Advanced Features of MPI-3 and MPI-4

Victor Eijkhout

PEARC 2023

Materials

Textbooks and repositories:
`https://theartofhpc.com/pcse`

Contact:
eijkhout@tacc.utexas.edu

Justification

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

Overview

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.

MPI headers

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).

Optional error parameter

Old Fortran90 style:

```
call MPI_Init(ierr)
! your code
call MPI_Finalize(ierr)
```

New Fortran2008 style:

```
call MPI_Init()
! your code
call MPI_Finalize()
```

Communicators

Communicators are now derived types:

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

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

Requests

Requests are also derived types
note that ...NULL entities are now objects, not integers

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

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

More

More handles that are now derived types:

```
Type(MPI_Datatype) :: newtype ! F2008
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

Status

Fortran2008: status is a `Type` with fields:

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

Fortran90: status is an array with named indexing

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

Type checking

Type checking catches potential problems:

```
!! typecheckarg.F90
  integer,parameter :: n=2
  Integer,dimension(n) :: source
  call MPI_Init()
  call MPI_Send(source,MPI_INTEGER,n, &
       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'

Type checking does not catch all problems:

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

Buffer/type mismatch is not caught.

Part II

Big data communication

Overview

This section discusses big messages.

Commands learned:

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

The problem with large messages

- There is no problem allocating large buffers:

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

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

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

  because the size argument has to be `int`.

MPI 3 count type

Count type since MPI 3
C:

```
MPI_Count count;
```

Fortran:

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

Big enough for

- `int`;
- `MPI_Aint`, used in one-sided (and therefore big enough for `intptr_t` and `ptrdiff_t`);
- `MPI_Offset`, used in file I/O.

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

MPI 4 large count routines

C: routines with `_c` suffix

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

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

Fortran: polymorphism rules

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

Big count example

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

Same in F08

```
!! pingpongbig.F90
  integer :: power,countbytes
  Integer(KIND=MPI_COUNT_KIND) :: length
  call MPI_Sizeof(length,countbytes,ierr)
  if (procno==0) &
       print *,"Bytes in count:",countbytes
        length = 10**power
        allocate( senddata(length),recvdata(length) )
           call MPI_Send(senddata,length,MPI_DOUBLE_PRECISION, &
                processB,0, comm)
           call MPI_Recv(recvdata,length,MPI_DOUBLE_PRECISION, &
                processB,0, comm,MPI_STATUS_IGNORE)
```

MPI_Send

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)
	)			

MPI 4 large count querying

C:

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

Fortran:

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

Compound messages

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

```
MPI_Datatype blocktype;
MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
MPI_Type_commit(&blocktype);
if (procno==sender) {
  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.

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

Part III

Advanced collectives

Non-blocking 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:

```
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

MPI_Ibcast

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)
	)			

Overlapping collectives and computation

Independent collective and local operations:

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

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

Simultaneous reductions

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

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

which translates to:

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

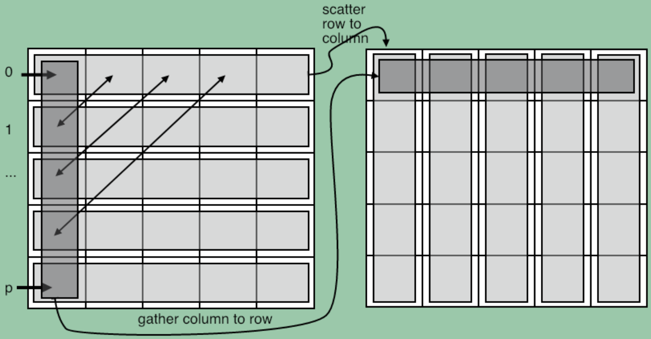
Matching collectives

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

```
if (rank==root)
  MPI_Reduce( &x /* ... */ root,comm );
else
  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.

Transpose as gather/scatter



Every process needs to do a scatter or gather.

Simultaneous collectives

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

```
MPI_Request scatter_requests[nprocs];
for (int iproc=0; iproc<nprocs; iproc++) {
  MPI_Iscatter( regular,1,MPI_DOUBLE,
                     &(transpose[iproc]),1,MPI_DOUBLE,
                     iproc,comm,scatter_requests+iproc);
}
MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE);
```

Persistent collectives

Persistent collectives (MPI-4)

Similar to persistent send/recv:

```
double *buffer;
MPI_Allreduce_init( buffer ...., &request );
for ( ... ) {
  // fill buffer
  MPI_Start( request );
  // possibly other activities
  MPI_Wait( &request );
}
MPI_Request_free( &request );
```

Available for all collectives and neighborhood collectives.

Example

```
// powerpersist1.c
double localnorm,globalnorm=1.;
MPI_Request reduce_request;
MPI_Allreduce_init
  ( &localnorm,&globalnorm,1,MPI_DOUBLE,MPI_SUM,
    comm,MPI_INFO_NULL,&reduce_request);
for (int it=0; ; it++) {
  /*
   * Matrix vector product
   */
  matmult(indata,outdata,buffersize);

// start computing norm of output vector
  localnorm = local_l2_norm(outdata,buffersize);
  double old_globalnorm = globalnorm;
  MPI_Start( &reduce_request );

// end computing norm of output vector
  MPI_Wait( &reduce_request,MPI_STATUS_IGNORE );
  globalnorm = sqrt(globalnorm);
// now ‘globalnorm’ is the L2 norm of ‘outdata’
  scale(outdata,indata,buffersize,1./globalnorm);
}
```

Persistent vs non-blocking

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, still requires `MPI_Request_free`.

Non-blocking barrier

Just what is a barrier?

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

Use case: adaptive refinement

- 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.

Use case: distributed termination detection

- 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

MPI_Ibarrier

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

Step 1

Do sends, post barrier.

```
// ibarrierprobe.c
if (i_do_send) {
  /*
   * Pick a random process to send to,
   * not yourself.
   */
  int receiver = rand()%nprocs;
  MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
}
/*
 * Everyone posts the non-blocking barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);
```

Step 2

Poll for barrier and messages

```
for ( ; ; step++) {
  int barrier_done_flag=0;
  MPI_Test(&barrier_request,&barrier_done_flag,
           MPI_STATUS_IGNORE);
//stop if you're done!
  if (barrier_done_flag) {
    break;
  } else {
// if you're not done with the barrier:
    int flag; MPI_Status status;
    MPI_Iprobe
      ( MPI_ANY_SOURCE,MPI_ANY_TAG,
        comm, &flag, &status );
    if (flag) {
// absorb message!
```

Part IV

Shared memory

Shared memory myths

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.

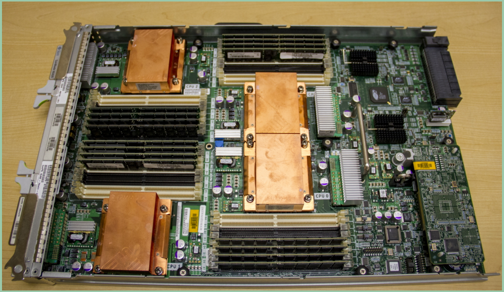
MPI shared memory

- 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.

Shared memory threatments in MPI

- MPI uses optimizations for shared memory: copy instead of socket call
- One-sided offers ‘fake shared memory’: you 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.

Shared memory per cluster node



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

Shared memory interface

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`.

Discover shared memory

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

Code:

```
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type
  (MPI_COMM_WORLD,
   MPI_COMM_TYPE_SHARED,
   procno,info,&sharedcomm);
MPI_Comm_size
  (sharedcomm,&new_nprocs);
MPI_Comm_rank
  (sharedcomm,&new_procno);
```

Output:

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

Exercise 1 (shareddata)

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?

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

Allocate shared window

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.

```c
// sharedbulk.c
MPI_Win node_window;
MPI_Aint window_size; double *window_data;
if (onnode_procid==0)
  window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
  ( window_size,sizeof(double),MPI_INFO_NULL,
    nodecomm,
    &window_data,&node_window);
```

Get pointer to other windows

Use MPI_Win_shared_query:

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

MPI_Win_shared_query

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)
	)			

Allocated memory

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?

Exciting example: bulk data

- Application: ray tracing:
  large read-only data strcture 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.

Split by other hardware types

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

Part V

Atomic operations

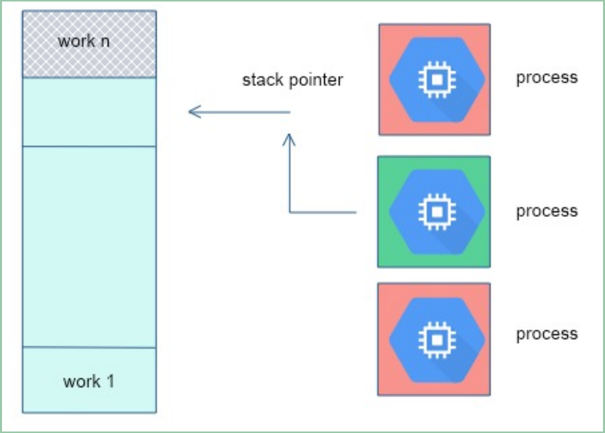
Justification

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 communicatio

- 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.

In a picture



Simplified model

- 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?

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	read I = 0	read I = 0	read I = 0	read I = 0	
local I = 2	local I = 3	local I = 2	local I = 3	local I = 2	
write I = 2			write I = 3	write I = 2	
	write I = 3	write I = 2			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)

Case study in shared memory: 1, wrong

```
// countdownput.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
         counter_process,0,1,MPI_INT,
         the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
  int decrement = -1;
  counter_value += decrement;
  MPI_Put
    ( &counter_value,   1,MPI_INT,
      counter_process,0,1,MPI_INT,
      the_window);
}
MPI_Win_fence(0,the_window);
```

Discussion

- 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

```
MPI_Win_fence(0,the_window);
if (i_am_available) {
  MPI_Get( &counter_value, ... )
  MPI_Win_fence(0,the_window);
  MPI_Put( ... )
}
MPI_Win_fence(0,the_window);
```

?

Case study in shared memory: 2, hm

```
// countdownacc.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
         counter_process,0,1,MPI_INT,
         the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
  int decrement = -1;
  MPI_Accumulate
     ( &decrement,        1,MPI_INT,
       counter_process,0,1,MPI_INT,
       MPI_SUM,
       the_window);
}
MPI_Win_fence(0,the_window);
```

Discussion: need for atomics

- `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.

MPI_Fetch_and_op

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)
	)			

Case study in shared memory: 3, good

```
MPI_Win_fence(0,the_window);
int
  counter_value;
if (i_am_available) {
  int
    decrement = -1;
  total_decrement++;
  MPI_Fetch_and_op
    ( /* operate with data from origin: */   &decrement,
      /* retrieve data from target:     */   &counter_value,
      MPI_INT, counter_process, 0, MPI_SUM,
      the_window);
}
MPI_Win_fence(0,the_window);
if (i_am_available) {
  my_counter_values[n_my_counter_values++] = counter_value;
}
```

Allowable operators. (Hint!)

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

No user-defined operators.

Problem

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.

Passive target epoch

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

No action on the target required!

Exercise 2 (lockfetch)

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:

```
int one=1;
MPI_Fetch_and_op(&one, &readout,
    MPI_INT, repo, zero_disp, MPI_SUM,
    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?

Exercise 3 (lockfetchshared)

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

Partitioned communication (MPI-4)

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

Create partitions

Buffer consists of equal sized partitions:

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

Init calls

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

```
MPI_Psend_init
  (sendbuffer,nparts,SIZE,MPI_DOUBLE,tgt,0,
   comm,MPI_INFO_NULL,&send_request);
MPI_Precv_init
  (recvbuffer,nparts,SIZE,MPI_DOUBLE,src,0,
   comm,MPI_INFO_NULL,&recv_request);
```

Send and recv need both be of partitioned type: can not match partitioned send and non-partitioned recv or vv.

Partitioned send

Sender-side scenario, multiple messages:

```
MPI_Request send_request;
MPI_Psend_init
  (sendbuffer,nparts,SIZE,MPI_DOUBLE,tgt,0,
   comm,MPI_INFO_NULL,&send_request);
for (int it=0; it<ITERATIONS; it++) {
  MPI_Start(&send_request);
  for (int ip=0; ip<nparts; ip++) {
    fill_buffer(sendbuffer,partitions[ip],partitions[ip+1],ip);
    MPI_Pready(ip,send_request);
        }
  MPI_Wait(&send_request,MPI_STATUS_IGNORE);
}
MPI_Request_free(&send_request);
```

Partitioned receive tests

Use

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

to test for arrived partitions.

Partitioned receive

Receiver-side: handle partitions as they come in:

```
double *recvbuffer = (double*)malloc(bufsize*sizeof(double));
MPI_Request recv_request;
MPI_Precv_init
  (recvbuffer,nparts,SIZE,MPI_DOUBLE,src,0,
   comm,MPI_INFO_NULL,&recv_request);
for (int it=0; it<ITERATIONS; it++) {
  MPI_Start(&recv_request); int r=1,flag;
  for (int ip=0; ip<nparts; ip++) // cycle this many times
    for (int ap=0; ap<nparts; ap++) { // check specific part
      MPI_Parrived(recv_request,ap,&flag);
      if (flag) {
        r *= chck_buffer
          (recvbuffer,partitions[ap],partitions[ap+1],ap);
        break; }
    }
  MPI_Wait(&recv_request,MPI_STATUS_IGNORE);
}
MPI_Request_free(&recv_request);
```

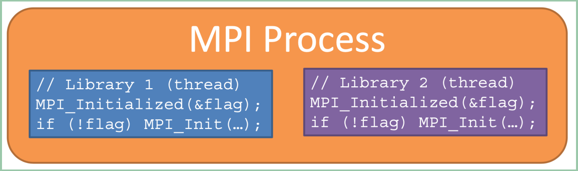
Part VII

Sessions model

Problems with the 'world model'

MPI is started exactly once:

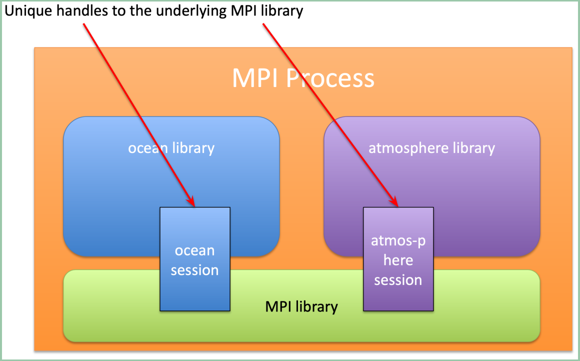
- MPI can not close down and restart.
- Libraries using MPI need to agree on threading and such.



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

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

Sketch of a solution



World and session model

- 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.

Session model

- 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

Session creating

```c
// session.c
MPI_Info session_request_info = MPI_INFO_NULL;
MPI_Info_create(&session_request_info);
char thread_key[] = "mpi_thread_support_level";
MPI_Info_set(session_request_info,
             thread_key,"MPI_THREAD_MULTIPLE");
```

Info object can also be MPI_INFO_NULL,
then

```c
MPI_Session the_session;
MPI_Session_init
  ( session_request_info,MPI_ERRORS_ARE_FATAL,
    &the_session );
MPI_Session_finalize( &the_session );
```

Session: process sets

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

```
int npsets;
MPI_Session_get_num_psets
  ( the_session,MPI_INFO_NULL,&npsets );
if (mainproc)
  printf("Number of process sets: %d\n",npsets);
for (int ipset=0; ipset<npsets; ipset++) {
  int len_pset; char name_pset[MPI_MAX_PSET_NAME_LEN];
  MPI_Session_get_nth_pset
    ( the_session,MPI_INFO_NULL,
      ipset,&len_pset,name_pset );
  if (mainproc)
    printf("Process set %2d: <<%s>>\n",
              ipset,name_pset);
```

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

Session: create communicator

Process set → group → communicator

```
MPI_Group world_group = MPI_GROUP_NULL;
MPI_Comm  world_comm  = MPI_COMM_NULL;
MPI_Group_from_session_pset
  ( the_session,world_name,&world_group );
MPI_Comm_create_from_group
  ( world_group,"victor-code-session.c",
    MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,
    &world_comm );
MPI_Group_free( &world_group );
int procid = -1, nprocs = 0;
MPI_Comm_size(world_comm,&nprocs);
MPI_Comm_rank(world_comm,&procid);
```

Multiple sessions

```
// sessionmulti.c
MPI_Info info1 = MPI_INFO_NULL, info2 = MPI_INFO_NULL;
char thread_key[] = "mpi_thread_support_level";
MPI_Info_create(&info1); MPI_Info_create(&info2);
MPI_Info_set(info1,thread_key,"MPI_THREAD_SINGLE");
MPI_Info_set(info2,thread_key,"MPI_THREAD_MULTIPLE");
MPI_Session session1,session2;
MPI_Session_init( info1,MPI_ERRORS_ARE_FATAL,&session1 );
MPI_Session_init( info2,MPI_ERRORS_ARE_FATAL,&session2 );
```

Practical use: libraries

```
// sessionlib.cxx
class Library {
private:
  MPI_Comm world_comm; MPI_Session session;
public:
  Library() {
    MPI_Info info = MPI_INFO_NULL;
    MPI_Session_init
      ( MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,&session );
    char world_name[] = "mpi://WORLD";
    MPI_Group world_group;
    MPI_Group_from_session_pset
      ( session,world_name,&world_group );
    MPI_Comm_create_from_group
      ( world_group,"world-session",
        MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,
        &world_comm );
    MPI_Group_free( &world_group );
  };
  ~Library() { MPI_Session_finalize(&session); };
```

Practical use: main

```
int main(int argc,char **argv) {

  Library lib1,lib2;
  MPI_Init(0,0);
  MPI_Comm world = MPI_COMM_WORLD;
  int procno,nprocs;
  MPI_Comm_rank(world,&procno);
  MPI_Comm_size(world,&nprocs);
  auto sum1 = lib1.compute(procno);
  auto sum2 = lib2.compute(procno+1);
```

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

Part VIII

Process topologies

Overview

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

Process topologies

- 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

Process reordering

- 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

MPI-1 topology

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

MPI-3 topology

- 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.

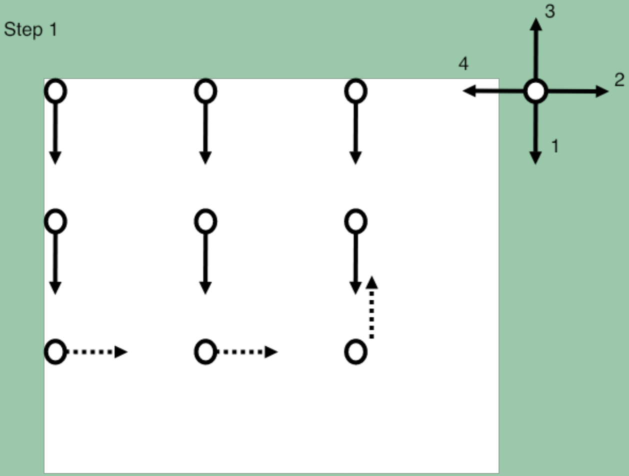
Graph topologies

Example: 5-point stencil

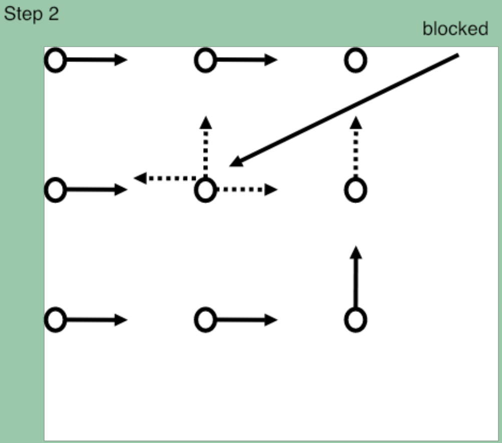
Neighbor exchange, spelled out:

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

Step 1



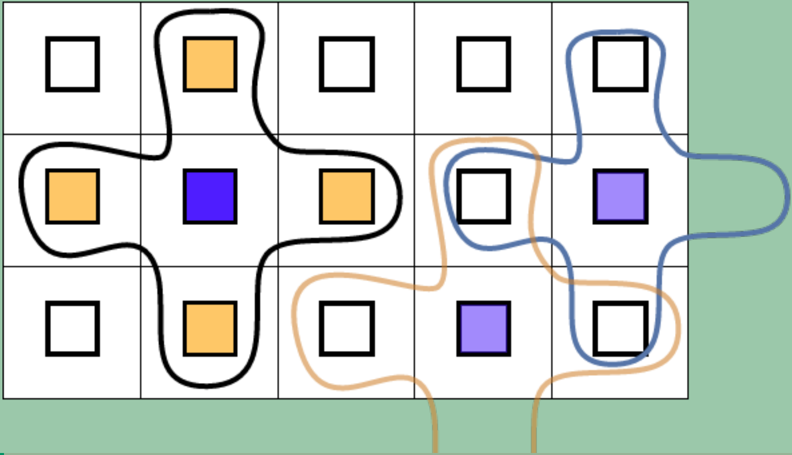
Step 2



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

Neighborhood collective

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



Why neighborhood collectives?

- 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.

Create graph topology

```
int MPI_Dist_graph_create
  (MPI_Comm comm_old, int nsources, const int sources[],
   const int degrees[], const int destinations[],
   const int weights[], MPI_Info info, int reorder,
   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

Neighborhood collectives

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

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

Neighbor querying

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.

MPI_Dist_graph_neighbors_count

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
	)			

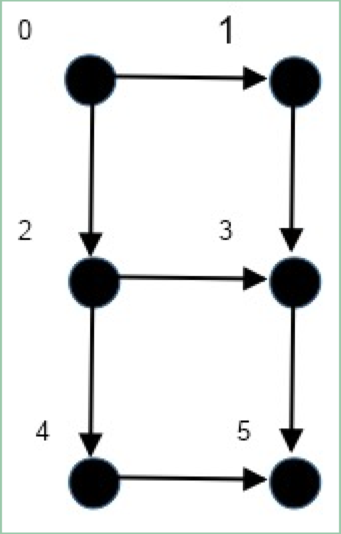
MPI_Dist_graph_neighbors

Name	Param name	Explanation	C type	F type
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(*)
	)			

Example: Systolic graph

Code:

```
// graph.c
for ( int i=0; i<=1; i++ ) {
  int neighb_i = proci+i;
  if (neighb_i<0 || neighb_i>=idim)
    continue;
  int j = 1-i;
  int neighb_j = procj+j;
  if (neighb_j<0 || neighb_j>=jdim)
        continue;
  destinations[ degree++ ] =
        PROC(neighb_i,neighb_j,idim,jdim);
}
MPI_Dist_graph_create
  (comm,
   /* I specify just one proc: me */ 1,
   &procno,&degree,destinations,weights,
   MPI_INFO_NULL,0,
   &comm2d
   );
```



Output

Code:

```
int indegree,outdegree,
  weighted;
MPI_Dist_graph_neighbors_count
  (comm2d,
   &indegree,&outdegree,
   &weighted);
int
  my_ij[2] = {proci,procj},
  other_ij[4][2];
MPI_Neighbor_allgather
  ( my_ij,2,MPI_INT,
    other_ij,2,MPI_INT,
    comm2d );
```

Output:

```
[ 0 = (0,0)] has  2 outbound: 1, 2,
   0 inbound:
[ 1 = (0,1)] has  1 outbound: 3,
   1 inbound: (0,0)=0
[ 2 = (1,0)] has  2 outbound: 3, 4,
   1 inbound: (0,0)=0
[ 3 = (1,1)] has  1 outbound: 5,
   2 inbound: (0,1)=1 (1,0)=2
[ 4 = (2,0)] has  1 outbound: 5,
   1 inbound: (1,0)=2
[ 5 = (2,1)] has  0 outbound:
   2 inbound: (1,1)=3 (2,0)=4
```

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

Query

Explicit query of neighbor process ranks.

Code:

```
int indegree,outdegree,
  weighted;
MPI_Dist_graph_neighbors_count
  (comm2d,
   &indegree,&outdegree,
   &weighted);
int
  my_ij[2] = {proci,procj},
  other_ij[4][2];
MPI_Neighbor_allgather
  ( my_ij,2,MPI_INT,
    other_ij,2,MPI_INT,
    comm2d );
```

Output:

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

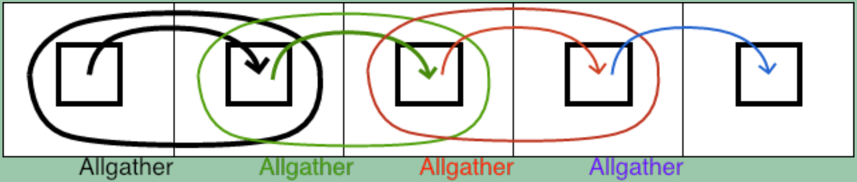
Exercise 4 (rightgraph)

▸ 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.

Inspiring picture for the previous exercise



Solving the right-send exercise with neighborhood collectives

Hints for the previous exercise

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.

More graph collectives

- 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

Better aborts

- 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.

Error as C-string

MPI_Info_get and MPI_Info_get_valuelen are not robust with respect to the *null terminator*.
Replace by:

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

Part X

Summary

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

Exercise 5 (`serialsend`)

(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.

Exercise 6 (procgrid)

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 colum:
  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`.