MPI for Scalable Computing

Pavan Balaji,¹ Bill Gropp,² Rajeev Thakur¹

¹Argonne National Laboratory
²University of Illinois
The MPI Part of ATPESC

- We assume everyone already has some MPI experience
- We will focus more on understanding MPI concepts than on coding details
- Emphasis will be on issues affecting scalability and performance
- There will be code walkthroughs and hands-on exercises
Outline

- **Morning**
  - Introduction to MPI
  - Performance issues in MPI programs
  - Sources of scalability problems
  - Avoiding communication delays
    - understanding synchronization
  - Minimizing data motion
    - using MPI datatypes
  - Topics in collective communication
  - Hands-on exercises

- **Afternoon**
  - Using remote memory access to avoid extra synchronization and data motion
  - Hands-on exercises
  - Hybrid programming
  - Process topologies

- **After dinner**
  - Hands-on exercises
What is MPI?

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
  - Classical message-passing programming model

- MPI-1 was defined (1994) by a broadly-based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process

- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.

- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)
Timeline of the MPI Standard

- **MPI-1 (1994), presented at SC’93**
  - Basic point-to-point communication, collectives, datatypes, etc

- **MPI-2 (1997)**
  - Added parallel I/O, Remote Memory Access (one-sided operations), dynamic processes, thread support, C++ bindings, ...

- ---- Unchanged for 10 years ----

- **MPI-2.1 (2008)**
  - Minor clarifications and bug fixes to MPI-2

- **MPI-2.2 (2009)**
  - Small updates and additions to MPI 2.1

- **MPI-3.0 (2012)**
  - Major new features and additions to MPI

- **MPI-3.1 (2015)**
  - Small updates to MPI 3.0
Important considerations while using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs
Basic MPI Communication

OR

MPI_Recv

MPI_Send

OR

MPI_Irecv

MPI_Isend

MPI_Wait

MPI_Wait
Web Pointers

- MPI Standard: [http://www.mpi-forum.org/docs/docs.html](http://www.mpi-forum.org/docs/docs.html)

- MPI implementations:
  - MPICH: [http://www.mpich.org](http://www.mpich.org)
  - MVAPICH: [http://mvapich.cse.ohio-state.edu/](http://mvapich.cse.ohio-state.edu/)
  - Open MPI: [http://www.open-mpi.org/](http://www.open-mpi.org/)
  - IBM MPI, Cray MPI, HP MPI, TH MPI, ...

- Several MPI tutorials can be found on the web
Tutorial Books on MPI  (Released November 2014)

Using MPI
Portable Parallel Programming with the Message-Passing Interface
third edition

William Gropp
Ewing Lusk
Anthony Skjellum

Basic MPI

Using Advanced MPI
Modern Features of the Message-Passing Interface

William Gropp
Torsten Hoefler
Rajeev Thakur
Ewing Lusk

Advanced MPI, including MPI-2 and MPI-3
Understanding MPI Performance on Modern Processors

- MPI was developed when a single processor required multiple chips and most processors and nodes had a single core.
- Building effective, scalable applications requires having a model of how the system executes, how it performs, and what operations it can perform
  - This is (roughly) the execution model for the system, along with a performance model
- For decades, a simple model worked for designing and understanding MPI programs
  - Programs communicate either with point-to-point communication (send/recv), with a performance model of $T = s + r n$, where $s$ is latency (startup) and $r$ is inverse bandwidth (rate), or collective communication
- But today, processors are multi-core and many nodes are multi-chip.
  - How does that change how we think about performance and MPI?
SMP Nodes: One Model
Classic Performance Model

- $s + r \cdot n$
  - Sometimes called the “postal model”
- Model combines overhead and network latency ($s$) and a single communication rate $1/r$ for $n$ bytes of data
- Good fit to machines when it was introduced
- But does it match modern SMP-based machines?
  - Let’s look at the communication rate per process with processes communicating between two nodes
Rates Per MPI Process

- Ping-pong between 2 nodes using 1-16 cores on each node
- Top is BG/Q, bottom Cray XE6
- “Classic” model predicts a single curve – rates independent of the number of communicating processes
Why this Behavior?

- The $T = s + r n$ model predicts the *same* performance independent of the number of communicating processes
  - What is going on?
  - How should we model the time for communication?
SMP Nodes: One Model
Modeling the Communication

- Each link can support a rate $r_L$ of data
- Data is pipelined (LogP model)
  - Store and forward analysis is different
- Overhead is completely parallel
  - $k$ processes sending one short message each takes the same time as one process sending one short message
A Slightly Better Model

- Assume that the sustained communication rate is limited by
  - The maximum rate along any shared link
    - The link between NICs
  - The aggregate rate along parallel links
    - Each of the “links” from an MPI process to/from the NIC

![Aggregate Bandwidth Diagram](image-url)
A Slightly Better Model

- For $k$ processes sending messages, the sustained rate is
  - $\min(R_{\text{NIC-NIC}}, k \ R_{\text{CORE-NIC}})$

- Thus
  - $T = s + k \frac{n}{\min(R_{\text{NIC-NIC}}, k \ R_{\text{CORE-NIC}})}$

- Note if $R_{\text{NIC-NIC}}$ is very large (very fast network), this reduces to
  - $T = s + k \frac{n}{k \ R_{\text{CORE-NIC}}} = s + \frac{n}{R_{\text{CORE-NIC}}}$
Two Examples

- Two simplified examples:
  - Blue Gene/Q
  - Cray XE6

  - Note differences:
    - BG/Q: Multiple paths into the network
    - Cray XE6: Single path to NIC (shared by 2 nodes)
    - Multiple processes on a node sending can exceed the available bandwidth of the single path
The Test

- Nodecomm discovers the underlying physical topology
- Performs point-to-point communication (ping-pong) using 1 to # cores per node to another node (or another chip if a node has multiple chips)
- Outputs communication time for 1 to # cores along a single channel
  - Note that hardware may route some communication along a longer path to avoid contention.
- The following results use the code available soon at
  - https://bitbucket.org/william_gropp/baseenv
How Well Does this Model Work?

- Tested on a wide range of systems:
  - Cray XE6 with Gemini network
  - IBM BG/Q
  - Cluster with InfiniBand
  - Cluster with another network

- Results in
  - Modeling MPI Communication Performance on SMP Nodes: Is it Time to Retire the Ping Pong Test
    - W Gropp, L Olson, P Samfass
    - Proceedings of EuroMPI 16
    - https://doi.org/10.1145/2966884.2966919

- Cray XE6 results follow
Cray: Measured Data

The graph shows the aggregate effective bandwidth [bytes/second] as a function of message length [bytes]. The x-axis represents the message length in bytes, ranging from $10^0$ to $10^6$, while the y-axis represents the aggregate effective bandwidth in bytes/second, ranging from $10^6$ to $10^{10}$. The data is represented by different colored lines and markers for varying numbers of pair(s) as indicated in the legend.
Cray: 3 parameter (new) model

![Graph showing aggregate effective bandwidth vs. message length for different numbers of pairs. The x-axis represents message length in bytes, ranging from $10^0$ to $10^6$, and the y-axis represents aggregate effective bandwidth in bytes/second, ranging from $10^6$ to $10^{10}$. The graph includes lines for 1 pair(s) up to 16 pair(s), with each line representing a different number of pairs. The red horizontal lines at the top indicate the maximum bandwidth values for each number of pairs.](image)
Cray: 2 parameter model
Both Cray XE6 and IBM BG/Q have inadequate bandwidth to support each core sending data along the same link

- But BG/Q has more independent links, so it is able to sustain a higher effective “halo exchange”

What about other systems?

- We see the same behavior on a wide variety of systems and networks
- Many-core systems are strongly affected...
  - Test on next slide from a simpler version of the test code available from [http://wgropp.cs.illinois.edu/mpimesh-0.3.tgz](http://wgropp.cs.illinois.edu/mpimesh-0.3.tgz) in the code mpingpong
Mpingpong results for Theta

Intranode Pingpong Performance

- Rate B/sec
- Message Size Doubles

Series: 1 to 60
Modeling Communication

- For $k$ processes sending messages concurrently from the same node, the correct (more precisely, a much better) time model is
  - $T = s + k \frac{n}{\min(R_{NIC-NIC}, k R_{CORE-NIC})}$
- Further terms improve this model, but this one is sufficient for many uses
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>Start Send</th>
<th>Start Send</th>
<th>Start Send</th>
<th>Start Send</th>
<th>Start Send</th>
<th>Start Send</th>
<th>SendRecv</th>
<th>Recv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>Send</td>
<td>Send</td>
<td>Send</td>
<td>Send</td>
<td>Send</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>Recv</td>
<td>Send Recv</td>
<td>Send Recv</td>
<td>Send Recv</td>
<td>Send Recv</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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
Timeline from IBM SP

- Note that process 1 finishes last, as predicted
Distribution of Sends

'SEND' state length distribution

(in seconds)

58 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_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_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(n_neighbors+i), ierr)

Enddo

Call MPI_Waitall(2*n_neighbors, 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

- Consider the use of nonblocking operations and MPI_Waitall to defer synchronization
  - Effectiveness depends on how data is moved my the MPI implementation
  - E.g., If large messages are moved by blocking RMA operations “under the covers,” the implementation can’t adapt to contention at the target processes, and you may see no benefit.
  - This is more likely with larger messages
Hotspot results for Theta

2-d Mesh Exchange Comparison

Time/iteration in Secs

Edge size in words

W-Isend/Irecv
W-Send/Irecv
NC-Isend/Irecv
NC-Send/Irecv
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.
vector
struct
indexed
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

- 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

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

- 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

MPI_Type_create_indexed_block(int count, int blocklength, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Like Create_indexed but blocklength is the same
  - blen=2
  - displs={0,5,9,13,18}
MPI_Type_create_hindexed

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

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

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

MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

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

<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><strong>(1,2)</strong></td>
<td>(2,2)</td>
<td><strong>(3,2)</strong></td>
<td></td>
</tr>
<tr>
<td>(0,3)</td>
<td><strong>(1,3)</strong>*</td>
<td>(2,3)</td>
<td><strong>(3,3)</strong></td>
<td></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_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

<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_
Datatype Performance in Practice

- Datatypes *can* provide performance benefits, particularly for certain regular patterns
  - However, many implementations do not optimize datatype operations
  - If performance is critical, you will need to test
    - Even manual packing/unpacking can be slow if not properly optimized by the compiler – make sure to check optimization reports or if the compiler doesn’t provide good reports, inspect the assembly code
- For parallel I/O, datatypes *do* provide large performance benefits in many cases
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: A
  - P1: A
  - P2: A
  - P3: A

- **Scatter**
  - P0: A B C D
  - P1: A
  - P2: B
  - P3: C

- **Gather**
  - P0: A
  - P1: B
  - P2: C
  - P3: D
More Collective Data Movement

**Allgather**

- **P0**: A
- **P1**: B
- **P2**: C
- **P3**: D

- **Allgather**
  - A
  - B
  - C
  - D

**Alltoall**

- **P0**: A0 A1 A2 A3
- **P1**: B0 B1 B2 B3
- **P2**: C0 C1 C2 C3
- **P3**: D0 D1 D2 D3

- **Alltoall**
  - A0 A1 A2 A3
  - B0 B1 B2 B3
  - C0 C1 C2 C3
  - D0 D1 D2 D3
Collective Computation

Reduce

Scan
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**, **MPI_NO_OP**
  - Replace and no operation (RMA)
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

Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI
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
Advanced Topics: 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
Two-sided Communication Example
One-sided Communication Example
Comparing One-sided and Two-sided Programming

**One-sided Programming**

- **Process 0**: `SEND(data)`
- **Process 1**: `RECV(data)`

Even the sending process is delayed.

**Two-sided Programming**

- **Process 0**: `PUT(data)`
- **Process 1**: `GET(data)`

Delay in process 1 does not affect process 0.
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 used by a process is, by default, only locally accessible
  - \(X = \text{malloc}(100);\)
- Once the memory is allocated, 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
Window creation models

- Four models exist
  - MPI_WIN_ALLOCATE
    - You want to create a buffer and directly make it remotely accessible
  - MPI_WIN_CREATE
    - You already have an allocated buffer that you would like to make remotely accessible
  - MPI_WIN_CREATE_DYNAMIC
    - You don’t have a buffer yet, but will have one in the future
    - You may want to dynamically add/remove buffers to/from the window
  - MPI_WIN_ALLOCATE_SHARED
    - You want multiple processes on the same node share a buffer
MPI_WIN_ALLOCATE

- 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
  - win - window (handle)
Example with MPI_WIN_ALLOCATE

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

    MPI_Init(&argc, &argv);

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

    /* Array ‘a’ is now accessible from all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

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

- 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)
  - win - window (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 */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* 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_Free_mem(a);
    MPI_Finalize(); return 0;
}
```
MPI_WIN_CREATE_DYNAMIC

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops
- Initially “empty”
  - Application can dynamically attach/detach memory to this window by calling MPI_Win_attach/detach
  - Application can access data on this window only after a memory region has been attached
- Window origin is MPI_BOTTOM
  - Displacements are segment addresses relative to MPI_BOTTOM
  - Must tell others the displacement after calling attach

MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
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 = (int *) 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 accessible from all processes */

    /* undeclare remotely accessible memory */
    MPI_Win_detach(win, a);  free(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_PUT
  - MPI_GET
  - MPI_ACCUMULATE (atomic)
  - MPI_GET_ACCUMULATE (atomic)
  - MPICOMPARE_AND_SWAP (atomic)
  - MPI_FETCH_AND_OP (atomic)
Data movement: *Put*

- Move data **from** origin, **to** target
- Separate data description triples for **origin** and **target**

```
MPI_Put(const void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Win win)
```
Data movement: *Get*

- Move data to origin, from target
- Separate data description triples for **origin** and **target**

```c
MPI_Get(void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Win win)
```
Atomic Data Aggregation: Accumulate

- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, ...
  - Predefined ops only, no user-defined operations

- Different data layouts between target/origin OK
  - Basic type elements must match

- Op = MPI_REPLACE
  - Implements \( f(a,b)=b \)
  - Atomic PUT
Atomic Data Aggregation: Get Accumulate

**MPI_Get_accumulate**

```c
MPI_Get_accumulate(const void *origin_addr,
                   int origin_count, MPI_Datatype origin_dtype,
                   void *result_addr, int result_count,
                   MPI_Datatype result_dtype, int target_rank,
                   MPI_Aint target_disp, int target_count,
                   MPI_Datatype target_dtype, MPI_Op op, MPI_Win win)
```

- Atomic read-modify-write
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, ...
  - Predefined ops only
- Result stored in target buffer
- Original data stored in result buf
- Different data layouts between target/origin OK
  - Basic type elements must match
- Atomic get with MPI_NO_OP
- Atomic swap with MPI_REPLACE

Remote Accessible Memory

Private Memory
Atomic Data Aggregation: CAS and FOP

- **FOP**: Simpler version of MPI_Get_accumulate
  - All buffers share a single predefined datatype
  - No count argument (it’s always 1)
  - Simpler interface allows hardware optimization

- **CAS**: Atomic swap if target value is equal to compare value

```c
MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
                 MPI_Datatype dtype, int target_rank,
                 MPI_Aint target_disp, MPI_Op op, MPI_Win win)
```

```c
MPI_Compare_and_swap(const void *origin_addr,
                      const void *compare_addr, void *result_addr,
                      MPI_Datatype dtype, int target_rank,
                      MPI_Aint target_disp, MPI_Win win)
```
Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent Put/Accumulate undefined
  - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which the occurred
  - Atomic put: Accumulate with op = MPI_REPLACE
  - Atomic get: Get_accumulate with op = MPI_NO_OP

- Accumulate operations from a given process are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW
Examples with operation ordering

1. Concurrent Puts: **undefined**

2. Concurrent Get and Put/Accumulates: **undefined**

3. Concurrent Accumulate operations to the same location: **ordering is guaranteed**
RMA Synchronization Models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics

- Three synchronization models provided by MPI:
  - Fence (active target)
  - Post-start-complete-wait (generalized active target)
  - Lock/Unlock (passive target)

- Data accesses occur within “epochs”
  - Access epochs: contain a set of operations issued by an origin process
  - Exposure epochs: enable remote processes to update a target’s window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs
    - E.g., starting, ending, and synchronizing epochs
Fence: Active Target Synchronization

- Collective synchronization model
- Starts and ends access and exposure epochs on all processes in the window
- All processes in group of “win” do an MPI_WIN_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
- All operations complete at the second fence synchronization
Implementing Stencil Computation with RMA Fence

Origin buffers

Target buffers

PUT

PUT

PUT

RMA window
Code Example

- Code example from the examples set
PSCW: Generalized Active Target Synchronization

- Like FENCE, but origin and target specify who they communicate with

- 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_complete

- All synchronization operations may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets
Passive mode: One-sided, *asynchronous* communication
- Target does **not** participate in communication operation

Shared memory-like model
Passive Target Synchronization

- **Lock/Unlock**: Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed (affects threads)

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

- **Flush**: Remotely complete RMA operations to the target process
  - After completion, data can be read by target process or a different process

- **Flush_local**: Locally complete RMA operations to the target process

**MPI**

- **MPI_Win_lock**(int locktype, int rank, int assert, MPI_Win win)
- **MPI_Win_unlock**(int rank, MPI_Win win)
- **MPI_Win_flush/flush_local**(int rank, MPI_Win win)
Advanced Passive Target Synchronization

- **Lock_all**: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: lock_all, put/get, flush, ..., unlock_all
- **Flush_all** – remotely complete RMA operations to all processes
- **Flush_local_all** – locally complete RMA operations to all processes
NWChem [1]

- High performance computational chemistry application suite
- Quantum level simulation of molecular systems
  - Very expensive in computation and data movement, so is used for small systems
  - Larger systems use molecular level simulations
- Composed of many simulation capabilities
  - Molecular Electronic Structure
  - Quantum Mechanics/Molecular Mechanics
  - Pseudo potential Plane-Wave Electronic Structure
  - Molecular Dynamics
- Very large code base
  - 4M LOC; Total investment of ~200M $ to date

NWChem Communication Runtime

Applications

Global Arrays [2]

ARMCI: Communication interface for RMA [3]

Global Address Space

Physically distributed to different processes

Hidden from user

Irregularly access large amount of remote memory regions

Get-Compute-Update

- Typical Get-Compute-Update mode in GA programming

```
Perform DGEMM in local buffer

Mock figure showing 2D DGEMM with block-sparse computations. In reality, NWChem uses 6D tensors.

Pseudo code

for i in I blocks:
  for j in J blocks:
    for k in K blocks:
      GET block a from A
      GET block b from B
      \[ c += a \times b \] /*computing*/
    end do
  ACC block c to C
NXTASK
end do
end do
```
Which synchronization mode should I use, when?

- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc...
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)

- Active mode: bulk synchronization
  - E.g. ghost cell exchange

- Passive mode: asynchronous data movement
  - Useful when dataset is large, requiring memory of multiple nodes
  - Also, when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays

- Passive target locking mode
  - Lock/unlock – Useful when exclusive epochs are needed
  - Lock_all/unlock_all – Useful when only shared epochs are needed
MPI RMA Memory Model

- MPI-3 provides two memory models: separate and unified
- MPI-2: Separate Model
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don’t provide hardware coherence
- MPI-3: New Unified Model
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics
    - E.g. allows concurrent local/remote access
  - Provides access to full performance potential of hardware
MPI RMA Memory Model (separate windows)

- Very portable, compatible with non-coherent memory systems
- Limits concurrent accesses to enable software coherence
MPI RMA Memory Model (unified windows)

- Allows concurrent local/remote accesses
- Concurrent, conflicting operations are allowed (not invalid)
  - Outcome is not defined by MPI (defined by the hardware)
- Can enable better performance by reducing synchronization
### MPI RMA Operation Compatibility (Separate)

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Store</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Get</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Put</td>
<td>NOVL</td>
<td>X</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Acc</td>
<td>NOVL</td>
<td>X</td>
<td>NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
</tr>
</tbody>
</table>

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

- **OVL** – Overlapping operations permitted
- **NOVL** – Nonoverlapping operations permitted
- **X** – Combining these operations is OK, but data might be garbage
### MPI RMA Operation Compatibility (Unified)

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Store</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Get</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Put</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Acc</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
</tr>
</tbody>
</table>

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

OVL  – Overlapping operations permitted
NOVL – Nonoverlapping operations permitted
Advanced Topics: Hybrid Programming with Threads, Shared Memory, and Accelerators
Hybrid MPI + X : Most Popular Forms

**MPI + X**

- **MPI + 0**
- **MPI + Threads**
- **MPI + Shared Memory**
- **MPI + ACC**
MPI + Threads
Why Hybrid MPI+X? Towards Strong Scaling (1/2)

- Strong scaling applications is increasing in importance
  - Hardware limitations: not all resources scale at the same rate as cores (e.g., memory capacity, network resources)
  - Desire to solve the same problem faster on a bigger machine
    - Nek5000, HACC, LAMMPS

- Strong scaling pure MPI applications is getting harder
  - On-node communication is costly compared to load/stores
  - $O(P^x)$ communication patterns (e.g., All-to-all) costly

- MPI+X benefits ($X=\{\text{threads, MPI shared-memory, etc.}\}$)
  - Less memory hungry (MPI runtime consumption, $O(P)$ data structures, etc.)
  - Load/stores to access memory instead of message passing
  - $P$ is reduced by constant $C$ (#cores/process) for $O(P^x)$ communication patterns
Why Hybrid MPI+X? Towards Strong Scaling (2/2)

- **Example 1: Quantum Monte Carlo Simulation (QCMPACK)**
  - Size of the physical system to simulate is bound by memory capacity [1]
  - Memory space dominated by large interpolation tables (typically several Giga Bytes of storage)
  - Threads are used to share those tables
  - Memory for communication buffers must be kept low to be allow simulation of larger and highly detailed simulations.

- **Example 2: the Nek5000 team is working at the strong scaling limit**

**Nek5000**

---

MPI + Threads: How To? (1/2)

Multi- or Many-core Nodes

MPI only

MPI + Threads

Advanced MPI, Argonne (06/26/2017)
MPI + Threads: How To? (2/2)

- **MPI** describes parallelism between *processes* (with separate address spaces)

- *Thread* parallelism provides a shared-memory model within a process

- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.
**MPI + Threads: How To? (2/2)**

Interoperation or thread levels:

- **MPI_THREAD_SINGLE**
  - No additional threads

- **MPI_THREAD_FUNNELED**
  - Master thread communication only

- **MPI_THREAD_SERIALIZED**
  - Threaded communication serialized

- **MPI_THREAD_MULTIPLE**
  - No restrictions

- **Restriction**
  - Low Thread-Safety Costs

- **Flexibility**
  - High Thread-Safety Costs
MPI’s Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZE
- MPI defines an alternative to MPI_Init
  - MPI_Init_thread(requested, provided): Application specifies level it needs; MPI implementation returns level it supports
**MPI_THREAD_SINGLE**

- There are no additional user threads in the system
  - E.g., there are no OpenMP parallel regions

```c
int buf[100];
int main(int argc, char ** argv)
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```
MPI_THREAD_FUNNELED

- All MPI calls are made by the **master** thread
  - Outside the OpenMP parallel regions
  - In OpenMP master regions

```c
int buf[100];
int main(int argc, char ** argv)
{
    int provided;

    MPI_Init_thread(&argc, &argv,
        MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD,1);

    for (i = 0; i < 100; i++)
        pthread_create(...,func,(void*)i);
    for (i = 0; i < 100; i++)
        pthread_join();

    /* Do MPI stuff */

    MPI_Finalize();
    return 0;
}
```

```c
void* func(void* arg) {
    int i = (int)arg;
    compute(buf[i]);
}
```
**MPI_THREAD_SERIALIZE**

- **Only one** thread can make MPI calls at a time
  - Protected by OpenMP critical regions

```c
int buf[100];
int main(int argc, char **argv)
{
    int provided;
    pthread_mutex_t mutex;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZE, &provided);
    if (provided < MPI_THREAD_SERIALIZE)
        MPI_Abort(MPI_COMM_WORLD, 1);

    for (i = 0; i < 100; i++)
        pthread_create(..., func, (void*)i);
    for (i = 0; i < 100; i++)
        pthread_join();

    MPI_Finalize();
    return 0;
}
```

```c
void* func(void* arg) {
    int i = (int)arg;
    compute(buf[i]);
    pthread_mutex_lock(&mutex);
    /* Do MPI stuff */
    pthread_mutex_unlock(&mutex);
}
```
MPI_THREAD_MULTIPLE

- **Any** thread can make MPI calls any time (restrictions apply)

```c
int buf[100];
int main(int argc, char ** argv)
{
    int provided;

    MPI_Init_thread(&argc, &argv, 
                    MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_SERIALIZED)
        MPI_Abort(MPI_COMM_WORLD,1);

    for (i = 0; i < 100; i++)
        pthread_create(...,func,(void*)i);

    MPI_Finalize();
    return 0;
}

void* func(void* arg) {
    int i = (int)arg;
    compute(buf[i]);

    /* Do MPI stuff */
}
```
Threads and MPI

An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe

A fully thread-safe implementation will support MPI_THREAD_MULTIPLE

A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported

- MPI Standard mandates MPI_THREAD_SINGLE for MPI_Init

A threaded MPI program that does not call MPI_Init_thread is an incorrect program (common user error we see)
MPI Semantics and MPI_THREAD_MULTIPLE

- **Ordering:** When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
  - Ordering is maintained within each thread
  - User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
    - E.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator
  - It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
    - E.g., accessing an info object from one thread and freeing it from another thread

- **Progress:** Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
<td><strong>Thread 0</strong></td>
</tr>
<tr>
<td>MPI_Bcast(comm)</td>
<td>MPI_Bcast(comm)</td>
</tr>
<tr>
<td><strong>Thread 1</strong></td>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>MPI_Barrier(comm)</td>
<td>MPI_Barrier(comm)</td>
</tr>
</tbody>
</table>
Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives

- P0 and P1 can have different orderings of Bcast and Barrier
- Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI
Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Object Management

- The user has to make sure that one thread is not using an object while another thread is freeing it
  - This is essentially an ordering issue; the object might get freed before it is used
Blocking Calls in MPI_THREAD_MULTIPLE: Correct Example

- An implementation must ensure that the above example never deadlocks for any ordering of thread execution.
- That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress.

**Process 0**

- Thread 1: `MPI_Recv(src=1)`
- Thread 2: `MPI_Send(dst=1)`

**Process 1**

- MPI_Recv(src=0)
- MPI_Send(dst=0)
The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE
- They probably support MPI_THREAD_FUNNELED even if they don’t admit it.
  - Does require thread-safety for some system routines (e.g. malloc)
  - On most systems -pthread will guarantee it (OpenMP implies -pthread)
- Many (but not all) implementations support THREAD_MULTIPLE
  - Hard to implement efficiently though (thread synchronization issues)
- Bulk-synchronous OpenMP programs (loops parallelized with OpenMP, communication between loops) only need FUNNELED
  - So don’t need “thread-safe” MPI for many hybrid programs
  - But watch out for Amdahl’s Law!
Hybrid Programming: Correctness Requirements

- Hybrid programming with MPI+threads does not do much to reduce the complexity of thread programming
  - Your application still has to be a correct multi-threaded application
  - On top of that, you also need to make sure you are correctly following MPI semantics

- Many commercial debuggers offer support for debugging hybrid MPI+threads applications (mostly for MPI+Pthreads and MPI+OpenMP)
An Example we encountered

- We received a bug report about a very simple multithreaded MPI program that hangs
- Run with 2 processes
- Each process has 2 threads
- Both threads communicate with threads on the other process as shown in the next slide
- We spent several hours trying to debug MPICH before discovering that the bug is actually in the user’s program 😞
2 Processes, 2 Threads, Each Thread Executes this Code

for (j = 0; j < 2; j++) {
    if (rank == 1) {
        for (i = 0; i < 2; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        for (i = 0; i < 2; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
    }
    else /* rank == 0 */
    
        for (i = 0; i < 2; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
        for (i = 0; i < 2; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
}

}
Intended Ordering of Operations

Every send matches a receive on the other rank
Because the MPI operations can be issued in an arbitrary order across threads, all threads could block in a RECV call
MPI+OpenMP correctness semantics

- For OpenMP threads, the MPI+OpenMP correctness semantics are similar to that of MPI+threads
  - Caution: OpenMP iterations need to be carefully mapped to which thread executes them (some schedules in OpenMP make this harder)

- For OpenMP tasks, the general model to use is that an OpenMP thread can execute one or more OpenMP tasks
  - An MPI blocking call should be assumed to block the entire OpenMP thread, so other tasks might not get executed
OpenMP threads: MPI blocking Calls (1/2)

```c
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
        if (i % 2 == 0)
            MPI_Send(.., to_myself, ..);
        else
            MPI_Recv(.., from_myself, ..);
    }

    MPI_Finalize();

    return 0;
}
```

*Iteration to OpenMP thread mapping needs to explicitly be handled by the user; otherwise, OpenMP threads might all issue the same operation and deadlock*
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

    #pragma omp parallel
    {
        assert(omp_get_num_threads() > 1)
        #pragma omp for schedule(static, 1)
        for (i = 0; i < 100; i++) {
            if (i % 2 == 0)
                MPI_Send(.., to_myself, ..);
            else
                MPI_Recv(.., from_myself, ..);
        }
    }

    MPI_Finalize();

    return 0;
}

Either explicit/careful mapping of iterations to threads, or using nonblocking versions of send/recv would solve this problem
OpenMP tasks: MPI blocking Calls (1/5)

```c
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 100; i++) {
            #pragma omp task
            {
                if (i % 2 == 0)
                    MPI_Send(.., to_myself, ..);
                else
                    MPI_Recv(.., from_myself, ..);
            }
        }
    }

    MPI_Finalize();
    return 0;
}
```

This can lead to deadlocks. No ordering or progress guarantees in OpenMP task scheduling should be assumed; a blocked task blocks it’s thread and tasks can be executed in any order.
OpenMP tasks: MPI blocking Calls (2/5)

```c
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

    #pragma omp parallel
    {
        #pragma omp taskloop
        for (i = 0; i < 100; i++) {
            if (i % 2 == 0)
                MPI_Send(.., to_myself, ..);
            else
                MPI_Recv(.., from_myself, ..)
        }
    }

    MPI_Finalize();
    return 0;
}
```

Same problem as before.
OpenMP tasks: MPI blocking Calls (3/5)

```c
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

    #pragma omp parallel
    {
        #pragma omp taskloop
        for (i = 0; i < 100; i++) {
            MPI_Request req;
            if (i % 2 == 0)
                MPI_Isend(..., to_myself, ..., &req);
            else
                MPI_Irecv(..., from_myself, ..., &req);
            MPI_Wait(&req);
        }
    }
    MPI_Finalize();
    return 0;
}
```

*Using nonblocking operations but with MPI_Wait inside the task region does not solve the problem*
OpenMP tasks: MPI blocking Calls (4/5)

```c
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

    #pragma omp parallel
    {
        #pragma omp taskloop
        for (i = 0; i < 100; i++) {
            MPI_Request req; int done = 0;
            if (i % 2 == 0)
                MPI_Isend(.., to_myself, .., &req);
            else
                MPI_Irecv(.., from_myself, .., &req);
            While (!done) {
                #pragma omp taskyield
                MPI_Test(&req, &done, ..);
            }
        }
    }

    MPI_Finalize();
    return 0;
}
```

Still incorrect; taskyield does not guarantee a task switch
OpenMP tasks: MPI blocking Calls (5/5)

```c
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    MPI_Request req[100];

    #pragma omp parallel
    {
        #pragma omp taskloop
        for (i = 0; i < 100; i++) {
            if (i % 2 == 0)
                MPI_Isend(., to_myself, ., &req[i]);
            else
                MPI_Irecv(., from_myself, ., &req[i]);
        }
    }
    MPI_Waitall(100, req, .);
    MPI_Finalize();
    return 0;
}
```

Correct example. Each task is nonblocking.
Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with RMA

```c
int main(int argc, char ** argv)
{
    /* Initialize MPI and RMA window */

    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
        target = rand();
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target, 0, win);
        MPI_Put(..., win);
        MPI_Win_unlock(target, win);
    }

    /* Free MPI and RMA window */

    return 0;
}
```

Different threads can lock the same process causing multiple locks to the same target before the first lock is unlocked.
Implementing Stencil Computation using MPI_THREAD_FUNNELED
Implementing Stencil Computation using MPI_THREAD_MULTIPLE
MPI + Shared-Memory
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads
Creating Shared Memory Regions in MPI

MPI_COMM_WORLD

MPI_Comm_split_type (MPI_COMM_TYPE_SHARED)

Shared memory communicator

MPI_Win_allocate_shared

Shared memory window

Shared memory communicator

Shared memory window

Shared memory communicator

Shared memory window
Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
- E.g., x[100] = 10

All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations

Can be very useful when processes want to use threads only to get access to all of the memory on the node
- You can create a shared memory window and put your shared data

**Traditional RMA windows**

**Shared memory windows**
**MPI_COMM_SPLIT_TYPE**

Create a communicator where processes “share a property”
- Properties are defined by the “split_type”

**Arguments:**
- comm - input communicator (handle)
- Split_type - property of the partitioning (integer)
- Key - Rank assignment ordering (nonnegative integer)
- info - info argument (handle)
- newcomm - output communicator (handle)
MPI_WIN_ALLOCATE_SHARED

- Create a remotely accessible memory region in an RMA window
  - Data exposed in a window can be accessed with RMA ops or load/store

- 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
  - win - window (handle)
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Win_sync(win);

    /* use shared memory */

    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
Memory allocation and placement

- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed (e.g., which physical memory it will be pinned to)
  - Implementations can choose their own strategies, though it is expected that an implementation will try to place shared memory allocated by a process “close to it”
- The total allocated shared memory on a communicator is contiguous by default
  - Users can pass an info hint called “noncontig” that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement
Example Computation: Stencil

Message passing model requires ghost-cells to be explicitly communicated to neighbor processes.

In the shared-memory model, there is no communication. Neighbors directly access your data.
Which Hybrid Programming Method to Adopt?

- It depends on the application, target machine, and MPI implementation
- When should I use process shared memory?
  - The only resource that needs sharing is memory
  - Few allocated objects need sharing (easy to place them in a public shared region)
- When should I use threads?
  - More than memory resources need sharing (e.g., TLB)
  - Many application objects require sharing
  - Application computation structure can be easily parallelized with high-level OpenMP loops
Example: Quantum Monte Carlo

- Memory capacity bound with MPI-only
- Hybrid approaches
  - MPI + threads (e.g. X = OpenMP, Pthreads)
  - MPI + shared-memory (X = MPI)
- Can use direct load/store operations instead of message passing

**MPI + Shared-Memory (MPI 3.0)**
- Everything private by default
- Expose shared data explicitly

**MPI + Threads**
- Share everything by default
- Privatize data when necessary
MPI + Accelerators
Accelerators in Parallel Computing

- General purpose, highly parallel processors
  - High FLOPs/Watt and FLOPs/$
  - Unit of execution *Kernel*
  - Separate memory subsystem
  - Programming Models: OpenAcc, CUDA, OpenCL, ...

- Clusters with accelerators are becoming common

- New programmability and performance challenges for programming models and runtime systems
FAQ: How to move data between GPUs with MPI?
**MPI + Accelerator Programming Examples (2/2)**

```c
double *dev_buf, *host_buf;
cudaMalloc(&dev_buf, size);
cudaMallocHost(&host_buf, size);

if (my_rank == sender) {
    computation_on_GPU(dev_buf);
    cudaMemcpy(host_buf, dev_buf, size, ...);
    MPI_Isend(host_buf, size, ...);
} else {
    MPI_Irecv(host_buf, size, ...);
    cudaMemcpy(dev_buf, host_buf, size, ...);
    computation_on_GPU(dev_buf);
}
```

```c
double *buf;
buf = (double*)malloc(size * sizeof(double));
#pragma acc enter data create(buf[0:size])

if (my_rank == sender) {
    computation_on_GPU(buf);
    #pragma acc update host (buf[0:size])
    MPI_Isend(buf, size, ...);
} else {
    MPI_Irecv(buf, size, ...);
    #pragma acc update device (buf[0:size])
    computation_on_GPU(buf);
}
```
**MPI with Old GPU Technologies**

- MPI only sees host memory
- User has to ensure data copies on host and device are updated consistently
- Several memory copy operations are required
- No overlapping between device-host memory transfers and network communication
- **No MPI optimization opportunities**

```
computation_on_GPU(dev_buf);
cudaMemcpy(host_buf, dev_buf, size, ...);
MPI_Isend(host_buf, size, ...);

MPI_Irecv(host_buf, size, ...);
cudaMemcpy(dev_buf, host_buf, size, ...);
computation_on_GPU(dev_buf);
```

```
computation_on_GPU(buf);
#pragma acc update host (buf[0:size])
MPI_Isend(buf, size, ...);

MPI_Irecv(buf, size, ...);
#pragma acc update device (buf[0:size])
computation_on_GPU(buf);
```
MPI with Unified Virtual Addressing (UVA)

- The same virtual address space for all processors, host or device (e.g., CUDA >= 4)
- User can pass device pointer to MPI
- MPI implementation needs to query for the owner (host or device) of the data
- If data on device, the MPI implementation can optimize as follows:
  - Reduce the number of memory copies and DMA operations
  - Do better overlapping of data transfers

```c
computation_on_GPU(dev_buf);
MPI_Isend(dev_buf, size, ...);

MPI_Irecv(dev_buf, size, ...);
computation_on_GPU(dev_buf);
```

```c
computation_on_GPU(buf);
#pragma acc host_data use_device (buf)
MPI_Isend(buf, size, ...);

#pragma acc host_data use_device (buf)
MPI_Irecv(buf, size, ...);
computation_on_GPU(buf);
```
MPI with UVA + GPUDirect

- The hardware supports direct GPU-to-GPU data transfers within or across nodes
- MPI implementations may use the following optimizations to transfer data between GPUs
  - Can use directly GPU memory for RDMA communication
  - Peer-to-peer data transfers when GPUs are on the same node

```c
computation_on_GPU(dev_buf);
MPI_Isend(dev_buf, size, ...);

MPI_Irecv(dev_buf, size, ...);
computation_on_GPU(dev_buf);
```

```c
computation_on_GPU(buf);
#pragma acc host_data use_device (buf)
MPI_Isend(buf, size, ...);

#pragma acc host_data use_device (buf)
MPI_Irecv(buf, size, ...);
computation_on_GPU(buf);
```
Topology Mapping
Topology Mapping Basics

- First type: Allocation mapping (when job is submitted)
  - 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 maximization”
  - Also problematic to specify communication pattern upfront, not always possible (or static)
Topology Mapping Basics contd.

- **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

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

- **Cartesian Topologies**
  - MPI_Dims_create is required to provide a “square” decomposition
    - May not match underlying physical network
    - Even if it did, hard to define unless physical network is mesh or torus
  - MPI_Cart_create is supposed to provide a “good” remapping (if requested)
    - But implementations are poor and may just return the original mapping

- **Graph Topologies**
  - The general process mapping problem is very hard
  - Most (all?) MPI implementations are poor
  - Some research work has developed tools to create better mappings
    - You can use them with MPI_Comm_dup to create a “well ordered” communicator

- **Neighbor collectives**
  - MPI 3 introduced these; permit collective communication with just the neighbors as defined by the MPI process topology
  - Offers opportunities for the MPI implementation to optimize; not realized yet
Hotspot results for Theta

- Communicator from MPI_Cart_create has same order as MPI_COMM_WORLD
- For 2-d mesh exchange with 512 processes and 64 processes/node, MPI_Cart_create has an average of 28 target off-node processes.
- The “node-cart” communicator has an average of 20 target off-node processes
  - Communication time is 26% faster with the hand-optimized communicator
**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 in dims
MPIDims_create Example

```c
int p;
int dims[3] = {0,0,0};
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
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 topocomm;
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!
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

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

- 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
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