

Advanced Features of MPI-3 and MPI-4

Victor Eijkhout

PEARC 2023



Textbooks and repositories:
<https://theartofhpc.com/pcse>

Contact:
eijkhout@tacc.utexas.edu



Version 3 of the MPI standard has added a number of features, some geared purely towards functionality, others with an eye towards efficiency at exascale.

Version 4 adds yet more features for exascale, and more flexible process management.

- MPI-3 as of 2012, 3.1 as of 2015. Fully supported everywhere.
- MPI-4 as of June 2021; Supported in mpich version 4.1, *not* in OpenMPI version 4.
- MPI-4.1 to be ratified late 2023.



Part I

Fortran bindings



The Fortran interface to MPI had some defects. With Fortran2008 these have been largely repaired.

- The trailing error parameter is now optional;
- MPI data types are now actual **Type** objects, rather than **Integer**
- Strict type checking on arguments.



New module:

```
use mpi_f08      ! for Fortran2008
use mpi          ! for Fortran90
```

True Fortran bindings as of the 2008 standard. Provided in

- Intel compiler version 18 or newer,
- gcc 9 and later (not with Intel MPI, use mvapich).



Old Fortran90 style:

```
1  call MPI_Init(ierr)
2  ! your code
3  call MPI_Finalize(ierr)
```

New Fortran2008 style:

```
1  call MPI_Init()
2  ! your code
3  call MPI_Finalize()
```



Communicators are now derived types:

```
1  !! Fortran 2008 interface
2  use mpi_f08
3  Type(MPI_Comm) :: comm = MPI_COMM_WORLD
```

```
1  !! Fortran legacy interface
2  use mpi
3  !! deprecated: #include <mpif.h>
4  Integer :: comm = MPI_COMM_WORLD
```



Requests are also derived types

note that ...NULL entities are now objects, not integers

```
1  !! waitnull.F90
2  Type(MPI_Request),dimension(:),allocatable :: requests
3  allocate(requests(ntids-1))
4      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
5      if ( .not. requests(index)==MPI_REQUEST_NULL) then
6          print *, "This request should be null:", index
```

(Q for the alert student: do you see anything halfway remarkable about that index?)



More handles that are now derived types:

```
1  Type(MPI_Datatype) :: newtype ! F2008
2  Integer              :: newtype ! F90
```

Also: MPI_Comm, MPI_Datatype, MPI_Errhandler, MPI_Group, MPI_Info,
MPI_File, MPI_Message, MPI_Op, MPI_Request, MPI_Status, MPI_Win



Fortran2008: status is a **Type** with fields:

```
1  !! anysource.F90
2  Type(MPI_Status)  :: status
3      allocate(recv_buffer(ntids-1))
4      do p=0,ntids-2
5          call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
6                      MPI_ANY_SOURCE,0,comm,status)
7          sender = status%MPI_SOURCE
```

Fortran90: status is an array with named indexing

```
1  !! anysource.F90
2  integer :: status(MPI_STATUS_SIZE)
3      allocate(recv_buffer(ntids-1))
4      do p=0,ntids-2
5          call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
6                      MPI_ANY_SOURCE,0,comm,status,err)
7          sender = status(MPI_SOURCE)
```



Type checking catches potential problems:

```
1  !! typecheckarg.F90
2  integer,parameter :: n=2
3  Integer,dimension(n) :: source
4  call MPI_Init()
5  call MPI_Send(source,MPI_INTEGER,n, &
6             1,0,MPI_COMM_WORLD)
```

typecheck.F90(20): error #6285:

```
There is no matching specific subroutine
for this generic subroutine call.    [MPI_SEND]
call MPI_Send(source,MPI_INTEGER,n,
-----^
```



Type checking does not catch all problems:

```
1  !! typecheckbuf.F90
2  integer,parameter :: n=1
3  Real,dimension(n) :: source
4  call MPI_Init()
5  call MPI_Send(source,n,MPI_INTEGER, &
6               1,0,MPI_COMM_WORLD)
```

Buffer/type mismatch is not caught.



Part II

Big data communication



This section discusses big messages.

Commands learned:

- `MPI_Send_c`, `MPI_Allreduce_c`, `MPI_Get_count_c` etc. (MPI-4)



- There is no problem allocating large buffers:

```
1  size_t bigsize = 1<<33;  
2  double *buffer =  
3      (double*) malloc(bigsize*sizeof(double));
```

- But you can not tell MPI how big the buffer is:

```
1  MPI_Send(buffer, bigsize, MPI_DOUBLE, ...) // WRONG
```

because the size argument has to be `int`.



Count type since MPI 3
C:

```
1 MPI_Count count;
```

Fortran:

```
1 Integer(kind=MPI_COUNT_KIND) :: count
```

Big enough for

- `int`;
- `MPI_Aint`, used in one-sided;
- `MPI_Offset`, used in file I/O.

However, this type could not be used in MPI-3 to describe send buffers.



C: routines with `_c` suffix

```
1 MPI_Count count;  
2 MPI_Send_c( buff, count, MPI_INT, ... );
```

also `MPI_Reduce_c`, `MPI_Get_c`, ... (some 190 routines in all)

Fortran: polymorphism rules

```
1 Integer(kind=MPI_COUNT_KIND) :: count  
2 call MPI_Send( buff, count, MPI_INTEGER, ... )
```



```
1 // pingpongbig.c
2 assert( sizeof(MPI_Count)>4 );
3 for ( int power=3; power<=10; power++) {
4     MPI_Count length=pow(10,power);
5     buffer = (double*)malloc( length*sizeof(double) );
6     MPI_Ssend_c
7         (buffer,length,MPI_DOUBLE,
8          processB,0,comm);
9     MPI_Recv_c
10        (buffer,length,MPI_DOUBLE,
11         processB,0,comm,MPI_STATUS_IGNORE);
```



```
1  !! pingpongbig.F90
2  integer :: power, countbytes
3  Integer(KIND=MPI_COUNT_KIND) :: length
4  call MPI_Sizeof(length, countbytes, ierr)
5  if (procno==0) &
6      print *, "Bytes in count:", countbytes
7      length = 10**power
8      allocate( senddata(length), recvdata(length) )
9      call MPI_Send(senddata, length, MPI_DOUBLE_PRECISION, &
10         processB, 0, comm)
11     call MPI_Recv(recvdata, length, MPI_DOUBLE_PRECISION, &
12        processB, 0, comm, MPI_STATUS_IGNORE)
```



Name	Param name	Explanation	C type	F type
MPI_Send (
MPI_Send_c (
	buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)
	count	number of elements in send buffer	[int MPI_Count	INTEGER
	datatype	datatype of each send buffer element	MPI_Datatype	TYPE(MPI_Datatype)
	dest	rank of destination	int	INTEGER
	tag	message tag	int	INTEGER
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
)			



C:

```
1 MPI_Count count;  
2 MPI_Get_count_c( &status,MPI_INT, &count );  
3 MPI_Get_elements_c( &status,MPI_INT, &count );
```

Fortran:

```
1 Integer(kind=MPI_COUNT_KIND) :: count  
2 call MPI_Get_count( status,MPI_INTEGER,count )  
3 call MPI_Get_elements( status,MPI_INTEGER,count )
```



MPI-3 mechanism, deprecated in MPI-4.1:
send a number of contiguous types:

```
1 MPI_Datatype blocktype;  
2 MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);  
3 MPI_Type_commit(&blocktype);  
4 if (procno==sender) {  
5     MPI_Send(source,nblocks,blocktype,receiver,0,comm);
```

By composing types you can make a 'big type'. Use
`MPI_Type_get_extent_x`, `MPI_Type_get_true_extent_x`, `MPI_Get_elements_x`
to query.

```
1 MPI_Count recv_count;  
2 MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```



Part III

Advanced collectives



- Collectives are blocking.
- Compare blocking/non-blocking sends:
`MPI_Send` → `MPI_Isend`
immediate return of control, produce request object.
- Non-blocking collectives:
`MPI_Bcast` → `MPI_Ibcast`
Same:

```
1 MPI_Isomething( <usual arguments>, MPI_Request *req);
```
- Considerations:
 - Calls return immediately;
 - the usual story about buffer reuse
 - Requires `MPI_Wait...` for completion.
 - Multiple collectives can complete in any order
- Why?
 - Use for overlap communication/computation
 - Imbalance resilience
 - Allows pipelining



Name	Param name	Explanation	C type	F type
MPI_Ibcast (
MPI_Ibcast_c (
	buffer	starting address of buffer	void*	TYPE(*), DIMENSION(..)
	count	number of entries in buffer	[int MPI_Count	INTEGER
	datatype	datatype of buffer	MPI_Datatype	TYPE(MPI_Datatype)
	root	rank of broadcast root	int	INTEGER
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
	request	communication request	MPI_Request*	TYPE(MPI_Request)
)			



Independent collective and local operations:

$$y \leftarrow Ax + (x^t x)y$$

```
1 MPI_Iallreduce( .... x ..., &request);  
2 // compute the matrix vector product  
3 MPI_Wait(request);  
4 // do the addition
```



Do two reductions (on the same communicator) with different operators simultaneously:

$$\alpha \leftarrow x^t y$$
$$\beta \leftarrow \|z\|_\infty$$

which translates to:

```
1  MPI_Request reqs[2];
2  MPI_Iallreduce
3      ( &local_xy, &global_xy, 1,MPI_DOUBLE,MPI_SUM,comm,
4        &(reqs[0]) );
5  MPI_Iallreduce
6      ( &local_xinf,&global_xin,1,MPI_DOUBLE,MPI_MAX,comm,
7        &(reqs[1]) );
8  MPI_Waitall(2,reqs,MPI_STATUSES_IGNORE);
```

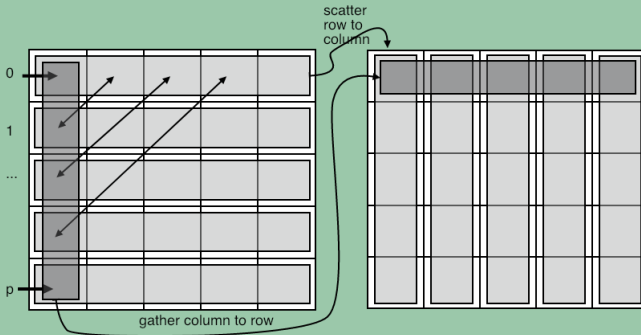


Blocking and non-blocking don't match: either all processes call the non-blocking or all call the blocking one. Thus the following code is incorrect:

```
1  if (rank==root)
2      MPI_Reduce( &x /* ... */ root,comm );
3  else
4      MPI_Ireduce( &x /* ... */ root,comm,&req);
```

This is unlike the point-to-point behavior of non-blocking calls: you can catch a message with `MPI_Irecv` that was sent with `MPI_Send`.





Every process needs to do a scatter or gather.



Transpose matrix by scattering all rows simultaneously.
Each scatter involves all processes, but with a different spanning tree.

```
1  MPI_Request scatter_requests[nprocs];
2  for (int iproc=0; iproc<nprocs; iproc++) {
3      MPI_Isscatter( regular,1,MPI_DOUBLE,
4                    &(transpose[iproc]),1,MPI_DOUBLE,
5                    iproc,comm,scatter_requests+iproc);
6  }
7  MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE);
```



Persistent collectives



Similar to persistent send/recv:

```
1 MPI_Allreduce_init( ...., &request );  
2 for ( ... ) {  
3     MPI_Start( request );  
4     MPI_Wait( request );  
5 }  
6 MPI_Request_free( &request );
```

Available for all collectives and neighborhood collectives.



```

1 // powerpersist1.c
2 double localnorm,globalnorm=1.;
3 MPI_Request reduce_request;
4 MPI_Allreduce_init
5     ( &localnorm,&globalnorm,1,MPI_DOUBLE,MPI_SUM,
6       comm,MPI_INFO_NULL,&reduce_request);
7 for (int it=0; ; it++) {
8     /*
9      * Matrix vector product
10     */
11     matmult(indata,outdata,buffersize);
12
13     // start computing norm of output vector
14     localnorm = local_l2_norm(outdata,buffersize);
15     double old_globalnorm = globalnorm;
16     MPI_Start( &reduce_request );
17
18     // end computing norm of output vector
19     MPI_Wait( &reduce_request,MPI_STATUS_IGNORE );
20     globalnorm = sqrt(globalnorm);
21     // now 'globalnorm' is the L2 norm of 'outdata'
22     scale(outdata,indata,buffersize,1./globalnorm);
23 }

```



Both request-based.

- Non-blocking is 'ad hoc': buffer info not known before the collective call.
- Persistent allows 'planning ahead': management of internal buffers and such.

Request handling:

- Non-blocking: wait deallocates the request
- Persistent: wait deactivates the request



Non-blocking barrier



- Barrier is not *time* synchronization but *state* synchronization.
- Test on non-blocking barrier: 'has everyone reached some state'



- Some processes decide locally to alter their structure
- ... need to communicate that to neighbors
- Problem: neighbors don't know whether to expect update calls, if at all.
- Solution:
 - send update msgs, if any;
 - then post barrier.
 - Everyone probe for updates, test for barrier.



- Distributed termination detection (Matocha and Kamp, 1998): draw a global conclusion with local operations
- Everyone posts the barrier when done;
- keeps doing local computation while testing for the barrier to complete



Name	Param name	Explanation	C type	F type
MPI_Ibarrier (comm request)		communicator communication request	MPI_Comm MPI_Request*	TYPE(MPI_Comm) TYPE(MPI_Request)



Do sends, post barrier.

```
1 // ibarrierprobe.c
2 if (i_do_send) {
3     /*
4      * Pick a random process to send to,
5      * not yourself.
6      */
7     int receiver = rand()%nprocs;
8     MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
9 }
10 /*
11  * Everyone posts the non-blocking barrier
12  * and gets a request to test/wait for
13  */
14 MPI_Request barrier_request;
15 MPI_Ibarrier(comm,&barrier_request);
```



Poll for barrier and messages

```
1  for ( ; ; step++) {
2      int barrier_done_flag=0;
3      MPI_Test(&barrier_request,&barrier_done_flag,
4              MPI_STATUS_IGNORE);
5      //stop if you're done!
6      if (barrier_done_flag) {
7          break;
8      } else {
9          // if you're not done with the barrier:
10         int flag; MPI_Status status;
11         MPI_Iprobe
12             ( MPI_ANY_SOURCE,MPI_ANY_TAG,
13              comm, &flag, &status );
14         if (flag) {
15             // absorb message!
```



Part IV

Shared memory



Myth:

MPI processes use network calls, whereas OpenMP threads access memory directly, therefore OpenMP is more efficient for shared memory.

Truth:

MPI implementations use copy operations when possible, whereas OpenMP has thread overhead, and affinity/coherence problems.

Main problem with MPI on shared memory: data duplication.

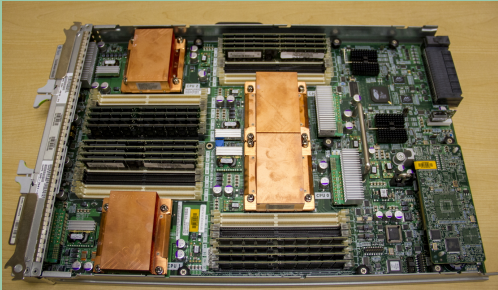


- Shared memory access: two processes can access each other's memory through `double*` (and such) pointers, if they are on the same shared memory.
- Limitation: only window memory.
- Non-use case: remote update. This has all the problems of traditional shared memory (race conditions, consistency).
- Good use case: every process needs access to large read-only dataset
Example: ray tracing.



- MPI uses optimizations for shared memory: copy instead of socket call
- One-sided offers 'fake shared memory': yes, can access another process' data, but only through function calls.
- MPI-3 shared memory gives you a pointer to another process' memory,
if that process is on the same shared memory.





- Cluster node has shared memory
- Memory is attached to specific socket
- beware Non-Uniform Memory Access (NUMA) effects



Here is the high level overview; details next.

- Use `MPI_Comm_split_type` to find processes on the same shared memory
- Use `MPI_Win_allocate_shared` to create a window between processes on the same shared memory (MPI-4.1: other window creation calls allowed, but the burden is on you!)
- Use `MPI_Win_shared_query` to get pointer to another process' window data.
- You can now use `memcpy` instead of `MPI_Put`.



- `MPI_Comm_split_type` splits into communicators of same type.
- Use type: `MPI_COMM_TYPE_SHARED` splitting by shared memory.

Code:

```
1 // commsplitttype.c
2 MPI_Info info;
3 MPI_Comm_split_type
4   (MPI_COMM_WORLD,
5    MPI_COMM_TYPE_SHARED,
6    procno, info, &sharedcomm);
7 MPI_Comm_size
8   (sharedcomm, &new_nprocs);
9 MPI_Comm_rank
10  (sharedcomm, &new_procno);
```

Output:

```
1 make[3]: 'commsplitttype' is up to date.
2 TACC: Starting up job 4356245
3 TACC: Starting parallel tasks...
4 There are 10 ranks total
5 [0] is processor 0 in a shared group of
   ↳5, running on
   ↳c209-010.frontera.tacc.utexas.edu
6 [5] is processor 0 in a shared group of
   ↳5, running on
   ↳c209-011.frontera.tacc.utexas.edu
7 TACC: Shutdown complete. Exiting.
```



Write a program that uses `MPI_Comm_split_type` to analyze for a run

1. How many nodes there are;
2. How many processes there are on each node.

If you run this program on an unequal distribution, say 10 processes on 3 nodes, what distribution do you find?

```
1  Nodes: 3; processes: 10
2  TACC:  Starting up job 4210429
3  TACC:  Starting parallel tasks...
4  There are 3 nodes
5  Node sizes: 4 3 3
6  TACC:  Shutdown complete. Exiting.
```



Use `MPI_Win_allocate_shared` to create a window that can be shared;

- Has to be on a communicator on shared memory
- Example: window is one double.

```
1 // sharedbulk.c
2 MPI_Win node_window;
3 MPI_Aint window_size; double *window_data;
4 if (onnode_procid==0)
5     window_size = sizeof(double);
6 else window_size = 0;
7 MPI_Win_allocate_shared
8     ( window_size,sizeof(double),MPI_INFO_NULL,
9     nodecomm,
10     &window_data,&node_window);
```



Use `MPI_Win_shared_query`:

```
1 MPI_Aint window_size0; int window_unit; double *win0_addr;  
2 MPI_Win_shared_query  
3   ( node_window, 0,  
4     &window_size0, &window_unit, &win0_addr );
```



Name	Param name	Explanation	C type	F type
MPI_Win_shared_query (
MPI_Win_shared_query_c (
	win	shared memory window object	MPI_Win	TYPE(MPI_Win)
	rank	rank in the group of window win or MPI_PROC_NULL	int	INTEGER
	size	size of the window segment	MPI_Aint*	INTEGER (KIND=MPI_ADDRESS_KIND)
	disp_unit	local unit size for displacements, in bytes	[int* MPI_Aint*	INTEGER
	baseptr	address for load/store access to window segment	void*	TYPE(C_PTR)
)			



Memory will be allocated contiguously
convenient for address arithmetic,
not for NUMA: set `alloc_shared_noncontig` true in `MPI_Info` object.

Example: each window stores one double. Measure distance in bytes:

Strategy: default behavior of
shared window allocation

Distance 1 to zero: 8
Distance 2 to zero: 16

Strategy: allow non-contiguous
shared window allocation

Distance 1 to zero: 4096
Distance 2 to zero: 8192

Question: what is going on here?



- Application: ray tracing:
large read-only data structure describing the scene
- traditional MPI would duplicate:
excessive memory demands
- Better: allocate shared data on process 0 of the shared communicator
- Everyone else points to this object.



MPI-4: split by other hardware features through
`MPI_COMM_TYPE_HW_GUIDED` and `MPI_Get_hw_resource_types`)

```
1  
2  \endinput
```



Part V

Atomic operations



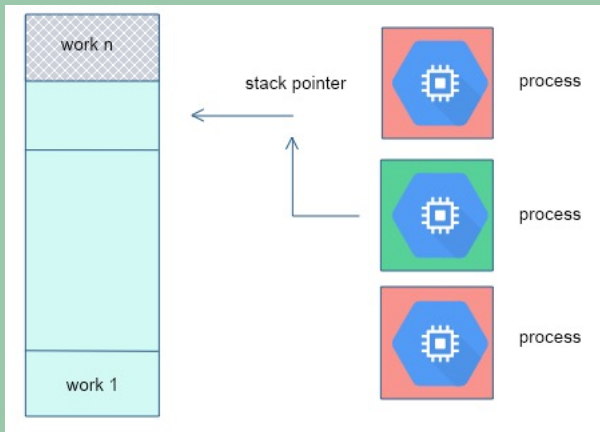
MPI-1/2 lacked tools for race condition-free one-sided communication. These have been added in MPI-3.



Emulating shared memory with one-sided communication

- One process stores a table of work descriptors, and a 'stack pointer' stating how many there are.
- Each process reads the pointer, reads the corresponding descriptor, and decrements the pointer; and
- A process that has read a descriptor then executes the corresponding task.
- Non-collective behavior: processes only take a descriptor when they are available.





- One process has a counter, which models the shared memory;
- Each process, if available, reads the counter; and
- ... decrements the counter.
- No actual work: random decision if process is available.



Shared memory problems: what is a race condition? TA

Race condition: outward behavior depends on timing/synchronization of low-level events.
In shared memory associated with shared data.

Example:

Init: $I=0$
process 1: $I=I+2$
process 2: $I=I+3$

scenario 1.	scenario 2.	scenario 3.
$I = 0$		
read $I = 0$ local $I = 2$ write $I = 2$	read $I = 0$ local $I = 2$ write $I = 2$	read $I = 0$ local $I = 2$ write $I = 2$
write $I = 3$	read $I = 0$ local $I = 3$ write $I = 3$	read $I = 2$ local $I = 5$ write $I = 5$
$I = 3$	$I = 2$	$I = 5$

(In MPI, the read/write would be **MPI_Get** / **MPI_Put** calls)



```
1 // countdownput.c
2 MPI_Win_fence(0,the_window);
3 int counter_value;
4 MPI_Get( &counter_value,1,MPI_INT,
5         counter_process,0,1,MPI_INT,
6         the_window);
7 MPI_Win_fence(0,the_window);
8 if (i_am_available) {
9     int decrement = -1;
10    counter_value += decrement;
11    MPI_Put
12        ( &counter_value, 1,MPI_INT,
13         counter_process,0,1,MPI_INT,
14         the_window);
15 }
16 MPI_Win_fence(0,the_window);
```



- The multiple `MPI_Put` calls conflict.
- Code is correct if in each iteration there is only one writer.
- Question: In that case, can we take out the middle fence?
- Question: what is wrong with

```
1  MPI_Win_fence(0,the_window);
2  if (i_am_available) {
3      MPI_Get( &counter_value, ... )
4      MPI_Win_fence(0,the_window);
5      MPI_Put( ... )
6  }
7  MPI_Win_fence(0,the_window);

?
```




```
1 // countdownacc.c
2 MPI_Win_fence(0,the_window);
3 int counter_value;
4 MPI_Get( &counter_value,1,MPI_INT,
5         counter_process,0,1,MPI_INT,
6         the_window);
7 MPI_Win_fence(0,the_window);
8 if (i_am_available) {
9     int decrement = -1;
10    MPI_Accumulate
11        ( &decrement,          1,MPI_INT,
12         counter_process,0,1,MPI_INT,
13         MPI_SUM,
14         the_window);
15 }
16 MPI_Win_fence(0,the_window);
```



- `MPI_Accumulate` is atomic, so no conflicting writes.
- What is the problem?
- Answer: Processes are not reading unique `counter_value` values.
- Conclusion: Read and update need to come together:
read unique value and immediately update.

Atomic 'get-and-set-with-no-one-coming-in-between':

`MPI_Fetch_and_op` / `MPI_Get_accumulate`.

Former is simple version: scalar only.



Name	Param name	Explanation	C type	F type
MPI_Fetch_and_op (
	origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)
	result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)
	datatype	datatype of the entry in origin, result, and target buffers	MPI_Datatype	TYPE(MPI_Datatype)
	target_rank	rank of target	int	INTEGER
	target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
	op	reduce operation	MPI_Op	TYPE(MPI_Op)
	win	window object	MPI_Win	TYPE(MPI_Win)
)				



```
1  MPI_Win_fence(0,the_window);
2  int
3      counter_value;
4  if (i_am_available) {
5      int
6          decrement = -1;
7      total_decrement++;
8      MPI_Fetch_and_op
9          ( /* operate with data from origin: */    &decrement,
10           /* retrieve data from target:          */    &counter_value,
11            MPI_INT, counter_process, 0, MPI_SUM,
12            the_window);
13  }
14  MPI_Win_fence(0,the_window);
15  if (i_am_available) {
16      my_counter_values[n_my_counter_values++] = counter_value;
17  }
```



MPI type	meaning	applies to
<code>MPI_MAX</code>	maximum	integer, floating point
<code>MPI_MIN</code>	minimum	
<code>MPI_SUM</code>	sum	integer, floating point, complex, multilanguage types
<code>MPI_REPLACE</code>	overwrite	
<code>MPI_NO_OP</code>	no change	C integer, logical
<code>MPI_PROD</code>	product	
<code>MPI_LAND</code>	logical and	
<code>MPI_LOR</code>	logical or	
<code>MPI_LXOR</code>	logical xor	integer, byte, multilanguage types
<code>MPI_BAND</code>	bitwise and	
<code>MPI_BOR</code>	bitwise or	
<code>MPI_BXOR</code>	bitwise xor	<code>MPI_DOUBLE_INT</code> and such
<code>MPI_MAXLOC</code>	max value and location	
<code>MPI_MINLOC</code>	min value and location	

No user-defined operators.



We are using fences, which are collective.
What if a process is still operating on its local work?

Better (but more tricky) solution:
use passive target synchronization and locks.



```
1  if (rank == 0) {  
2      MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);  
3      MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);  
4      MPI_Win_unlock (1, win);  
5  }
```

No action on the target required!



Investigate atomic updates using passive target synchronization. Use `MPI_Win_lock` with an exclusive lock, which means that each process only acquires the lock when it absolutely has to.

- All processs but one update a window:

```
1  int one=1;
2  MPI_Fetch_and_op(&one, &readout,
3      MPI_INT, repo, zero_disp, MPI_SUM,
4      the_win);
```

- while the remaining process spins until the others have performed their update.

Use an atomic operation for the latter process to read out the shared value.

Can you replace the exclusive lock with a shared one?



As exercise 2, but now use a shared lock: all processes acquire the lock simultaneously and keep it as long as is needed.

The problem here is that coherence between window buffers and local variables is now not forced by a fence or releasing a lock. Use `MPI_Win_flush_local` to force coherence of a window (on another process) and the local variable from `MPI_Fetch_and_op`.



Part VI

Partitioned communication



Hybrid scenario:
multiple threads contribute to one large message

Partitioned send/recv:
the contributions can be declared/tested

Flexibility:
MPI can send big message or pipeline or send-on-demand



Buffer consists of equal sized partitions:

```
1 // partition.c
2 int bufsize = nparts*SIZE;
3 int *partitions = (int*)malloc((nparts+1)*sizeof(int));
4 for (int ip=0; ip<=nparts; ip++)
5     partitions[ip] = ip*SIZE;
6 if (procno==src) {
7     double *sendbuffer = (double*)malloc(bufsize*sizeof(double));
```



Similar to init calls for persistent sends,
but specify the number of partitions.

```
1 MPI_Psend_init
2   (sendbuffer, nparts, SIZE, MPI_DOUBLE, tgt, 0,
3    comm, MPI_INFO_NULL, &send_request);
4 MPI_Precv_init
5   (recvbuffer, nparts, SIZE, MPI_DOUBLE, src, 0,
6    comm, MPI_INFO_NULL, &recv_request);
```



Sender-side scenario, multiple messages:

```
1  MPI_Request send_request;
2  MPI_Psend_init
3      (sendbuffer, nparts, SIZE, MPI_DOUBLE, tgt, 0,
4       comm, MPI_INFO_NULL, &send_request);
5  for (int it=0; it<ITERATIONS; it++) {
6      MPI_Start(&send_request);
7      for (int ip=0; ip<nparts; ip++) {
8          fill_buffer(sendbuffer, partitions[ip], partitions[ip+1], ip);
9          MPI_Pready(ip, send_request);
10         }
11     MPI_Wait(&send_request, MPI_STATUS_IGNORE);
12 }
13 MPI_Request_free(&send_request);
```



Use

```
1 MPI_Parrived(recv_request, ipart, &flag);
```

to test for arrived partitions.



Receiver-side: handle partitions as they come in:

```
1  double *recvbuffer = (double*)malloc(bufsize*sizeof(double));
2  MPI_Request recv_request;
3  MPI_Precv_init
4      (recvbuffer,nparts,SIZE,MPI_DOUBLE,src,0,
5       comm,MPI_INFO_NULL,&recv_request);
6  for (int it=0; it<ITERATIONS; it++) {
7      MPI_Start(&recv_request); int r=1,flag;
8      for (int ip=0; ip<nparts; ip++) // cycle this many times
9          for (int ap=0; ap<nparts; ap++) { // check specific part
10             MPI_Parrived(recv_request,ap,&flag);
11             if (flag) {
12                 r *= chck_buffer
13                     (recvbuffer,partitions[ap],partitions[ap+1],ap);
14                 break; }
15             }
16      MPI_Wait(&recv_request,MPI_STATUS_IGNORE);
17  }
18  MPI_Request_free(&recv_request);
```



Part VII

Sessions model



MPI is started exactly once:

- MPI can not close down and restart.
- Libraries using MPI need to agree on threading and such.

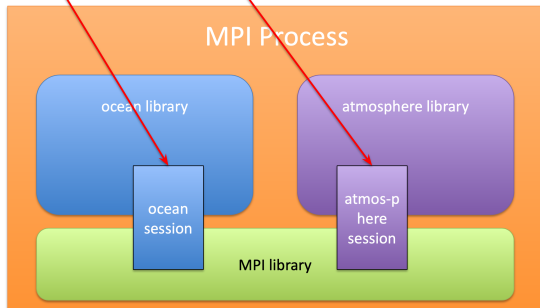
MPI Process

```
// Library 1 (thread)
MPI_Initialized(&flag);
if (!flag) MPI_Init(...);
```

```
// Library 2 (thread)
MPI_Initialized(&flag);
if (!flag) MPI_Init(...);
```



Unique handles to the underlying MPI library



- World model: what you have been doing so far;
Start with `MPI_COMM_WORLD` and make subcommunicators,
or spawn new world communicators and bridge them
- Session model: have multiple sessions active,
each starting/ending MPI separately.



- Create a session;
- a session has multiple 'process sets'
- from a process set you make a communicator;
- Potentially create multiple sessions in one program run
- Can not mix objects from multiple simultaneous sessions, or from session and world model



```
1 // session.c
2 MPI_Info session_request_info = MPI_INFO_NULL;
3 MPI_Info_create(&session_request_info);
4 char thread_key[] = "mpi_thread_support_level";
5 MPI_Info_set(session_request_info,
6               thread_key, "MPI_THREAD_MULTIPLE");
```

Info object can also be `MPI_INFO_NULL`,
then

```
1 MPI_Session the_session;
2 MPI_Session_init
3   ( session_request_info, MPI_ERRORS_ARE_FATAL,
4     &the_session );
5 MPI_Session_finalize( &the_session );
```



Process sets, identified by name (not a data type):

```
1  int npsets;
2  MPI_Session_get_num_psets
3      ( the_session, MPI_INFO_NULL, &npsets );
4  if (mainproc)
5      printf("Number of process sets: %d\n", npsets);
6  for (int ipset=0; ipset<npsets; ipset++) {
7      int len_pset; char name_pset[MPI_MAX_PSET_NAME_LEN];
8      MPI_Session_get_nth_pset
9          ( the_session, MPI_INFO_NULL,
10            ipset, &len_pset, name_pset );
11      if (mainproc)
12          printf("Process set %2d: <<%s>>\n",
13                ipset, name_pset);
```

the sets `mpi://SELF` and `mpi://WORLD` are always defined.



Process set → group → communicator

```
1  MPI_Group world_group = MPI_GROUP_NULL;
2  MPI_Comm world_comm = MPI_COMM_NULL;
3  MPI_Group_from_session_pset
4      ( the_session, world_name, &world_group );
5  MPI_Comm_create_from_group
6      ( world_group, "victor-code-session.c",
7        MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL,
8        &world_comm );
9  MPI_Group_free( &world_group );
10 int procid = -1, nprocs = 0;
11 MPI_Comm_size(world_comm, &nprocs);
12 MPI_Comm_rank(world_comm, &procid);
```




```
1 // sessionmulti.c
2 MPI_Info info1 = MPI_INFO_NULL, info2 = MPI_INFO_NULL;
3 char thread_key[] = "mpi_thread_support_level";
4 MPI_Info_create(&info1); MPI_Info_create(&info2);
5 MPI_Info_set(info1,thread_key,"MPI_THREAD_SINGLE");
6 MPI_Info_set(info2,thread_key,"MPI_THREAD_MULTIPLE");
7 MPI_Session session1,session2;
8 MPI_Session_init( info1,MPI_ERRORS_ARE_FATAL,&session1 );
9 MPI_Session_init( info2,MPI_ERRORS_ARE_FATAL,&session2 );
```



```
1 // sessionlib.cxx
2 class Library {
3 private:
4     MPI_Comm world_comm; MPI_Session session;
5 public:
6     Library() {
7         MPI_Info info = MPI_INFO_NULL;
8         MPI_Session_init
9             ( MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &session );
10        char world_name[] = "mpi://WORLD";
11        MPI_Group world_group;
12        MPI_Group_from_session_pset
13            ( session, world_name, &world_group );
14        MPI_Comm_create_from_group
15            ( world_group, "world-session",
16              MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL,
17              &world_comm );
18        MPI_Group_free( &world_group );
19    };
20    ~Library() { MPI_Session_finalize(&session); };
```



```
1  int main(int argc, char **argv) {  
2  
3      Library lib1, lib2;  
4      MPI_Init(0,0);  
5      MPI_Comm world = MPI_COMM_WORLD;  
6      int procno, nprocs;  
7      MPI_Comm_rank(world, &procno);  
8      MPI_Comm_size(world, &nprocs);  
9      auto sum1 = lib1.compute(procno);  
10     auto sum2 = lib2.compute(procno+1);
```

You can not do MPI sends between libraries or library and main seems reasonable.



Part VIII

Process topologies



This section discusses topologies:

- Cartesian topology
- MPI-1 Graph topology
- MPI-3 Graph topology

Commands learned:

- `MPI_Dist_graph_create`, `MPI_DIST_GRAPH`,
`MPI_Dist_graph_neighbors_count`
- `MPI_Neighbor_allgather` and such



- Processes don't communicate at random
- Example: Cartesian grid, each process 4 (or so) neighbors
- Express operations in terms of topology
- Elegance of expression
- MPI can optimize



- Consecutive process numbering often the best:
divide array by chunks
- Not optimal for grids or general graphs:
- MPI is allowed to renumber ranks
- Graph topology gives information from which MPI can deduce renumbering



- Cartesian topology
- Graph topology, globally specified.
Not scalable, do not use!



- Graph topologies locally specified: scalable!
Limit cases: each process specifies its own connectivity one process specifies whole graph.
- Neighborhood collectives:
expression close to the algorithm.



Graph topologies

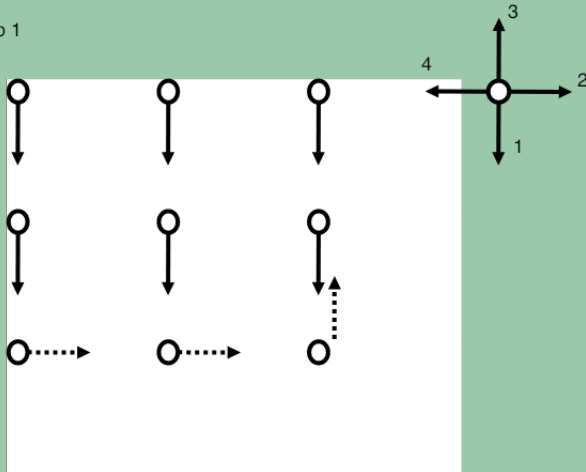


Neighbor exchange, spelled out:

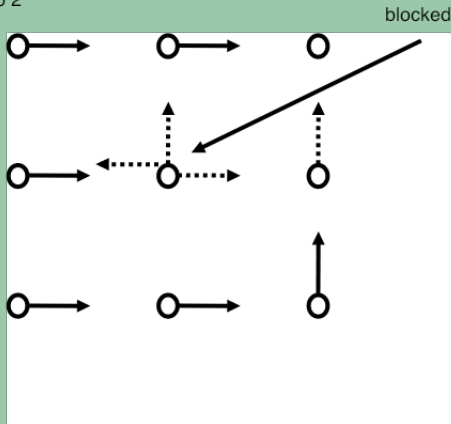
- Each process communicates down/right/up/left
- Send and receive at the same time.
- Can optimally be done in four steps



Step 1



Step 2

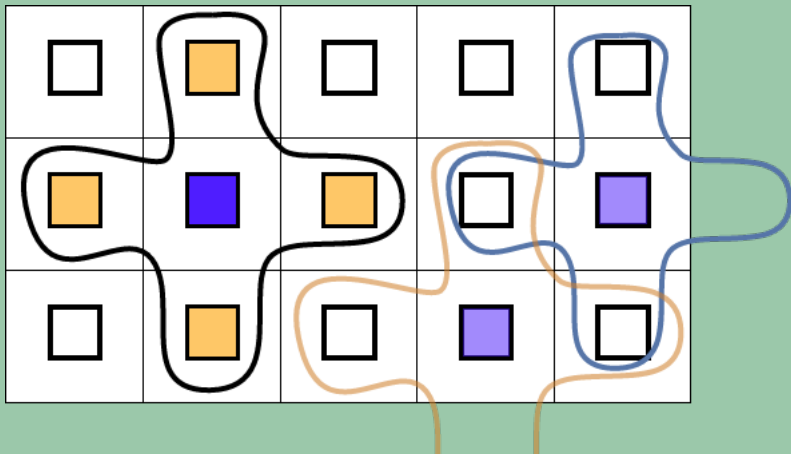


The middle node is blocked because all its targets are already receiving or a channel is occupied:
one missed turn



This is really a 'local gather':
each node does a gather from its neighbors in whatever order.

`MPI_Neighbor_allgather`



- Using `MPI_Isend` / `MPI_Irecv` is like spelling out a collective, imposes order;
- Collectives can use pipelining as opposed to sending a whole buffer;
- Collectives can use spanning trees as opposed to direct connections.



```
1 int MPI_Dist_graph_create
2   (MPI_Comm comm_old, int nsources, const int sources[],
3    const int degrees[], const int destinations[],
4    const int weights[], MPI_Info info, int reorder,
5    MPI_Comm *comm_dist_graph)
```

- *nsources* how many source nodes described? (Usually 1)
- *sources* the processes being described (Usually `MPI_Comm_rank` value)
- *degrees* how many processes to send to
- *destinations* their ranks
- *weights*: usually set to `MPI_UNWEIGHTED`.
- *info*: `MPI_INFO_NULL` will do
- *reorder*: 1 if dynamically reorder processes




```
1  int MPI_Neighbor_allgather
2      (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
3       void *recvbuf, int recvcnt, MPI_Datatype recvtpe,
4       MPI_Comm comm)
```

Like an ordinary `MPI_Allgather`, but
the receive buffer has a length enough for *degree* messages
(instead of comm size).



After `MPI_Neighbor_allgather` data in the buffer is *not* in normal rank order.

- `MPI_Dist_graph_neighbors_count` gives actual number of neighbors.
(Why do you need this?)
- `MPI_Dist_graph_neighbors` lists neighbor numbers.



Name	Param name	Explanation	C type	F type
MPI_Dist_graph_neighbors_count (
	comm	communicator with distributed graph topology	MPI_Comm	TYPE(MPI_Comm)
	indegree	number of edges into this process	int*	INTEGER
	outdegree	number of edges out of this process	int*	INTEGER
	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise	int*	LOGICAL
)				

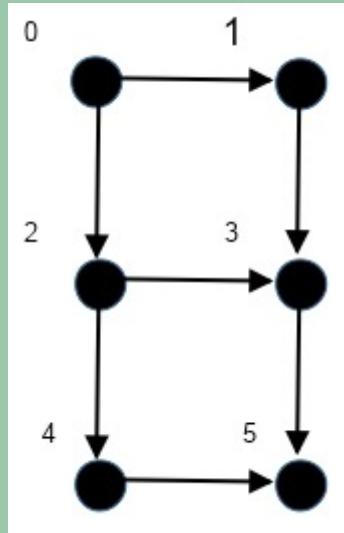


Name	Param name	Explanation	C type	F type
<hr/>				
MPI_Dist_graph_neighbors (comm	communicator with distributed graph topology	MPI_Comm	TYPE(MPI_Comm)
	maxindegree	size of sources and sourceweights arrays	int	INTEGER
	sources	processes for which the calling process is a destination	int[]	INTEGER(maxindegree)
	sourceweights	weights of the edges into the calling process	int[]	INTEGER(*)
	maxoutdegree	size of destinations and destweights arrays	int	INTEGER
	destinations	processes for which the calling process is a source	int[]	INTEGER(maxoutdegree)
	destweights	weights of the edges out of the calling process	int[]	INTEGER(*)
<hr/>				
)			



Code:

```
1 // graph.c
2 for ( int i=0; i<=1; i++ ) {
3     int neighb_i = proci+i;
4     if (neighb_i<0 || neighb_i>=idim)
5         continue;
6     int j = 1-i;
7     int neighb_j = procj+j;
8     if (neighb_j<0 || neighb_j>=jdim)
9         continue;
10    destinations[ degree++ ] =
11        PROC(neighb_i,neighb_j,idim,jdim);
12 }
13 MPI_Dist_graph_create
14 (comm,
15  /* I specify just one proc: me */ 1,
16  &procno,&degree,destinations,weights,
17  MPI_INFO_NULL,0,
18  &comm2d
19  );
```



Code:

```

1  int indegree,outdegree,
2      weighted;
3  MPI_Dist_graph_neighbors_count
4      (comm2d,
5      &indegree,&outdegree,
6      &weighted);
7  int
8      my_ij[2] = {proci,procj},
9      other_ij[4][2];
10 MPI_Neighbor_allgather
11     ( my_ij,2,MPI_INT,
12     other_ij,2,MPI_INT,
13     comm2d );

```

Output:

```

1  [ 0 = (0,0)] has 2 outbound: 1, 2,
2      0 inbound:
3  [ 1 = (0,1)] has 1 outbound: 3,
4      1 inbound: (0,0)=0
5  [ 2 = (1,0)] has 2 outbound: 3, 4,
6      1 inbound: (0,0)=0
7  [ 3 = (1,1)] has 1 outbound: 5,
8      2 inbound: (0,1)=1 (1,0)=2
9  [ 4 = (2,0)] has 1 outbound: 5,
10     1 inbound: (1,0)=2
11 [ 5 = (2,1)] has 0 outbound:
12     2 inbound: (1,1)=3 (2,0)=4

```

Note that the neighbors are listed in correct order. This need not be the case.



Explicit query of neighbor process ranks.

Code:

```
1  int indegree,outdegree,
2      weighted;
3  MPI_Dist_graph_neighbors_count
4      (comm2d,
5       &indegree,&outdegree,
6       &weighted);
7  int
8      my_ij[2] = {proci,procj},
9      other_ij[4][2];
10 MPI_Neighbor_allgather
11   ( my_ij,2,MPI_INT,
12     other_ij,2,MPI_INT,
13     comm2d );
```

Output:

```
1      0 inbound:
2      1 inbound: 0
3      1 inbound: 0
4      2 inbound: 1 2
5      1 inbound: 2
6      2 inbound: 4 3
```

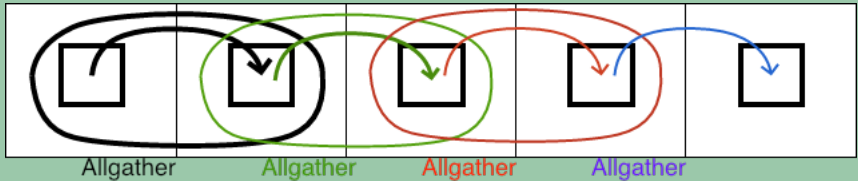


► Earlier rightsend exercise

Revisit exercise 5 and solve it using `MPI_Dist_graph_create`. Use figure 113 for inspiration.

Use a degree value of 1.





Solving the right-send exercise with neighborhood collectives



Two approaches:

1. Declare just one source: the previous process. Do this! Or:
2. Declare two sources: the previous and yourself. In that case bear in mind slide 106.



- Heterogeneous: `MPI_Neighbor_alltoallw`.
- Non-blocking: `MPI_Ineighbor_allgather` and such
- Persistent: `MPI_Neighbor_allgather_init`,
`MPI_Neighbor_allgatherv_init`.



Part IX

Other MPI-4 material



- Error handler `MPI_ERRORS_ABORT`: aborts on the processes in the communicator for which it is specified.
- Error code `MPI_ERR_PROC_ABORTED`: process tried to communicate with a process that has aborted.



`MPI_Info_get` and `MPI_Info_get_valuelen` are not robust with respect to the *null terminator*.

Replace by:

```
1  int MPI_Info_get_string
2      (MPI_Info info, const char *key,
3       int *buflen, char *value, int *flag)
```



Part X

Summary



- Fortran 2008 bindings (MPI-3)
- `MPI_Count` arguments for large buffers (MPI-4)
- Atomic one-sided communication (MPI-3)
- Non-blocking collectives (MPI-3) and persistent collectives (MPI-4)
- Shared memory (MPI-3)
- Graph topologies (MPI-3)
- Partitioned sends (MPI-4)
- Sessions model (MPI-4)



Supplemental material



(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

1. Give the paper to your right neighbor;
2. Accept the paper from your left neighbor.

Including boundary conditions for first and last process, that becomes the following program:

1. If you are not the rightmost student, turn to the right and give the paper to your right neighbor.
2. If you are not the leftmost student, turn to your left and accept the paper from your left neighbor.



Organize your processes in a grid, and make subcommunicators for the rows and columns. For this compute the row and column number of each process.

In the row and column communicator, compute the rank. For instance, on a 2×3 processor grid you should find:

Global ranks:	Ranks in row:	Ranks in column:
0 1 2	0 1 2	0 0 0
3 4 5	0 1 2	1 1 1

Check that the rank in the row communicator is the column number, and the other way around.

Run your code on different number of processes, for instance a number of rows and columns that is a power of 2, or that is a prime number. This is one occasion where you could use `ibrun -np 9`; normally you would *never* put a processor count on `ibrun`.

