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
Costs of Unintended Synchronization
Unexpected Hot Spots

- Even simple operations can give surprising performance behavior.
- Examples arise even in common grid exchange patterns.
- Message passing illustrates problems present even in shared memory:
  - Blocking operations may cause unavoidable stalls.
Mesh Exchange

- Exchange data on a mesh
Sample Code

- Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL,&
  nbr(i), tag,comm, ierr)
Enddo

Do i=1,n_neighbors
  Call MPI_Recv(edge(1,i), len, MPI_REAL,&
  nbr(i), tag, comm, status, ierr)
Enddo
Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of
  
  if (has down nbr) then
      Call MPI_Send( ... down ... )
  endif

  if (has up nbr) then
      Call MPI_Recv( ... up ... )
  endif

  ...

  sequentializes (all except the bottom process blocks)
## Sequentialization

<table>
<thead>
<tr>
<th></th>
<th>Start Send</th>
<th>Send</th>
<th>Recv</th>
<th>Start Send</th>
<th>Send</th>
<th>Recv</th>
<th>Start Send</th>
<th>Send</th>
<th>Recv</th>
<th>Start Send</th>
<th>Send</th>
<th>Recv</th>
<th>Send</th>
<th>Recv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start Send</td>
<td></td>
<td></td>
<td>Start Send</td>
<td></td>
<td></td>
<td>Start Send</td>
<td></td>
<td></td>
<td>Start Send</td>
<td></td>
<td></td>
<td>Send</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send</td>
<td></td>
<td></td>
<td>Recv</td>
<td></td>
<td></td>
<td>Send</td>
<td></td>
<td></td>
<td>Recv</td>
<td></td>
<td></td>
<td>Recv</td>
<td></td>
</tr>
</tbody>
</table>
Fix 1: Use Irecv

- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(i), ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, ierr)
Enddo
Call MPI_Waitall(n_neighbors, requests, statuses, ierr)

- Does not perform well in practice. Why?
Understanding the Behavior: Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)
Mesh Exchange - Step 1

- Exchange data on a mesh
Mesh Exchange - Step 2

- Exchange data on a mesh
Mesh Exchange - Step 3

- Exchange data on a mesh
Mesh Exchange - Step 4

- Exchange data on a mesh
Mesh Exchange - Step 5

- Exchange data on a mesh
Mesh Exchange - Step 6

- Exchange data on a mesh
• Note that process 1 finishes last, as predicted
Distribution of Sends

'SEND' state length distribution

(in seconds)
68 states of 96 (70%)
Why Six Steps?

- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory
Fix 2: Use Isend and Irecv

- Do \(i = 1, n_{\text{neighbors}}\)
  
  Call `MPI_Irecv(inedge(1,i),\text{len},\text{MPI\_REAL},\text{nbr}(i),\text{tag},&\text{comm, requests}(i),\text{ierr})`

  Enddo

- Do \(i = 1, n_{\text{neighbors}}\)
  
  Call `MPI_Isend(edge(1,i),\text{len},\text{MPI\_REAL},\text{nbr}(i),\text{tag},&\text{comm, requests}(n_{\text{neighbors}}+i),\text{ierr})`

  Enddo

Call `MPI_Waitall(2*n_{\text{neighbors}}, \text{requests, statuses, ierr})`
Mesh Exchange - Steps 1-4

- Four interleaved steps
Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors
Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization

- In many cases, there is more synchronization than required

- Use nonblocking operations and MPI_Waitall to defer synchronization
Datatypes
Introduction to Datatypes in MPI

- 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

contig.  contig.  contig.  indexed

vector  struct
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_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

```
struct
struct
struct
```
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

```c
MPI_Type_create_darray(int size, int rank, int ndims,
int array_of_gsizes[], int array_of_distribs[], 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

<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_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 to
  - build struct datatypes
  - If 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)

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 is to ensure that all processes start that portion of the code at the same time.

- Avoid using barriers other than for this.
Collective Data Movement

Broadcast

Scatter

Gather
More Collective Data Movement

---

**Allgather**

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Alltoall**

<table>
<thead>
<tr>
<th>P0</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
</tr>
<tr>
<td>P2</td>
<td>C0</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
</tr>
<tr>
<td>P3</td>
<td>D0</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
</tr>
</tbody>
</table>
Collective Computation

P0
P1
P2
P3

P0
P1
P2
P3

Reduce

Scan

ABCD

A
AB
ABC
ABCD
MPI Collective Routines

- Many Routines: `MPI_ALLGATHER`, `MPI_ALLGATHERV`, `MPI_ALLREDUCE`, `MPI_ALLTOALL`, `MPI_ALLTOALLV`, `MPI_BCAST`, `MPI_GATHER`, `MPI_GATHERV`, `MPI_REDUCE`, `MPI_REDUCESCATTER`, `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
- `MPI_ALLREDUCE`, `MPI_REDUCE`, `MPI_REDUCESCATTER`, 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
Defining your own Collective Operations

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

- The user function should perform:
  ```c
  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
One-Sided Communication
One-Sided Communication

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory
Comparing One-sided and Two-sided Programming

Even the sending process is delayed

Delay in process 1 does not affect process 0

Even the sending process is delayed

Delay in process 1 does not affect process 0
Advantages of RMA Operations

- Can do multiple data transfers with a single synchronization operation
  - like BSP model

- Bypass tag matching
  - effectively precomputed as part of remote offset

- Some irregular communication patterns can be more economically expressed

- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems
Irregular Communication Patterns with RMA

- If communication pattern is not known *a priori*, the send-receive model requires an extra step to determine how many sends-receives to issue.
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call.
- This makes dynamic communication easier to code in RMA.
What we need to know in MPI RMA

- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Model
Creating Public Memory

- Any memory created by a process is, by default, only locally accessible
  - `X = malloc(100);`
- Once the memory is created, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a “window”
  - A group of processes collectively create a “window”
- Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process
Remote Memory Access Windows and Window Objects

Process 0

Process 1

Get

Put

Process 2

Process 3

= address spaces

= window object

window
Basic RMA Functions for Communication

- **MPI_Win_create** exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- **MPI_Win_free** deallocates window object

- **MPI_Put** moves data from local memory to remote memory
- **MPI_Get** retrieves data from remote memory into local memory
- **MPI_Accumulate** updates remote memory using local values
- Data movement operations are non-blocking
- **Subsequent synchronization on window object** needed to ensure operation is complete
Window creation models

- Four models exist
  - MPI_WIN_CREATE
    • You already have an allocated buffer that you would like to make remotely accessible
  - MPI_WIN_ALLOCATE
    • You want to create a buffer and directly make it remotely accessible
  - MPI_WIN_CREATE_DYNAMIC
    • You don’t have a buffer yet, but will have one in the future
  - MPI_WIN_ALLOCATE_SHARED
    • You want multiple processes on the same node share a buffer
    • We will not cover this model today
**MPI_WIN_CREATE**

```c
int MPI_Win_create(void *base, MPI_Aint size,
                    int disp_unit, MPI_Info info,
                    MPI_Comm comm, MPI_Win *win)
```

- Expose a region of memory in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.

- Arguments:
  - base - pointer to local data to expose
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
Example with MPI_WIN_CREATE

```c
int main(int argc, char ** argv)
{
    int *a;       MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                   MPI_COMM_WORLD, &win);

    /* Array ‘a’ is now accessibly by all processes in
     * MPI_COMM_WORLD */
    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```
MPI_WIN_ALLOCATE

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
                     MPI_Comm comm, void *baseptr, MPI_Win *win)
```

- Create a remotely accessible memory region in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.

- Arguments:
  - `size` - size of local data in bytes (nonnegative integer)
  - `disp_unit` - local unit size for displacements, in bytes (positive integer)
  - `info` - info argument (handle)
  - `comm` - communicator (handle)
  - `baseptr` - pointer to exposed local data
Example with MPI_WIN_ALLOCATE

```c
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* collectively create remotely accessible memory in the window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                     MPI_COMM_WORLD, &a, &win);

    /* Array `a` is now accessible by all processes in
     * MPI_COMM_WORLD */
    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```
MPI_WIN_CREATE_DYNAMIC

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>MPI_Win_create_dynamic(…, MPI_Comm comm, MPI_Win *win)</td>
</tr>
</tbody>
</table>

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops
- Application can dynamically attach memory to this window
- Application can access data on this window only after a memory region has been attached
Example with `MPI_WIN_CREATE_DYNAMIC`

```c
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);
    MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* create private memory */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000*sizeof(int));

    /*Array 'a' is now accessibly by all processes in MPI_COMM_WORLD*/

    /* undeclare public memory */
    MPI_Win_detach(win, a);
    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```
Data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - MPI_GET
  - MPI_PUT
  - MPI_ACCUMULATE
  - MPI_GET_ACCUMULATE
  - MPI_COMPARE_AND_SWAP
  - MPI_FETCH_AND_OP
Data movement: Get

MPI_Get(
    origin_addr, origin_count, origin_datatype,
    target_rank,
    target Disp, target_count, target_datatype,
    win)

- Move data to origin, from target
- Separate data description triples for origin and target
Data movement: *Put*

MPI_Put(
    origin_addr, origin_count, origin_datatype,
    target_rank,
    target_disp, target_count, target_datatype,
    win)

- Move data from origin, to target
- Same arguments as MPI_Get
Data aggregation: *Accumulate*

- Like MPI_Put, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result ends up at target buffer
- Different data layouts between target/origin OK, basic type elements must match
- Put-like behavior with MPI_REPLACE (implements \(f(a,b)=b\))
  - Atomic PUT
Data aggregation: *Get Accumulate*

- Like MPI_Get, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result at target buffer; original data comes to the source
- Different data layouts between target/origin OK, basic type elements must match
- Get-like behavior with MPI_NO_OP
  - Atomic GET
Ordering of Operations in MPI RMA

- For Put/Get operations, ordering does not matter
  - If you do two concurrent PUTs to the same location, the result can be garbage

- Two accumulate operations to the same location are valid
  - If you want “atomic PUTs”, you can do accumulates with MPI_REPLACE

- All accumulate operations are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hints
  - You can ask for “read-after-write” ordering, “write-after-write” ordering, or “read-after-read” ordering
Additional Atomic Operations

- **Compare-and-swap**
  - Compare the target value with an input value; if they are the same, replace the target with some other value
  - Useful for linked list creations – if next pointer is NULL, do something

- **Fetch-and-Op**
  - Special case of Get accumulate for predefined datatypes – faster for the hardware to implement
RMA Synchronization Models

- **RMA data visibility**
  - When is a process allowed to read/write from remotely accessible memory?
  - How do I know when data written by process X is available for process Y to read?
  - RMA synchronization models provide these capabilities

- **MPI RMA model allows data to be accessed only within an “epoch”**
  - Three types of epochs possible:
    - Fence (active target)
    - Post-start-complete-wait (active target)
    - Lock/Unlock (passive target)

- **Data visibility is managed using RMA synchronization primitives**
  - MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL
  - Epochs also perform synchronization
Fence Synchronization

- MPI_Win_fence(assert, win)
- Collective synchronization model -- assume it synchronizes like a barrier
- Starts *and* ends access & exposure epochs (usually)
- Everyone does an MPI_WIN_FENCE to open an epoch
- Everyone issues PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
PSCW Synchronization

- **Target**: Exposure epoch
  - Opened with MPI_Win_post
  - Closed by MPI_Win_wait
- **Origin**: Access epoch
  - Opened by MPI_Win_start
  - Closed by MPI_Win_compete
- All may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets
- Like FENCE, but the target may allow a smaller group of processes to access its data
Lock/Unlock Synchronization

- Passive mode: One-sided, *asynchronous* communication
  - Target does **not** participate in communication operation
- Shared memory like model
Passive Target Synchronization

- Begin/end passive mode epoch
  - Doesn’t function like a mutex, name can be confusing
  - Communication operations within epoch are all nonblocking

- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

```c
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
int MPI_Win_unlock(int rank, MPI_Win win)
```
When should I use passive mode?

- **RMA performance advantages from low protocol overheads**
  - Two-sided: Matching, queueing, buffering, unexpected receives, etc...
  - Direct support from high-speed interconnects (e.g. InfiniBand)

- **Passive mode: asynchronous one-sided communication**
  - Data characteristics:
    - Big data analysis requiring memory aggregation
    - Asynchronous data exchange
    - Data-dependent access pattern
  - Computation characteristics:
    - Adaptive methods (e.g. AMR, MADNESS)
    - Asynchronous dynamic load balancing

- **Common structure: shared arrays**
Topology Mapping and Neighborhood Collectives
Topology Mapping Basics

- First type: Allocation mapping
  - Up-front specification of communication pattern
  - Batch system picks good set of nodes for given topology

- Properties:
  - Not widely supported by current batch systems
  - Either predefined allocation (BG/P), random allocation, or “global bandwidth maximation”
  - Also problematic to specify communication pattern upfront, not always possible (or static)
Topology Mapping Basics

- Rank reordering
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)

- Properties
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies
    - Network topology is not exposed
  - Manual data shuffling after remapping step
On-Node Reordering

Naïve Mapping

Optimized Mapping

Topomap

Koutsonas/He@Fl: 343791 nodes, 15248661 edges.

Gottschling and Hoefler: Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption
Off-Node (Network) Reordering

Application Topology

Naïve Mapping

Network Topology

Optimal Mapping

Topomap
MPI Topology Intro

- Convenience functions (in MPI-1)
  - Create a graph and query it, nothing else
  - Useful especially for Cartesian topologies
    - Query neighbors in n-dimensional space
  - Graph topology: each rank specifies full graph 😞

- Scalable Graph topology (MPI-2.2)
  - Graph topology: each rank specifies its neighbors or an arbitrary subset of the graph

- Neighborhood collectives (MPI-3.0)
  - Adding communication functions defined on graph topologies (neighborhood of distance one)
**MPI_Cart_create**

MPI_Cart_create(MPI_Comm comm_old, int ndims, const int *dims, const int *periods, int reorder, MPI_Comm *comm_cart)

- Specify ndims-dimensional topology
  - Optionally periodic in each dimension (Torus)
- Some processes may return MPI_COMM_NULL
  - Product of dims must be <= P
- Reorder argument allows for topology mapping
  - Each calling process may have a new rank in the created communicator
  - Data has to be remapped manually
MPI_Cart_create Example

```c
int dims[3] = {5,5,5};
int periods[3] = {1,1,1};
MPI_Comm toppocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- But we’re starting MPI processes with a one-dimensional argument (-p X)
  - User has to determine size of each dimension
  - Often as “square” as possible, MPI can help!
MPI_Dims_create

MPI_Dims_create(int nnodes, int ndims, int *dims)

- Create dims array for Cart_create with nnodes and ndims
  - Dimensions are as close as possible (well, in theory)
- Non-zero entries in dims will not be changed
  - nnodes must be multiple of all non-zeroes
MPI_Dims_create Example

```c
int p;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Dims_create(p, 3, dims);

int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Makes life a little bit easier
  - Some problems may be better with a non-square layout though
Cartesian Query Functions

- Library support and convenience!
- `MPI_Cartdim_get()`
  - Gets dimensions of a Cartesian communicator
- `MPI_Cart_get()`
  - Gets size of dimensions
- `MPI_Cart_rank()`
  - Translate coordinates to rank
- `MPI_Cart_coords()`
  - Translate rank to coordinates
Cartesian Communication Helpers

- Shift in one dimension
  - Dimensions are numbered from 0 to ndims-1
  - Displacement indicates neighbor distance (-1, 1, ...)
  - May return MPI_PROC_NULL

- Very convenient, all you need for nearest neighbor communication

```c
MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
```
**MPI_Graph_create**

- Don’t use! Use one of the Dist_graph functions instead

```c
MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int *index, const int *edges, int reorder, MPI_Comm *comm_graph)
```

- nnodes is the total number of nodes
- index[i] stores the total number of neighbors for the first i nodes (sum)
  - Acts as offset into edges array
- edges stores the edge list for all processes
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges
Distributed graph constructor

- MPI_Graph_create is discouraged
  - Not scalable
  - Not deprecated yet but hopefully soon

- New distributed interface:
  - Scalable, allows distributed graph specification
    - Either local neighbors or any edge in the graph
  - Specify edge weights
    - Meaning undefined but optimization opportunity for vendors!
  - Info arguments
    - Communicate assertions of semantics to the MPI library
    - E.g., semantics of edge weights
MPI_Dist_graph_create_adjacent

MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree, const int sources[], const int sourceweights[], int outdegree, const int destinations[], const int destweights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)

- outdegree, destinations, weights – dest. proc. spec.
- info, reorder, comm_dist_graph – as usual
- directed graph
- Each edge is specified twice, once as out-edge (at the source) and once as in-edge (at the dest)
MPI_Dist_graph_create_adjacent

- Process 0:
  - Indegree: 0
  - Outdegree: 2
  - Dests: \{3,1\}

- Process 1:
  - Indegree: 3
  - Outdegree: 2
  - Sources: \{4,0,2\}
  - Dests: \{3,4\}

- ...
MPI_Dist_graph_create

MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[], const int degrees[], const int destinations[], const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)

- n – number of source nodes
- sources – n source nodes
- degrees – number of edges for each source
- destinations, weights – dest. processor specification
- info, reorder – as usual
- More flexible and convenient
  - Requires global communication
  - Slightly more expensive than adjacent specification

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2
**MPI_Dist_graph_create**

- **Process 0:**
  - $N$: 2
  - Sources: \{0,1\}
  - Degrees: \{2,2\}
  - Dests: \{3,1,4,3\}

- **Process 1:**
  - $N$: 2
  - Sources: \{2,3\}
  - Degrees: \{1,1\}
  - Dests: \{1,2\}

- ...
Distributed Graph Neighbor Queries

- **MPI_Dist_graph_neighbors_count()**
  - Query the number of neighbors of *calling process*
  - Returns indegree and outdegree!
  - Also info if weighted

- **MPI_Dist_graph_neighbors()**
  - Query the neighbor list of *calling process*
  - Optionally return weights
Further Graph Queries

- Status is either:
  - MPI_GRAPH (ugs)
  - MPI_CART
  - MPI_DIST_GRAPH
  - MPI_UNDEFINED (no topology)

- Enables to write libraries on top of MPI topologies!
Algorithms and Topology

- Complex hierarchy:
  - Multiple chips per node; different access to local memory and to interconnect; multiple cores per chip
  - Mesh has different bandwidths in different directions
  - Allocation of nodes may not be regular (you are unlikely to get a compact brick of nodes)
  - Some nodes have GPUs

- Most algorithms designed for simple hierarchies and ignore network issues

Recent work on general topology mapping e.g.,
Generic Topology Mapping Strategies for Large-scale Parallel Architectures, Hoefler and Snir
Dynamic Workloads Require New, More Integrated Approaches

- Performance irregularities mean that classic approaches to decomposition are increasingly ineffective
  - Irregularities come from OS, runtime, process/thread placement, memory, heterogeneous nodes, power/clock frequency management

- Static partitioning tools can lead to persistent load imbalances
  - Mesh partitioners have incorrect cost models, no feedback mechanism
  - “Regrid when things get bad” won’t work if the cost model is incorrect; also costly

- Basic building blocks must be more dynamic without introducing too much overhead
Communication Cost Includes More than Latency and Bandwidth

- Communication does not happen in isolation
- Effective bandwidth on shared link is $\frac{1}{2}$ point-to-point bandwidth
- Real patterns can involve many more (integer factors)
- Loosely synchronous algorithms ensure communication cost is worst case
Halo Exchange on BG/Q and Cray XE6

- 2048 doubles to each neighbor
- Rate is MB/sec (for all tables)

<table>
<thead>
<tr>
<th>BG/Q</th>
<th>8 Neighbors</th>
<th>Irecv/_send</th>
<th>Irecv/Isend</th>
</tr>
</thead>
<tbody>
<tr>
<td>World</td>
<td>662</td>
<td>1167</td>
<td></td>
</tr>
<tr>
<td>Even/Odd</td>
<td>711</td>
<td>1452</td>
<td></td>
</tr>
<tr>
<td>1 sender</td>
<td></td>
<td>2873</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cray XE6</th>
<th>8 Neighbors</th>
<th>Irecv/_send</th>
<th>Irecv/Isend</th>
</tr>
</thead>
<tbody>
<tr>
<td>World</td>
<td>352</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>Even/Odd</td>
<td>338</td>
<td>324</td>
<td></td>
</tr>
<tr>
<td>1 sender</td>
<td></td>
<td>5507</td>
<td></td>
</tr>
</tbody>
</table>
Discovering Performance Opportunities

- Let's look at a single process sending to its neighbors.
- Based on our performance model, we expect the rate to be roughly twice that for the halo (since this test is only sending, not sending and receiving)

<table>
<thead>
<tr>
<th>System</th>
<th>4 neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG/L</td>
<td>488</td>
<td>Periodic</td>
</tr>
<tr>
<td></td>
<td>490</td>
<td>389</td>
</tr>
<tr>
<td>BG/P</td>
<td>1139</td>
<td>1136</td>
</tr>
<tr>
<td>BG/Q</td>
<td>1136</td>
<td></td>
</tr>
<tr>
<td>XT3</td>
<td>1005</td>
<td>1007</td>
</tr>
<tr>
<td>XT4</td>
<td>1634</td>
<td>1620</td>
</tr>
<tr>
<td>XE6</td>
<td></td>
<td>5507</td>
</tr>
</tbody>
</table>
Discovering Performance Opportunities

- Ratios of a single sender to all processes sending (in rate)
- *Expect* a factor of roughly 2 (since processes must also receive)

<table>
<thead>
<tr>
<th>System</th>
<th>4 neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Periodic</td>
<td>Periodic</td>
</tr>
<tr>
<td>BG/L</td>
<td>2.24</td>
<td>2.01</td>
</tr>
<tr>
<td>BG/P</td>
<td>3.8</td>
<td>2.2</td>
</tr>
<tr>
<td>BG/Q</td>
<td></td>
<td>1.98</td>
</tr>
<tr>
<td>XT3</td>
<td>7.5</td>
<td>8.1</td>
</tr>
<tr>
<td></td>
<td>9.08</td>
<td>9.41</td>
</tr>
<tr>
<td>XT4</td>
<td>10.7</td>
<td>10.7</td>
</tr>
<tr>
<td></td>
<td>13.0</td>
<td>13.7</td>
</tr>
<tr>
<td>XE6</td>
<td></td>
<td>15.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.9</td>
</tr>
</tbody>
</table>

- BG gives roughly double the halo rate. XTn and XE6 are much higher.
  - It should be possible to improve the halo exchange on the XT by scheduling the communication
  - Or improving the MPI implementation
Neighborhood Collectives
Neighborhood Collectives

- Topologies implement no communication!
  - Just helper functions
- Collective communications only cover some patterns
  - E.g., no stencil pattern
- Several requests for “build your own collective” functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication patterns!
Cartesian Neighborhood Collectives

- Communicate with direct neighbors in Cartesian topology
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
  - Buffers are laid out as neighbor sequence:
    - Defined by order of dimensions, first negative, then positive
    - 2*ndims sources and destinations
    - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!
Cartesian Neighborhood Collectives

- Allgather
- Buffer ordering example:

![Diagram of Cartesian Neighborhood Collectives](image)
Graph Neighborhood Collectives

- Collective Communication along arbitrary neighborhoods
  - Order is determined by order of neighbors as returned by (dist_) graph_neighbors.
  - Distributed graph is directed, may have different numbers of send/recv neighbors
  - Can express dense collective operations 😊
  - Any persistent communication pattern!
MPI_Neighbor_allgather

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

- Sends the same message to all neighbors
- Receives indegree distinct messages
- Similar to MPI_Gather
  - The all prefix expresses that each process is a “root” of his neighborhood
- Also a vector “v” version for full flexibility
MPI_Neighbor_alltoall

MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

- Sends outdegree distinct messages
- Received indegree distinct messages
- Similar to MPI_Alltoall
  - Neighborhood specifies full communication relationship
- Vector and w versions for full flexibility
Nonblocking Neighborhood Collectives

- Very similar to nonblocking collectives
- Collective invocation
- Matching in-order (no tags)
  - No wild tricks with neighborhoods! In order matching per communicator!
Topology Summary

- Topology functions allow users to specify application communication patterns/topology
  - Convenience functions (e.g., Cartesian)
  - Storing neighborhood relations (Graph)
- Enables topology mapping (reorder=1)
  - Not widely implemented yet
  - May requires manual data re-distribution (according to new rank order)
- MPI does not expose information about the network topology (would be very complex)
Neighborhood Collectives Summary

- Neighborhood collectives add communication functions to process topologies
  - Collective optimization potential!

- Allgather
  - One item to all neighbors

- Alltoall
  - Personalized item to each neighbor

- High optimization potential (similar to collective operations)
  - Interface encourages use of topology mapping!
Acknowledgments

- Thanks to Torsten Hoefler and Pavan Balaji for some of the slides in this tutorial