MPI for Scalable Computing (continued from yesterday)

Bill Gropp, University of Illinois at Urbana-Champaign
Rusty Lusk, Argonne National Laboratory
Rajeev Thakur, Argonne National Laboratory
Datatypes
Datatypes allow users to serialize **arbitrary** data layouts into a message stream

- Networks provide serial channels
- Same for block devices and I/O

Several constructors allow arbitrary layouts

- Recursive specification possible
- *Declarative* specification of data-layout
  - “what” and not “how”, leaves optimization to implementation (*many unexplored* possibilities!)
  - Choosing the right constructors is not always simple
Derived Datatype Example
MPI’s Intrinsic Datatypes

- **Why intrinsic types?**
  - Heterogeneity, nice to send a Boolean from C to Fortran
  - Conversion rules are complex, not discussed here
  - Length matches to language types
    - No sizeof(int) mess

- **Users should generally use intrinsic types as basic types for communication and type construction!**
  - MPI_BYTE should be avoided at all cost

- **MPI-2.2 added some missing C types**
  - E.g., unsigned long long
MPI_Type_contiguous

MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)
MPI_Type_vector

MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Specify strided blocks of data of oldtype
- Very useful for Cartesian arrays
MPI_Type_create_hvector

- Create non-unit strided vectors
- Useful for composition, e.g., vector of structs

MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_Type_indexed

MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Pulling irregular subsets of data from a single array (cf. vector collectives)
  - Dynamic codes with index lists, expensive though!
    - blen={1,1,2,1,2,1}
    - displs={0,3,5,9,13,17}
MPI_Type_create_indexed_block

Like Create_indexed but blocklength is the same

- blen=2
- displs={0,5,9,13,18}
MPI_Type_create_hindexed

MPI_Type_create_hindexed(int count, int *arr_of_blocklengths, MPI_Aint *arr_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Indexed with non-unit displacements, e.g., pulling types out of different arrays
MPI_Type_create_struct

MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[], MPI_Datatype *newtype)

- Most general constructor, allows different types and arbitrary arrays (also most costly)
**MPI_Type_create_subarray**

Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(1,0)</td>
<td>(2,0)</td>
<td>(3,0)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(1,1)</td>
<td>(2,1)</td>
<td>(3,1)</td>
</tr>
<tr>
<td>(0,2)</td>
<td>(1,2)</td>
<td>(2,2)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(0,3)</td>
<td>(1,3)</td>
<td>(2,3)</td>
<td>(3,3)</td>
</tr>
</tbody>
</table>
MPI_Type_create_darray

MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distrib[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Create distributed array, supports block, cyclic and no distribution for each dimension
  - Very useful for I/O

<table>
<thead>
<tr>
<th></th>
<th>(0,0)</th>
<th>(1,0)</th>
<th>(2,0)</th>
<th>(3,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>(1,1)</td>
<td>(2,1)</td>
<td>(3,1)</td>
<td></td>
</tr>
<tr>
<td>(0,2)</td>
<td>(1,2)</td>
<td>(2,2)</td>
<td>(3,2)</td>
<td></td>
</tr>
<tr>
<td>(0,3)</td>
<td>(1,3)</td>
<td>(2,3)</td>
<td>(3,3)</td>
<td></td>
</tr>
</tbody>
</table>
MPI_BOTTOM and MPI_Get_address

- **MPI_BOTTOM** is the absolute zero address
  - Portability (e.g., may be non-zero in globally shared memory)

- **MPI_Get_address**
  - Returns address relative to MPI_BOTTOM
  - Portability (do not use “&” operator in C!)

- Very important when
  - Building struct datatypes
  - Data spans multiple arrays
Commit, Free, and Dup

- Types must be committed before use
  - Only the ones that are used!
  - MPI_Type_commit may perform heavy optimizations (and will hopefully)

- MPI_Type_free
  - Free MPI resources of datatypes
  - Does not affect types built from it

- MPI_Type_dup
  - Duplicates a type
  - Library abstraction (composability)
Other Datatype Functions

- **Pack/Unpack**
  - Mainly for compatibility to legacy libraries
  - Avoid using it yourself

- **Get_envelope/contents**
  - Only for expert library developers
  - Libraries like MPITypes\(^1\) make this easier

- **MPI_Type_create_resized**
  - Change extent and size (dangerous but useful)

http://www.mcs.anl.gov/mpitypes/
Datatype Selection Order

- Simple and effective performance model:
  - More parameters == slower

- contig < vector < index_block < index < struct

- Some (most) MPIs are inconsistent
  - But this rule is portable

W. Gropp et al.: Performance Expectations and Guidelines for MPI Derived Datatypes
Collectives and Nonblocking Collectives
Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI_REDUCE** combines data from all processes in the communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECV** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.
MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- Non-blocking collective operations in MPI-3
- Three classes of operations: synchronization, data movement, collective computation
Synchronization

- **MPI_BARRIER(comm)**
  - Blocks until all processes in the group of communicator `comm` call it
  - A process cannot get out of the barrier until all other processes have reached barrier

- Note that a barrier is rarely, if ever, necessary in an MPI program
- Adding barriers “just to be sure” is a bad practice and causes unnecessary synchronization. **Remove unnecessary barriers from your code.**

- One legitimate use of a barrier is before the first call to MPI_Wtime to start a timing measurement. This causes each process to start at **approximately** the same time.
  - Avoid using barriers other than for this.
Collective Data Movement

Broadcast

P0
P1
P2
P3

A
A
A
A

Scatter

Gather

P0
P1
P2
P3

A
B
C
D

A
B
C
D
More Collective Data Movement

P0
A

P1
B

P2
C

P3
D

Allgather

P0
A

P1
B

P2
C

P3
D

P0
A0
A1
A2
A3

P1
B0
B1
B2
B3

P2
C0
C1
C2
C3

P3
D0
D1
D2
D3

Alltoall

P0
A

P1
B

P2
C

P3
D

P0
A0
B0
C0
D0

P1
A1
B1
C1
D1

P2
A2
B2
C2
D2

P3
A3
B3
C3
D3
Collective Computation

P0  A
P1  B
P2  C
P3  D

Reduce

ABCD

P0  A
P1  B
P2  C
P3  D

Scan

A
AB
ABC
ABCD
MPI Collective Routines

- Many Routines, including: MPI_ALLGATHER, MPI_ALLGATHERV, MPI_ALLREDUCE, MPI_ALLTOALL, MPI_ALLTOALLV, MPI_BCAST, MPI_EXSCAN, MPI_GATHER, MPI_GATHERV, MPI_REDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_SCATTER, MPI_SCATTERV

- "All" versions deliver results to all participating processes
- "V" versions (stands for vector) allow the chunks to have different sizes
- "W" versions for ALLTOALL allow the chunks to have different sizes in bytes, rather than units of datatypes

- MPI_ALLREDUCE, MPI_REDUCE, MPI_REDUCE_SCATTER, MPI_REDUCE_SCATTER_BLOCK, MPI_EXSCAN, and MPI_SCAN take both built-in and user-defined combiner functions
MPI Built-in Collective Computation Operations

- `MPI_MAX`  
  Maximum
- `MPI_MIN`  
  Minimum
- `MPI_PROD`  
  Product
- `MPI_SUM`  
  Sum
- `MPI_LAND`  
  Logical and
- `MPI_LOR`  
  Logical or
- `MPI_LXOR`  
  Logical exclusive or
- `MPI_BAND`  
  Bitwise and
- `MPI_BOR`  
  Bitwise or
- `MPI_BXOR`  
  Bitwise exclusive or
- `MPI_MAXLOC`  
  Maximum and location
- `MPI_MINLOC`  
  Minimum and location
- `MPI_REPLACE`  
  Replace and no operation (RMA)
- `MPI_NO_OP`
Defining your own Collective Operations

- Create your own collective computations with:
  
  ```
  MPI_OP_CREATE(user_fn, commutes, &op);
  MPI_OP_FREE(&op);
  
  user_fn(invec, inoutvec, len, datatype);
  ```

- The user function should perform:
  ```
  inoutvec[i] = invec[i] op inoutvec[i];
  for i from 0 to len-1
  ```

- The user function can be non-commutative, but must be associative
Nonblocking Collectives
Nonblocking Collective Communication

- Nonblocking communication
  - Deadlock avoidance
  - Overlapping communication/computation

- Collective communication
  - Collection of pre-defined optimized routines

- Nonblocking collective communication
  - Combines both advantages
  - System noise/imbalance resiliency
  - Semantic advantages
Nonblocking Communication

- Semantics are simple:
  - Function returns no matter what
  - No progress guarantee!

- E.g., `MPI_Isend(<send-args>, MPI_Request *req);`

- Nonblocking tests:
  - Test, Testany, Testall, Testsome

- Blocking wait:
  - Wait, Waitany, Waitall, Waitsome
Nonblocking Collective Communication

- Nonblocking variants of all collectives
  - MPI_Ibcast(<bcast args>, MPI_Request *req);

Semantics:
- Function returns no matter what
- No guaranteed progress (quality of implementation)
- Usual completion calls (wait, test) + mixing
- Out-of-order completion

Restrictions:
- No tags, in-order matching
- Send and vector buffers may not be touched during operation
- MPI_Cancel not supported
- No matching with blocking collectives
Nonblocking Collective Communication

- Semantic advantages:
  - Enable asynchronous progression (and manual)
    - Software pipelining
  - Decouple data transfer and synchronization
    - Noise resiliency!
  - Allow overlapping communicators
    - See also neighborhood collectives
  - Multiple outstanding operations at any time
    - Enables pipelining window
A Non-Blocking Barrier?

- What can that be good for? Well, quite a bit!

- Semantics:
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary

- Uses:
  - Overlap barrier latency (small benefit)
  - Use the split semantics! Processes **notify** non-collectively but **synchronize** collectively!
Nonblocking And Collective Summary

- Nonblocking comm does two things:
  - Overlap and relax synchronization

- Collective comm does one thing
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance

- They can be composed
  - E.g., software pipelining