA operations
- the info key ALLOC_SHARED_NONCONTIG set to true allow for padding or special memory segments with potential for optimizations (think NUMA-aware allocation)
Shared Window Query

MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)

- **IN win**
  - shared memory window
- **IN rank**
  - rank in the group of the window
- **OUT size**
  - size of the window segment at rank
- **OUT disp_unit**
  - local unit size for displacements
- **OUT baseptr**
  - address for load/store access to window segment

- queries the process-local address for remote memory segments in a shared window
- remote address can only be computed locally if the allocated sizes are known
  - they are not if noncontig was used
  - baseptr are process-wide, they have no meaning in the context of another process (the window might have been attached to another base address)
Outline

1 MPI

2 Datatypes

3 Hybrid Programming
   Matching Probes
   New Communicator Creation Functions
   Shared Memory Windows

4 Nonblocking and Collective Communications
   Nonblocking Communications
   Nonblocking Collectives Communications
   Neighborhood Collectives
Nonblocking Communications

- Simple semantic: the function returns without a guarantee of progress
- Similar to the blocking versions but return an MPI_Request, which must be completed before the communication can be considered an completed
- The content of the buffers involved in the communication should not be altered until the completion of the corresponding request
- The request must be completed using
  - nonblocking tests such as MPI_Test (and friends: TestAny, TestSome, TestAll)
  - blocking calls such as MPI_Wait (and friends: WaitAny, WaitSome, WaitAll)
Nonblocking Communications

- Allow overlapping of communications and computation
- Can prevent deadlocks
- But they are susceptible to implementation details
  - Request Management overheads (always)
  - What is the underlying communication protocol used?
  - What is the switching limit between the eager and the rendez-vous mode? What is eager?
  - Is the network itself supporting asynchronous data movements?
  - Large messages may require extra hand-shakes in order to progress due to pipelining optimizations
Collective Communications

- Complex data movement patterns that can be highly optimized by the MPI library (in some cases there is hardware support)
- Belongs to three classes:
  - Data movers: Broadcast, Gather, Scatter, AlltoAll, AllGather
  - Reductions: Reduce, AllReduce, Scan, ExScan, ReduceScatter
  - Synchronization Barrier
- They all have blocking semantics, they don’t return until the local contribution is completed
- As they are blocking only one per communicator is allowed simultaneously
- Unlike point-to-point communications they have no need for tags
Outline

1. MPI
2. Datatypes
3. Hybrid Programming
   - Matching Probes
   - New Communicator Creation Functions
   - Shared Memory Windows
4. Nonblocking and Collective Communications
   - Nonblocking Communications
   - Nonblocking Collectives Communications
   - Neighborhood Collectives
Nonblocking Collectives Communications

- All collective have a corresponding nonblocking version (MPI_Ibarrier exists!)
- Semantic similar to nonblocking point-to-point: no guarantee of progress
- An MPI_Request is returned and should be completed
- Out-of-order completion is possible.

Restrictions
- The content of the buffers should not be altered
- They don’t match with blocking collectives
- Multiple outstanding nonblocking collectives are allowed, the matching is done in order.
Outline

1. MPI
2. Datatypes
3. Hybrid Programming
   - Matching Probes
   - New Communicator Creation Functions
   - Shared Memory Windows
4. Nonblocking and Collective Communications
   - Nonblocking Communications
   - Nonblocking Collectives Communications
   - Neighborhood Collectives
Topologies

- Cartesian (MPI_CART_CREATE and MPI_DIMS_CREATE) and Graphs (distributed or not)
- the non distributed topologies (MPI_GRAPH_CREATE) should not be used as they are non scalable (all processes must provide the entire topology)
- Distributed topologies: MPI_DIST_GRAPH_CREATE_ADJACENT

Process 0
- Indegree = 0
- Outdegree = 3
- Src = nil
- Dest = 3, 2, 1

Process 1
- Indegree = 3
- Outdegree = 1
- Src = 0, 3, 2
- Dest = 2
Topologies

- A more generic interface allowing each process to specify some of the edges (MPI_DIST_GRAPH_CREATE)
- All edges must appear at least once

Process 0
- N = 2
- Src = 0, 1
- Degrees = 2, 0
- Dest = 3, 2, 2

Process 1
- N = 3
- Src = 0, 3, 2
- Degrees = 1, 1, 1
- Dest = 1, 1, 1
Neighborhood Collectives

- The idea is to allow sparse collectives where each node only exchange data with the defined neighbors in the topology.
- Not all collective have such equivalent: MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV
- Order is determined by order of neighbors as returned by the accessor (dist_)graph_neighbors, thus each rank might have a different number of neighbors.
- Nonblocking versions of these functions exists.
MPI is wonderful!

- MPI is a rich set of concepts allowing highly efficient parallel applications
- MPI propose way more than 6 functions! Learn to use them wisely
- There are no magic in MPI but here are few tricks
  - Always post your receives before doing sends
  - When checking for completion do the minimal
  - If you expect MPI progress leave room for such progress to happen
- If your application does not perform as you expect blame MPI (that always worked for me even if I had to fix it after)
- Feel free to join the MPI Forum to evolutionarize or revolutionarize MPI