NON-BLOCKING COMMUNICATION
Non-blocking communication

Non-blocking sends and receives
- `MPI_Isend` & `MPI_Irecv`
  - returns immediately and sends/receives in background

Enables some computing concurrently with communication

Avoids many common dead-lock situations

Also non-blocking collective operations in MPI 3.0
Non-blocking communication

- Have to finalize send/receive operations
  - MPI_Wait, MPI_Waitall,…
    - Waits for the communication started with MPI_Isend or MPI_Irecv to finish (blocking)
  - MPI_Test,…
    - Tests if the communication has finished (non-blocking)
- You can mix non-blocking and blocking p2p routines
  - e.g., receive MPI_Isend with MPI_Recv
Typical usage pattern

MPI_Irecv(ghost_data)
MPI_Isend(border_data)
Compute(ghost_independent_data)
MPI_Waitall
Compute(border_data)
Non-blocking send

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

Parameters

Similar to MPI_Send but has an additional request parameter

buf send buffer that must not be written to until one has checked that the operation is over

request a handle that is used when checking if the operation has finished (integer in Fortran, MPI_Request in C)
Order of sends

- Sends done in the specified order even for non-blocking routines
- Beware of badly ordered sends!
Non-blocking receive

MPI_Irecv(buf, count, datatype, source, tag, comm, request)

parameters similar to MPI_Recv but has no status parameter

buf receive buffer guaranteed to contain the data only after one has checked that the operation is over

request a handle that is used when checking if the operation has finished
Wait for non-blocking operation

MPI_Wait(request, status)

Parameters
request  handle of the non-blocking communication
status   status of the completed communication, see MPI_Recv

A call to MPI_WAIT returns when the operation identified by request is complete
Wait for non-blocking operations

MPI_Waitall(count, requests, status)

Parameters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>number of requests</td>
</tr>
<tr>
<td>requests</td>
<td>array of requests</td>
</tr>
<tr>
<td>status</td>
<td>array of statuses for the operations that are waited for</td>
</tr>
</tbody>
</table>

A call to MPI_Waitall returns when all operations identified by the array of requests are complete
Additional completion operations

other useful routines:
- MPI_Waitany
- MPI_Waitsome
- MPI_Test
- MPI_Testall
- MPI_Testany
- MPI_Testsome
- MPI_Probe
Wait for non-blocking operations

MPI_Waitany(count, requests, index, status)

Parameters

- **count**: number of requests
- **requests**: array of requests
- **index**: index of request that completed
- **status**: status for the completed operations

A call to MPI_Waitany returns when one operation identified by the array of requests is complete
Wait for non-blocking operations

MPI_Waitsome(count, requests, done, index, status)

Parameters

- **count**: number of requests
- **requests**: array of requests
- **done**: number of completed requests
- **index**: array of indexes of completed requests
- **status**: array of statuses of completed requests

A call to MPI_Waitsome returns when one or more operation identified by the array of requests is complete.
Non-blocking test for non-blocking operations

MPI_Test(request, flag, status)

Parameters
- request: request
- flag: True if operation has completed
- status: status for the completed operations

A call to MPI_Test is non-blocking. It allows one to schedule alternative activities while periodically checking for completion.
Example: Non-blocking broadcasting

\textbf{MPI\_Ibcast}(*buffer*, *count*, *datatype*, *root*, *comm*, *request*)

- **buffer**: data to be distributed
- **count**: number of entries in buffer
- **datatype**: data type of buffer
- **root**: rank of broadcast root
- **comm**: communicator
- **request**: a handle that is used when checking if the operation has finished
Typical usage pattern

MPI_Ibcast(data,...,request)

! Do any kind of work not involving data

! ...

MPI_Wait(request)
Summary

- Non-blocking communication is usually the smarter way to do point-to-point communication in MPI.

- Non-blocking communication realization
  - `MPI_Isend`
  - `MPI_Irecv`
  - `MPI_Wait(all)`

- MPI-3 contains also non-blocking collectives.
USER-DEFINED DATATYPES
MPI datatypes

MPI datatypes are used for communication purposes
- Datatype tells MPI where to take the data when sending or where to put data when receiving

Elementary datatypes (MPI_INT, MPI_REAL, ...)
- Different types in Fortran and C, correspond to languages basic types
- Enable communication using contiguous memory sequence of identical elements (e.g. vector or matrix)
Sending a matrix row (Fortran)

Row of a matrix is not contiguous in memory in Fortran

Several options for sending a row:

- Use several send commands for each element of a row
- Copy data to temporary buffer and send that with one send command
- Create a matching datatype and send all data with one send command

![Logical layout](image)

![Physical layout](image)
User-defined datatypes

- Use elementary datatypes as building blocks
- Enable communication of
  - Non-contiguous data with a single MPI call, e.g. rows or columns of a matrix
  - Heterogeneous data (structs in C, types in Fortran)
- Provide higher level of programming & efficiency
  - Code is more compact and maintainable
  - Communication of non-contiguous data is more efficient
- Needed for getting the most out of MPI I/O
User-defined datatypes

User-defined datatypes can be used both in point-to-point communication and collective communication.
The datatype instructs where to take the data when sending or where to put data when receiving.
- Non-contiguous data in sending process can be received as contiguous or vice versa.
Using user-defined datatypes

- A new datatype is created from existing ones with a datatype constructor
  - Several routines for different special cases

- A new datatype must be committed before using it
  ```c
  MPI_Type_commit(newtype)
  
  newtype the new datatype to commit
  ```

- A type should be freed after it is no longer needed
  ```c
  MPI_Type_free(newtype)
  
  newtype the datatype for decommision
  ```
## Datatype constructors

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Type_contiguous</code></td>
<td>contiguous datatypes</td>
</tr>
<tr>
<td><code>MPI_Type_vector</code></td>
<td>regularly spaced datatype</td>
</tr>
<tr>
<td><code>MPI_Type_indexed</code></td>
<td>variably spaced datatype</td>
</tr>
<tr>
<td><code>MPI_Type_create_subarray</code></td>
<td>subarray within a multi-dimensional array</td>
</tr>
<tr>
<td><code>MPI_Type_create_hvector</code></td>
<td>like vector, but uses bytes for spacings</td>
</tr>
<tr>
<td><code>MPI_Type_create_hindexed</code></td>
<td>like index, but uses bytes for spacings</td>
</tr>
<tr>
<td><code>MPI_Type_create_struct</code></td>
<td>fully general datatype</td>
</tr>
</tbody>
</table>
MPI_TYPE_VECTOR

Creates a new type from equally spaced identical block

\[ \text{MPI\_Type\_vector}(\text{count}, \text{blocklen}, \text{stride}, \text{oldtype}, \text{newtype}) \]

- **count**: number of blocks
- **blocklen**: number of elements in each block
- **stride**: displacement between the blocks

Example:

\[ \text{MPI\_Type\_vector}(3, 2, 3, \text{oldtype}, \text{newtype}) \]

- **oldtype**: BLOCKLEN=2
- **newtype**: STRIDE=3
integer, parameter :: n=3, m=3
real, dimension(n,m) :: a
integer :: rowtype
! create a derived type
call mpi_type_vector(m, 1, n, mpi_real, rowtype, ierr)
call mpi_type_commit(rowtype, ierr)
! send a row
call mpi_send(a, 1, rowtype, dest, tag, comm, ierr)
! free the type after it is not needed
call mpi_type_free(rowtype, ierr)
MPI_TYPE_INDEXED

Creates a new type from blocks comprising identical elements

- The size and displacements of the blocks may vary

\[
\text{MPI\_Type\_indexed}(\text{count, blocklens, displs, oldtype, newtype})
\]

- **count**: number of blocks
- **blocklens**: lengths of the blocks (array)
- **displs**: displacements (array) in extent of oldtypes

\[
\begin{align*}
\text{count} &= 3 \\
\text{blocklens} &= (2,3,1) \\
\text{disps} &= (0,3,8)
\end{align*}
\]

oldtype

newtype
/* Upper triangular matrix */
double a[100][100];
int disp[100], blocklen[100], int i;
MPI_Datatype upper;
/* compute start and size of rows */
for (i=0;i<100;i++)
{
    disp[i]=100*i+i;
    blocklen[i]=100-i;
}
/* create a datatype for upper triangular matrix */
MPI_Type_indexed(100,blocklen,disp,MPI_DOUBLE,&upper);
MPI_Type_commit(&upper);
/* ... send it ... */
MPI_Send(a,1,upper,dest, tag, MPI_COMM_WORLD);
MPI_Type_free(&upper);
MPI_TYPE_CREATE_SUBARRAY

Creates a type describing an $N$-dimensional subarray within an $N$-dimensional array

\[
\text{MPI\_Type\_create\_subarray}(\text{ndims}, \text{sizes}, \text{subsizes}, \\
\text{offsets}, \text{order}, \text{oldtype}, \text{newtype})
\]

- **ndims**: number of array dimensions
- **sizes**: number of array elements in each dimension (array)
- **subsizes**: number of subarray elements in each dimension (array)
- **offsets**: starting point of subarray in each dimension (array)
- **order**: storage order of the array. Either MPI\_ORDER\_C or MPI\_ORDER\_FORTRAN
Example: subarray

```c
int array_size[2] = {5,5};
int subarray_size[2] = {2,2};
int subarray_start[2] = {1,1};
MPI_Datatype subtype;
double **array

for (i=0; i<array_size[0]; i++)
    for (j=0; j<array_size[1]; j++)
        array[i][j] = rank;

MPI_Type_create_subarray(2, array_size, subarray_size, subarray_start,
                          MPI_ORDER_C, MPI_DOUBLE, &subtype);

MPI_Type_commit(&subtype);
if (rank==0)
    MPI_Recv(array[0], 1, subtype, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if (rank==1)
    MPI_Send(array[0], 1, subtype, 0, 123, MPI_COMM_WORLD);

Rank 0: original array
0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0

Rank 0: array after receive
0.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 1.0 0.0 0.0 0.0
0.0 1.0 1.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0
```
Example: halo exchange with user defined types

Two-dimensional grid with two-element ghost layers

512 x 512 (+ halo)

```
int array_size[2] = {512 + 2+2, 512 + 2+2};
int x_size[2] = {512,2};
int offset[2] = {0,0};
MPI_Type_create_subarray(2, array_size, x_size, offset, MPI_ORDER_C, MPI_DOUBLE, &x_boundary);
```

```
int y_size[2] = {2,512};
MPI_Type_create_subarray(2, array_size, y_size, offset, MPI_ORDER_C, MPI_DOUBLE, &y_boundary);
```
Example: halo exchange with user defined types

```c
MPI_Sendrecv(array(2,2), 1, x_boundary, left, tag_left,
             array(2,0), 1, x_boundary, right, tag_right,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(array(2,2), 1, y_boundary, down, tag_down,
             array(0,2), 1, y_boundary, up, tag_up,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
...
From non-contiguous to contiguous data

```c
if (myid == 0)
    MPI_Type_vector(n, 1, 2, MPI_FLOAT, &newtype)
...
MPI_Send(A, 1, newtype, 1, ...)
else
    MPI_Recv(B, n, MPI_FLOAT, 0, ...)

if (myid == 0)
    MPI_Send(A, n, MPI_FLOAT, 1, ...)
else
    MPI_Type_vector(n, 1, 2, MPI_FLOAT, &newtype)
...
MPI_Recv(B, 1, newtype, 0, ...)
```
Performance

Performance depends on the datatype - more general datatypes are often slower

Overhead is potentially reduced by:

- Sending one long message instead of many small messages
- Avoiding the need to pack data in temporary buffers

Performance should be tested on target platforms
Performance

Example: Sending a row (in C) of 512x512 matrix on Cray XC30

- Several sends: 10 ms
- Manual packing: 1.1 ms
- User defined type: 0.6 ms
Summary

Derived types enable communication of non-contiguous or heterogenous data with single MPI calls
  – Improves maintainability of program
  – Allows optimizations by the system
  – Performance is implementation dependent

Life cycle of derived type: create, commit, free

MPI provides constructors for several specific types
COMMUNICATION TOPOLOGIES
**Process topologies**

- MPI process topologies allow for simple referencing scheme of processes
  - Process topology defines a new communicator
  - We focus on Cartesian topologies, although graph topologies are also supported

- MPI topologies are virtual
  - No relation to the physical structure of the computer
  - Data mapping "more natural" only to the programmer

- Usually no performance benefits
  - But code becomes more compact and readable
Creating a communicator ordered in Cartesian grid

MPI_Cart_create(oldcomm, ndims, dims, periods, reorder, newcomm)

oldcomm  communicator
ndims    dimension of the Cartesian topology
dims     integer array (size ndims) that defines the number of processes in each dimension
periods  array that defines the periodicity of each dimension
reorder  is MPI allowed to renumber the ranks
newcomm  new Cartesian communicator
Translating rank to coordinates

MPI_Cart_coords(comm, rank, maxdim, coords)

comm  Cartesian communicator
rank  rank to convert
maxdim  dimension of coords
coords  coordinates in Cartesian topology that corresponds to rank

Checking the Cartesian communication topology coordinates for a specific rank
Translating coordinates to rank

MPI_Cart_rank(comm, coords, rank)

comm Cartesian communicator
coords array of coordinates
rank a rank corresponding to coords

Checking the rank of the process at specific Cartesian communication topology coordinates
Creating a Cartesian communication topology

dims(1)=4
dims(2)=4
period=([/ .true., .true. /])
call mpi_cart_create(mpi_comm_world, 2, &
dims, period, .true., comm2d, rc)
call mpi_comm_rank(comm2d, my_id, rc)
call mpi_cart_coords(comm2d, my_id, 2, &
coords, rc)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
</tr>
<tr>
<td>4</td>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>8</td>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
</tr>
<tr>
<td>12</td>
<td>(3,0)</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
</tr>
</tbody>
</table>
How to communicate in a topology

MPI_Cart_shift(comm, direction, displ, source, dest)

comm          Cartesian communicator
direction     shift direction (0 or 1 in 2D)
displ         shift displacement (1 for next cell etc, < 0 for source from "down"/"right" directions)
source        rank of source process
dest          rank of destination process

We shift the grid to define sources/destinations

Note that both source and dest are output parameters. The coordinates of the calling task is implicit input.

With a non-periodic grid, source or dest can land outside of the grid; then MPI_PROC_NULL is returned.
Halo exchange

dims(1)=4

dims(2)=4

period =(/ .true., .true./)

call mpi_cart_create(mpi_comm_world, 2,&
   dims, period, .true., comm2d, rc)
call mpi_cart_shift(comm2d,0,1,nbr_up,nbr_down,rc)
call mpi_cart_shift(comm2d,1,1,nbr_left,nbr_right,rc)
...
call mpi_sendrecv(hor_send, msglen, mpi_double_precision, nbr_left,&
   tag_left, hor_recv, msglen, mpi_double_precision, nbr_right,&
   tag_left, comm2d, mpi_status_ignore, rc)
...
call mpi_sendrecv(vert_send, msglen, mpi_double_precision, nbr_up,&
   tag_up, vert_recv, msglen, mpi_double_precision, nbr_down,&
   tag_up, comm2d, mpi_status_ignore, rc)
...
ONE-SIDED COMMUNICATION
Two components of message-passing: sending and receiving

One-sided communication
- Only single process calls data movement functions (*put* or *get*) – remote memory access (RMA)
- Communication patterns specified by only a single process
- Always non-blocking
Why one-sided communication?

- Certain algorithms featuring unstructured communication easier to implement
- Potentially reduced overhead and improved scalability
- Hardware support for remote memory access has been restored in most current-generation architectures
Origin and target

Origin process: a process which calls data movement function

Target process: a process whose memory is accessed
Remote memory access window

*Window* is a region in process’s memory which is made available for remote operations.

Windows are created by collective calls.

Windows may be different in different processes.
Data movement operations in MPI

- PUT data to the memory in target process
  - From local buffer in origin to the window in target
- GET data from the memory of target process
  - From the window in target to the local buffer in origin
- ACCUMULATE data in target process
  - Use local buffer in origin and update the data (e.g. add the data from origin) in the window in target
  - One-sided reduction
Synchronization

Communication takes place within *epochs*

- Synchronization calls start and end an epoch
- There can be multiple data movement calls within an epoch
- An epoch is specific to a particular window

Active synchronization:
- Both origin and target perform synchronization calls

Passive synchronization:
- No MPI calls at target process
One-sided communication in a nutshell

- Define a memory window
- Start an epoch
  - Target: exposure epoch
  - Origin: access epoch
- GET, PUT, and/or ACCUMULATE data
- Complete the communications by ending the epoch
Creating a window

MPI_Win_create(base, size, disp_unit, info, comm, win)

- **base** (pointer to) local memory to expose for RMA
- **size** size of a window in bytes
- **disp_unit** local unit size for displacements in bytes
- **info** hints for implementation
- **comm** communicator
- **win** handle to window

The window object is deallocated with

MPI_Win_free(win)
Starting and ending an epoch

MPI_Win_fence(assert, win)

**assert** optimize for specific usage. Valid values are "0", MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE, MPI_MODE_NOSUCCEED

**win** window handle

- Used both for starting and ending an epoch
  - Should both precede and follow data movement calls

Collective, barrier-like operation
Data movement: Put

\[\text{MPI\_Put}(\text{origin}, \text{origin\_count}, \text{origin\_datatype}, \text{target\_rank}, \text{target\_disp}, \text{target\_count}, \text{target\_datatype}, \text{win})\]

- **origin**: (pointer to) local data to be send to target
- **origin\_count**: number of elements to put
- **origin\_datatype**: MPI datatype for local data
- **target\_rank**: rank of the target task
- **target\_disp**: starting point in target window
- **target\_count**: number of elements in target
- **target\_datatype**: MPI datatype for remote data
- **win**: RMA window
**Data movement: Get**

```c
MPI_Get(origin, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>origin</code></td>
<td>(pointer to) local buffer in which to receive the data</td>
</tr>
<tr>
<td><code>origin_count</code></td>
<td>number of elements to get</td>
</tr>
<tr>
<td><code>origin_datatype</code></td>
<td>MPI datatype for local data</td>
</tr>
<tr>
<td><code>target_rank</code></td>
<td>rank of the target task</td>
</tr>
<tr>
<td><code>target_disp</code></td>
<td>starting point in target window</td>
</tr>
<tr>
<td><code>target_count</code></td>
<td>number of elements from target</td>
</tr>
<tr>
<td><code>target_datatype</code></td>
<td>MPI datatype for remote data</td>
</tr>
<tr>
<td><code>win</code></td>
<td>RMA window</td>
</tr>
</tbody>
</table>
Data movement: Accumulate

MPI_Accumulate(
  origin, origin_count, origin_datatype,
  target_rank, target_disp, target_count,
  target_datatype, op, win)

- **origin**: (pointer to) local data to be accumulated
- **origin_count**: number of elements to put
- **origin_datatype**: MPI datatype for local data
- **target_rank**: rank of the target task
- **target_disp**: starting point in target window
- **target_count**: number of elements for target
- **target_datatype**: MPI datatype for remote data
- **op**: accumulation operation (as in MPI_Reduce)
- **win**: RMA window
Simple example: Put

```c
...
    int data;
    MPI_Win window;

    data = rank;
    // Create window
    MPI_Win_create(&data, sizeof(int), sizeof(int), MPI_INFO_NULL,
                        MPI_COMM_WORLD, &window);

    ...
    MPI_Win_fence(0, window);
    if (rank == 0)
        MPI_Put(&data, 1, MPI_INT, 1, 0, 1, MPI_INT, window);
    MPI_Win_fence(0, window);

    ...
    MPI_Win_free(&window);
```
Limitations for data access

Compatibility of local and remote operations when multiple processes access a window during an epoch

<table>
<thead>
<tr>
<th>Local/remote</th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td><img src="https://example.com/green.png" alt="Green" /></td>
<td><img src="https://example.com/green.png" alt="Green" /></td>
<td><img src="https://example.com/green.png" alt="Green" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
</tr>
<tr>
<td>Store</td>
<td><img src="https://example.com/green.png" alt="Green" /></td>
<td><img src="https://example.com/red.png" alt="Red" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/red.png" alt="Red" /></td>
<td><img src="https://example.com/red.png" alt="Red" /></td>
</tr>
<tr>
<td>Get</td>
<td><img src="https://example.com/green.png" alt="Green" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/green.png" alt="Green" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
</tr>
<tr>
<td>Put</td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/red.png" alt="Red" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
</tr>
<tr>
<td>Acc</td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/red.png" alt="Red" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/yellow.png" alt="Yellow" /></td>
<td><img src="https://example.com/green.png" alt="Green" /></td>
</tr>
</tbody>
</table>

- **Green**: No limitations
- **Yellow**: Operations on non-overlapping parts of window allowed
- **Red**: Not allowed
Advanced synchronization

Assert arguments for MPI_Win_fence:

MPI_MODE_NOSTORE
The local window was not updated by local stores (or local get or receive calls) since last synchronization

MPI_MODE_NOPUT
The local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization

MPI_MODE_NOPRECEDE
The fence does not complete any sequence of locally issued RMA calls

MPI_MODE_NOSUCCEED
The fence does not start any sequence of locally issued RMA calls
Advanced synchronization

More control on epochs can be obtained by starting and ending the exposure and access epochs separately

Target: Exposure epoch
– Start: MPI_Win_post
– End: MPI_Win_wait

Origin: Access epoch
– Start: MPI_Win_start
– End: MPI_Win_compete
Enhancements in MPI-3

- New window creation function: MPI_Win_allocate
  - Allocate memory and create a window at the same time

- Dynamic windows: MPI_Win_create_dynamic, MPI_WinAttach, MPI_Win_detach
  - Non-collective exposure of memory
Enhancements in MPI-3

- New data movement operations:
  MPI_Get_accumulate, MPI_Fetch_and_op, MPI_Compare_and_swap

- New memory model
  MPI_Win_allocate_shared
  - Allocate memory which is shared between MPI tasks

- Enhancements for passive target synchronization
Performance considerations

- Performance of the one-sided approach is highly implementation-dependent
- Maximize the amount of operations within an epoch
- Provide the assert parameters for MPI_Win_fence
Summary

One-sided communication allows communication patterns to be specified from a single process.

Can reduce synchronization overheads and provide better performance especially on recent hardware.

Basic concepts:
- Creation of the memory window
- Communication epoch
- Data movement operations (MPI_Put, MPI_Get etc.)