Message Passing Interface

James Davenport

University of Bath

16 September 2019

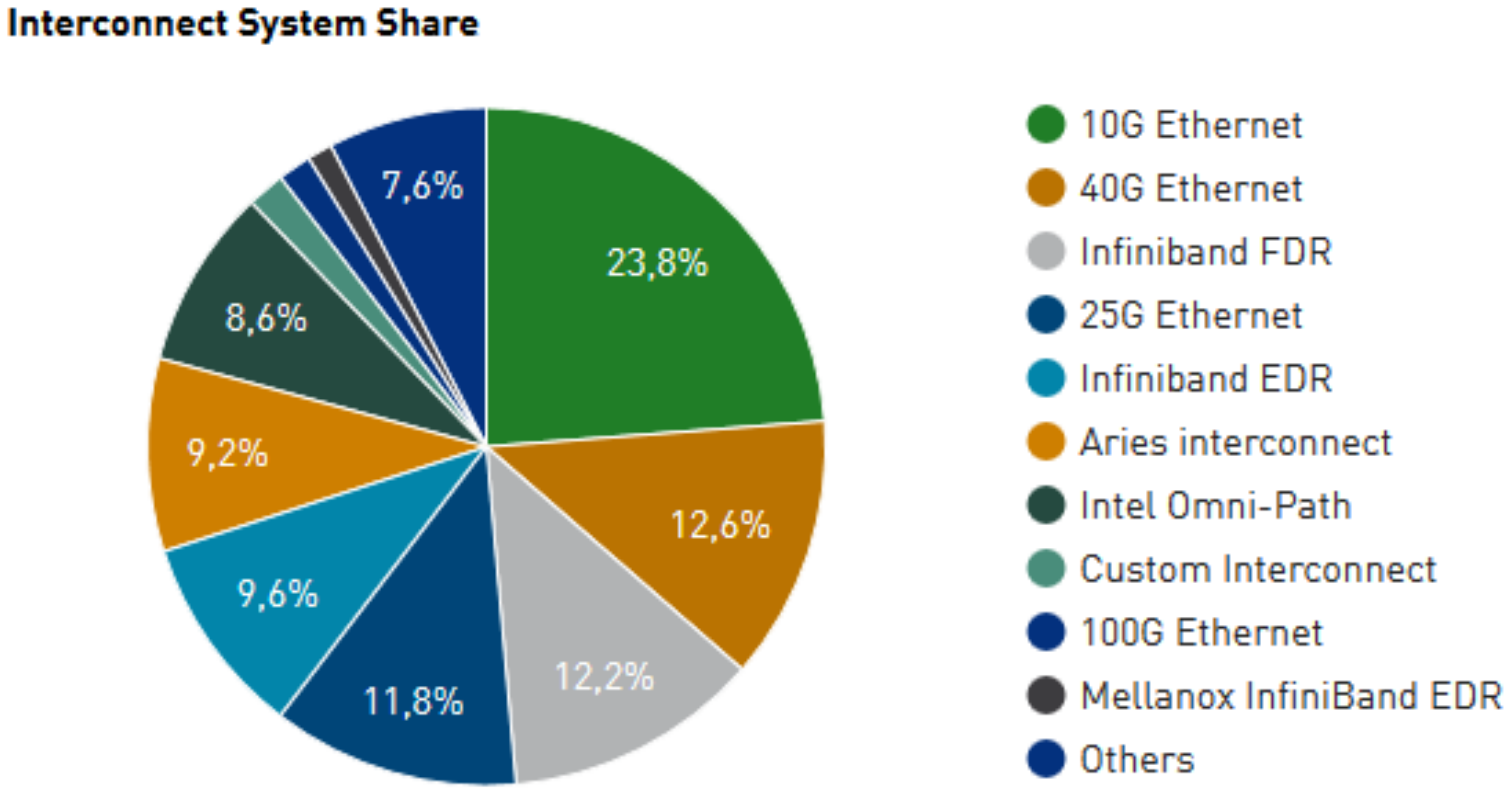
Modern HPC and Distributed Memory

An HPC isn't a faster computer, it's many computers (nodes).

- Each node (which may have multiple CPUs (typically 1 or 2), each with multiple cores) has its own memory.
- In a distributed memory machine, each node holds all variables in local memory – local addressing.
- Hence, work shared across processes will require communication.
- Message passing is the context in which this communication takes place.
- So how are the nodes connected?

Interconnects Top 500 November 2018 [All19]

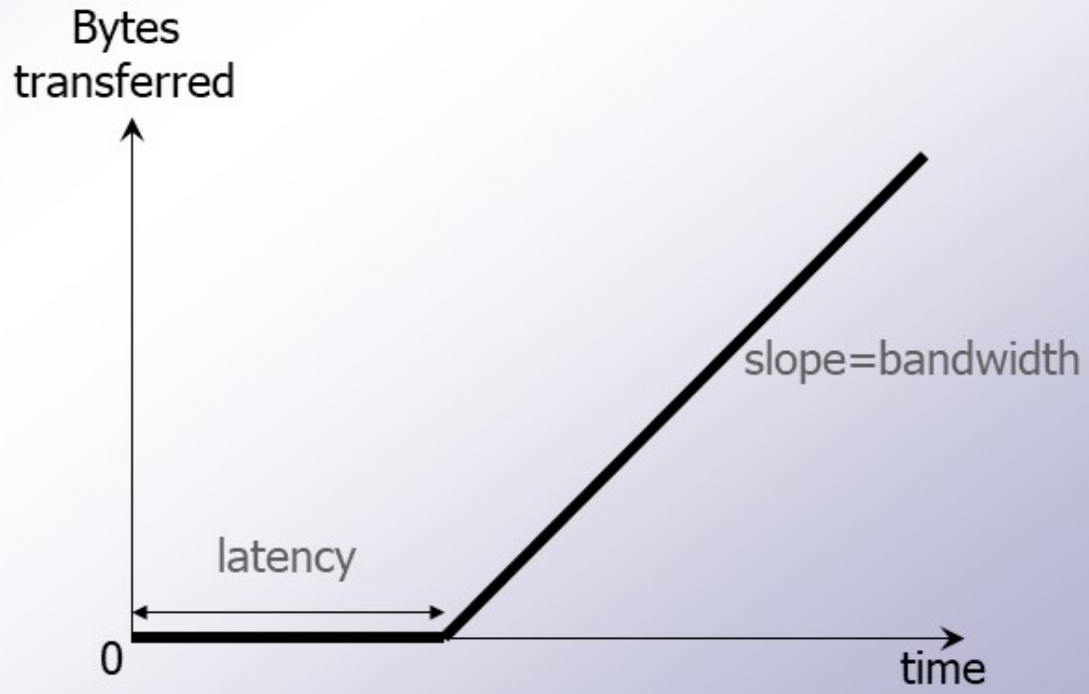
Infiniband: FDR = 14G, EDR=25G, but most people use four links in parallel, so 56G or 100G.
Adapter latencies 0.7μs (FDR) or 0.5μs (EDR)



Characterising Performance

Transferring data

- Latency - start-up time
- Bandwidth - (asymptotic) transfer rate (Bytes/second)



Parallel Programming with MPI

19

Top 3 from 500 June 2020

Computer	Cores	$R_{\max}$ TFlop/sec	R_{peak} TFlop/sec	Power kW
Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438

28MW is roughly 60 wind turbines; $7299072 = 48 \times 2^9 \times 3^3 \times 11$

Latency/bandwidth examples

1. Tofu D: 1 `double` is 490ns, 1000 `double` is 790ns (based on 38.1 GB/sec [AKO+18])
2. 4× EDR: latency 500ns, 80ps/byte, so 1 `double` is 500.64 ns; 1000 `double` is 1140 ns.
3. 100GHz Ethernet: latency 1600ns (or more), 80ps/byte, so 1 `double` is 1600.64 ns; 1000 `double` is 2240 ns.
4. 10GHz Ethernet: latency 5000ns (or more), 800ps/byte, so 1 `double` is 5006.4 ns; 1000 `double` is 11400 ns.

Note that these are "best case" figures, assuming the nodes are neighbours in whatever physical topology underpins the networking. If not, latency will certainly increase, and bandwidth may decrease depending on congestion.
The fancier interconnects (Fujitsu's Tofu; Cray's Aries) have fancier physical topology.

Message Passing Strategy

- Problem must be decomposed (By domain or function: if your problem is PDE, it's generally domain decomposition)
- Data distribution must be controlled.
- Local addressing implies that each core knows nothing about data on other cores. Information regarding local memory of other cores must be obtained via message passing.
- Accessing 'remote' data takes much longer than accessing local data. Hence, a major objective of HPC is to distribute data so as to minimise communication.
  - Bandwidth (amount of data being transferred)
  - Latency (time dependence on data being transferring)

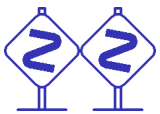
Message Passing Basics

- Fundamental requirements of message passing:
  - to send data to another process
  - to receive data from another process
  - to synchronise processes
- There are a great variety of ways to do this, and substantial flexibility to control data movement.
- MPI (Message Passing Interface) [Mes15]
  - The de facto standard for writing message-passing codes.
  - Development involved virtually every parallel computing vendor.
  - The library to use if you are starting message passing
- Note that MPI and other methods (OpenMP etc.) are not mutually exclusive [AAG+15], but MPI sometimes handles multiple nodes better than PGAS (coarrays in Fortran) [Ash14]; [BBH+19] claims the opposite for UPC++.

Why is MPI the library of choice?

- Portable code
  - Implementations exist for most parallel platforms.
  - Free, portable, downloadable versions available.
- Optimal performance
  - Considerable effort has been put into optimising the performance of the library and tuning it to specific hardware platforms and interconnects.
  - This development is ongoing.
- The standard itself is also continually being refined and updated (2018~~9~~ is in draft): 3.1 isn't fully 64-bit clean.
- 3.1 [Mes15], the current version, is not fully 64-bit native.

There may be multiple implementations of MPI available, and, while functionally equivalent, performance varies unpredictably (at least I can't predict!)



Do not "mix and match"

MPI paradigm

- The same program is launched on all the processors

SPMD Single Program, Multiple Data

- But the program can do (very) different things depending on which processor it's running, so it is more general than SIMD
- For example, in weather forecasting, some processors might be modelling the ocean, and others the atmosphere
- well-suited to multiple independent processors

Timings

```
if condition
        { code A }    // takes time t_A
  else  { code B }    // takes time t_B
```

One instance $t \in \{t_A, t_B\}$

SIMD $t = t_A + t_B$

! A processors idle during t_B, *vice versa*

SPMD $t = \max(t_A, t_B)$

So try to match t_A, t_B

Very conditional code and SIMD don't go well together

MPI names

- All MPI names have an `MPI_` prefix
- In Fortran, all characters in the name are capitals (although the language is not case-sensitive).
- In C/C++, which are case-sensitive, defined constants have all capital letters and defined types and functions have one capital letter after the prefix, with the rest being lower case.
- The user program must not declare variables or functions with names beginning with the prefix `MPI_` or `PMPI_`, which is used by the profiling interface

Errors in MPI operations

- By default, if an MPI call detects an error then the program will abort.
- Although rarely done in practice, it is possible to change this behaviour so that an MPI call just returns an error code, which the programmer must then check and act upon in an appropriate way.

Many libraries and applications do not check for error codes, and assume the default.

The MPI header file / module

```
/* In C or C++, include the header file. */
#include <mpi.h>

! In Fortran, always use the MPI module if one is
!available on your system. An MPI-2 compliant
! implementation should provide one.
USE MPI
! Otherwise, include the FORTRAN header file.
include 'mpif.h'
```

The MPI Environment

- Exactly how a multiple processor job is initialized is environment dependent (`mpirun` inside `sbatch`).
- MPI provides two functions interfacing with start-up and shutdown.
  `MPI_Init`
  `MPI_Finalize`
- Note that these calls do not start up or shutdown the processes themselves but the MPI environment which allows them to communicate. All the processes start when the job is launched.

Declarations

```
/* C and C++ startup/shutdown routines */
int MPI_Init(int *argc, char ***argv);
/* Note the extra * here */
int MPI_Finalize();
```

```
! FORTRAN startup and shutdown
SUBROUTINE MPI_INIT(IERROR)
INTEGER :: IERROR

SUBROUTINE MPI_FINALIZE(IERROR)
INTEGER :: IERROR
```

Fortan routines must always have `ERROR`, C doesn't need it if you're not checking.

Rank and Size

- These definitions are essential to any MPI code as the mechanism by which the programmer gets different processes to perform different tasks or work on different data.
- The size is the number of processes. The number of processes with which to run a job is normally specified at runtime (`sbatch` or whatever).
- The rank is a unique integer associated with each process:

$$0 \leq \text{rank} \leq \text{size} - 1$$

- Strictly speaking, these definitions of rank and size should say the rank within and size of the group of processes associated with a given communicator.

Communicators

- A communicator is an MPI variable which must be associated with a group of processes for communication to take place within that group. Of type INTEGER in Fortran and `MPI_Comm` in C/C++.
- There are two predefined communicators:
- `MPI_COMM_WORLD`

  Associated with all processes
- `MPI_COMM_SELF`

  Associated with an individual process only; rarely useful

Finding out the size/rank

- The function `MPI_Comm_size` reports the size.
  - The first argument is the communicator.
- In C/C++,

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- And in Fortran,

```
SUBROUTINE MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER :: COMM, SIZE, IERROR
```

- The function `MPI_Comm_rank` is used to establish the rank of a process
  - an integer in the range [0,size−1]

```
 /* Could someone please tell me who I am? */
int MPI_Comm_rank(MPI_Comm comm, int *rank);

SUBROUTINE MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER :: COMM, RANK, IERROR
```

Basic MPI code checklist

1. Include the appropriate header file or module
2. Initialise the MPI environment using `MPI_Init`
3. Each MPI process must find out the total number of processes using `MPI_Comm_size`
4. Each MPI process must find out its own unique rank using `MPI_Comm_rank`

* Now we can do the actual work

5. Shutdown the MPI environment using `MPI_Finalize`

Basic MPI code C template

```
#include <mpi.h>

int main(int argc, char ** argv){
  int size, rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* the body of the code goes here */

  MPI_Finalize();
}
```

Basic MPI code Fortran template

```
PROGRAM basic_MPI_template
  USE MPI
  IMPLICIT NONE
  INTEGER :: ierr, rank, size

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

! the body of the code goes here

  CALL MPI_FINALIZE(ierr)
END PROGRAM basic_MPI_template
```

Hello, world (abbreviated) I

```
# include <cstdlib>
# include <ctime>
# include <iomanip>
# include <iostream>
# include <mpi.h>
int main ( int argc, char *argv[] );
{ int id, ierr, p;
  double wtime;
  ierr = MPI_Init ( &argc, &argv );
  if ( ierr != 0 )
  { cout << "HELLO_MPI - Fatal error!\n";
    cout << "  MPI_Init returned nonzero ierr.\n";
    exit ( 1 ); }
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &p );
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &id );
```

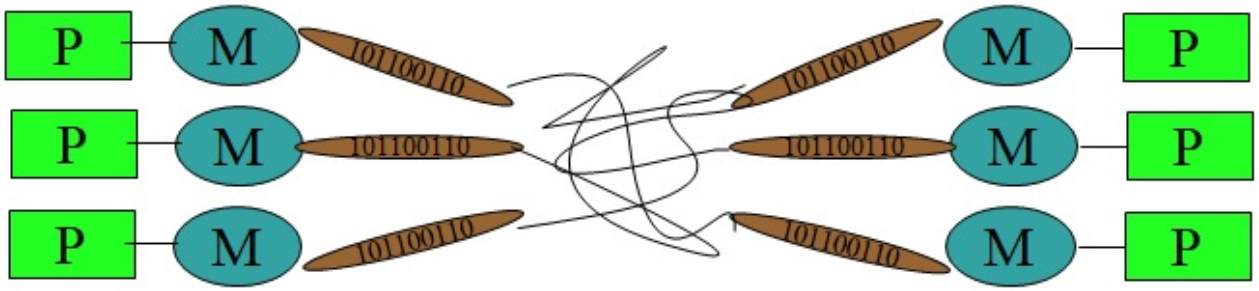
Hello, world (abbreviated) II

```
  if ( id == 0 )
  { cout << "P" << id << ":  HELLO_MPI - Master process:\n"
    cout << "P" << id << ":    The number of processes is "
  if ( id == 0 )
  { wtime = MPI_Wtime ( ); }
  cout << "P" << id << ":    'Hello, world!'\n";
  if ( id == 0 )
  { wtime = MPI_Wtime ( ) - wtime;
    cout << "P" << id << ":    Elapsed wall clock time = "
  MPI_Finalize ( );
  if ( id == 0 )
  { cout << "P" << id << ":  HELLO_MPI:\n";
    cout << "P" << id << ":    Normal end of execution.\n";
  return 0;
}
```

Point-to-point communication

- There are two modes of communication used in MPI
  - One process sending data to one other process is called *point-to-point communication*.

  - Communication involving a group of processes is called *collective*. We will consider this later.



Parallel Programming with MPI

Two-sided communication

The standard type of point-to-point communication in MPI is *two-sided communication*.

- This means that both the sender and receiver of the data need to call MPI routines, a send call and a receive call respectively, for the data to be transferred.
- For every send call, there must be a matching receive call.
- The basic calls for doing this are `MPI_Send` and `MPI_Recv`.
- An MPI message can be many items, but all of same data type.

MPI P2P Analogy

There's a large office block. Every worker has her own office, which she can't leave. There are also some (telepathic) messengers rushing round the building. Messengers never disturb workers.

- To `Send` a message, a worker writes an envelope (who to, data type and subject — `tag` in MPI-speak), and stands at the door of her office, waiting for a messenger to come and take the envelope.

A messenger doesn't have to take a message if there is no receiver waiting for it.

- To `Recv` a message, a worker stands at the door of her office, waiting for a messenger to come and deliver the sort of message she is expecting.
- She can say "from any" etc.
- Workers standing at the door don't do any useful work!
- Workers might stand at the door until they starve to death. [deadlock]

An MPI message

In MPI, a message consists of:

- a data buffer
- ! of a certain size — must specify
- a data type
- a sender (source)
- a receiver (destination)
- a tag

This information must be given to the corresponding send and receive calls.

MPI_Send

The basic MPI routine for sending a message is MPI_Send.

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm);
```

```
SUBROUTINE MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG,
        COMM, IERROR)
<type> :: BUF(*)
INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

MPI_Recv

The basic MPI routine for receiving a message is MPI_Recv.

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status);
```

```
SUBROUTINE MPI_RECV(BUF, COUNT, DATATYPE, SOURCE,
       TAG, COMM, STATUS, IERROR)
<type> :: BUF(*)
INTEGER :: COUNT, DATATYPE, SOURCE, TAG, COMM
INTEGER :: STATUS(MPI_STATUS_SIZE), IERROR
```

MPI Message — C example

```
int rank;
MPI_Status status;
float a[10], b[10];
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
if (rank == 0){
MPI_Send(a, 10, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1){
MPI_Recv(b, 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
}
...
```

You can use `MPI_STATUS_IGNORE` for `status` (more efficient, *if* that's what you want).

MPI Message — Fortran example

```
INTEGER :: rank, ierr
INTEGER :: status(MPI_STATUS_SIZE)
REAL, DIMENSION(10) :: a, b
...
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
...
IF (rank .EQ. 0) THEN
CALL MPI_SEND(a(1), 10, MPI_REAL, 1, 0, MPI_COMM_WORLD, &
        ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(b(1), 10, MPI_REAL, 0, 0, MPI_COMM_WORLD, &
        status, ierr)
END IF
...
```

You can use `MPI_STATUS_IGNORE` for `status` (more efficient, *if* that's what you want).

The data component

- The data buffer, `buf`, tells MPI where to find the first item of data to be sent and where to start writing the received data, i.e. the appropriate variable name.
- This argument is passed by reference in C so must be a pointer to the data.
- `MPI_Send` sends `count` data elements starting at `buf`.
- `count` may be zero, in which case no data is sent.
- `MPI_Recv` receives up to count data elements and places them at buf.
- The receive data buffer must be at least big enough to hold all the incoming data
- The length of a received message can be found from `status` with a call to `MPI_Get_count`.

The source and dest arguments

- In a call to `MPI_Send`, the `dest` argument is the rank of the process to which the data is to be transmitted.
- Similarly, the `source` argument in the call to `MPI_Recv` is the rank of the process from which data is to be received.
- receivers can use `MPI_ANY_SOURCE`
- `status.MPI_SOURCE` is the actual source, and `status.MPI_TAG` the actual tag.

Datatypes in MPI

- As MPI does not require communicating processes to use the same representation of a datatype, it needs to keep track of possible datatypes. This facilitates:
  - parallel computations in heterogeneous environments
  - porting of parallel programs between machines using different representations of basic datatypes.
- MPI requires that all basic datatypes in FORTRAN and C have a corresponding MPI datatype.
- The programmer can construct data types for `struct`/`type` etc.

C MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	
MPI_PACKED	

Note that whether `char` is signed is implementation-dependent for C/C++, not for MPI.

Fortran MPI Datatypes

MPI datatype	FORTRAN datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

In a FORTRAN program, there might be complications if your program uses variables of non-standard size: see `MPI_TYPE_CREATE_F90_REAL`

MPI_BYTE and MPI_PACKED

- MPI_BYTE and MPI_PACKED are the only elementary datatypes common to Fortran and C.
- A value of type MPI_BYTE consists of a byte (i.e. 8 bits).
- A byte is NOT interpreted and is different from a character.
- Different machines may have different representations for characters, or may use more than one byte to represent characters (UTF-8; Chinese etc.)

Datatypes convert representations, not types

- The MPI datatype specified in the send and receive calls must be the same.
- MPI Datatypes selected using the `MPI_TYPE_CREATE_F90_*` calls must have identical p and r values —- it is not enough that the selected variable has the same KIND value.
- MPI communication never entails type conversions, e.g. from INTEGER to REAL.
- If you need to convert the data, you can always do so before sending it.
- MPI communication is guaranteed to handle representation conversions in a (machine) heterogeneous environment correctly.

The tag and communicator

- The message tag can be used to distinguish various message types (like sorting interesting mail from junk).
- The tag is an integer in the range 0,...,UB, where the value of UB can be found by querying the predefined constant `MPI_TAG_UB`.
- The standard states that UB must be at least 32,767.
- The communicator is a handle to the group of processes involved in the communication. E.g. `MPI_COMM_WORLD`

Message matching rules

- When a message posted by a send has been collected by a receive, the message is said to have completed.
- The entire envelope (`dest`/`source`, `datatype`, `tag` and `communicator`) must match between the send and receive for the message to complete.
- The `count` and the data buffer, `buf`, are allowed to differ

Wildcards in receives

- If a message from any source is acceptable to a receiver, the wildcard source `MPI_ANY_SOURCE` can be used in a call to `MPI_Recv`.
- Similarly, the receive can specify the wildcard tag `MPI_ANY_TAG` to match any tag.
- Although sometimes very useful, they can lead to mistakes — the programmer needs to consider whether the receive could potentially make any undesired matches with send calls.
- If a wildcard is used for the source or tag argument, their actual values can be found from the `status` argument.

Blocking communication

- `MPI_Send` and `MPI_Recv` are blocking calls. This means that
- `MPI_Send` does not return until the data in the send buffer (i.e. the variable in the user program) can be safely changed.

This does not necessarily mean that it's arrived at its destination. It may be in an internal system buffer used by MPI (especially with offload adaptors).

- `MPI_Recv` does not return until the receive buffer (i.e. the variable in the user program) contains all the requested data.

i.e. the complete message that was sent, which may have fewer items than the `count` value the receiver specified.

Deadlocks, a new type of bug

- When a process makes a call to `MPI_Recv`, it will wait patiently until a matching send is posted.
- If the matching send is never posted, the receive will wait forever

* or, in practice, until the system crashes or some time-limit on the job is exceeded.

Hence advice to use a 1-minute job limit when debugging this!

- This introduces a new type of bug that the programmer needs to be aware of.

Deadlocks!

Did your program work?

My program's kernel was

```
MPI_Send(send,strlen(send)+1,  //+1 to send the terminator
         MPI_CHAR,1-id,  // 0 talks to 1, 1 talks to 0
         0,MPI_COMM_WORLD);
MPI_Recv(receive,20,MPI_CHAR,1-id,0,MPI_COMM_WORLD,&status)
MPI_Get_count(&status,MPI_CHAR,&receive_len);
```

It worked at Cambridge, but just hung at Bath.
0 and 1 are standing at the doors of their offices, waiting for messengers to take envelopes.

If a messenger decides to take 0's envelope for 1

Then 0 will wait for a message from 1, which happens

And when 1's message is taken she will receive 0's message

But The messengers could sit in their room, as there are no receives pending

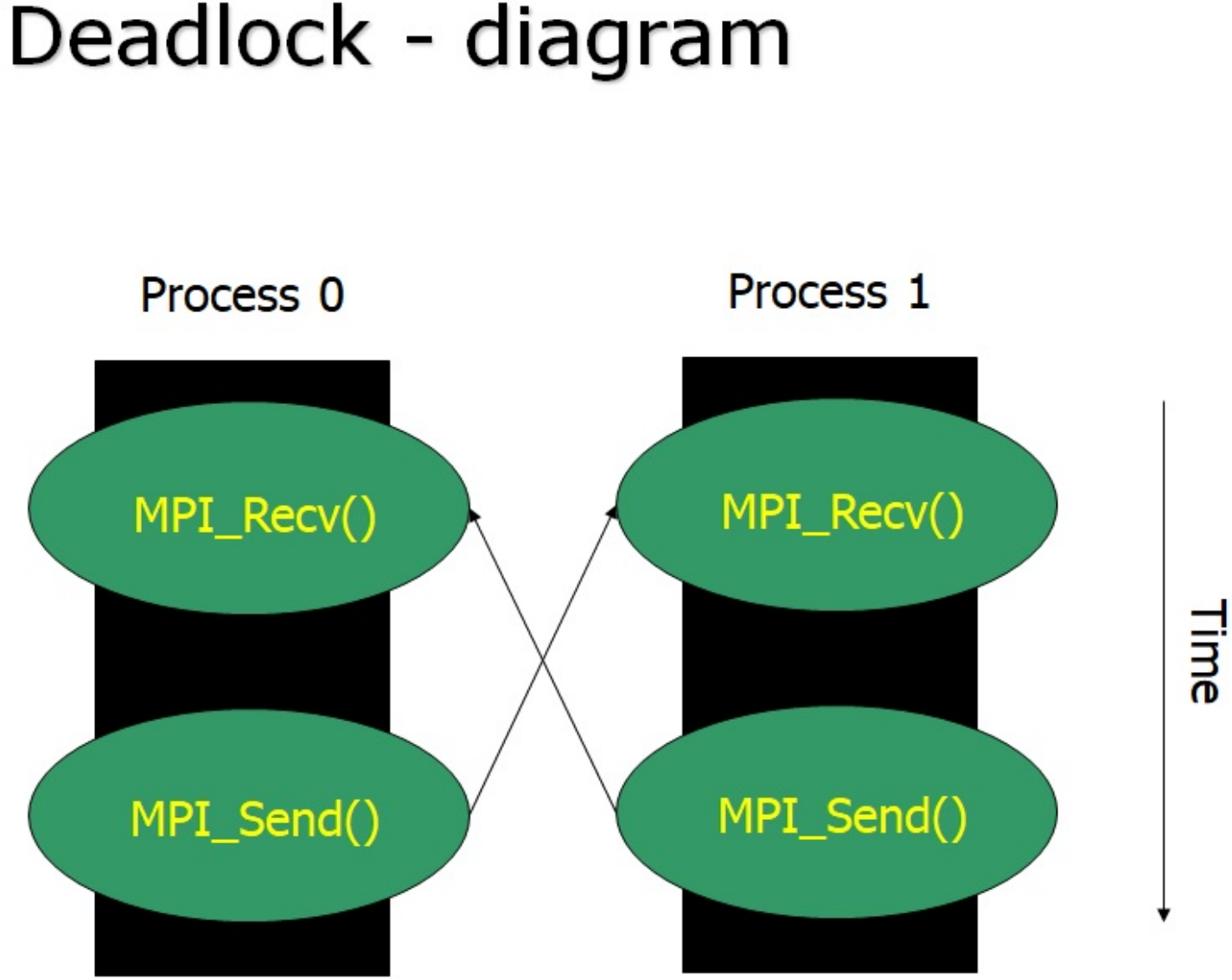
Depends on the MPI implementation (and length of messages . . .)

Guaranteed deadlock example (C++)

```
#include <mpi.h>
extern int rank, size;
int talk_to_neighbour(char *sdat, char *rdat, int n){
  int talk_to;
  MPI_Status status;

  talk_to = (rank%2) ? rank-1 : rank+1;
  if (talk_to < size) {
    MPI_Recv(rdat, n, MPI_CHAR, talk_to, 0,
        MPI_COMM_WORLD, &status);
    cout<<"Proc "<<rank<<" heard"<<rdat<<" from "<<talk_to;
    MPI_Send(sdat, n, MPI_CHAR, talk_to, 0,
        MPI_COMM_WORLD);
  }
  return;
}
```

Guaranteed deadlock example (Fortran)

```
SUBROUTINE talk_to_neighbour(rank, size, sdat, rdat, n)
USE MPI
IMPLICIT NONE
INTEGER, INTENT(IN) :: rank, size, n
CHARACTER(LEN=*), INTENT(IN) :: sdat, rdat
INTEGER :: talk_to, ierr
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
IF (MOD(rank,2) == 0) THEN
   talk_to = rank+1
ELSE
   talk_to = rank-1
END IF
IF (talk_to < size) THEN
   CALL MPI_RECV(rdat, n, MPI_CHARACTER, talk_to, 0, &
        MPI_COMM_WORLD, status, ierr)
   WRITE(*,*) 'Proc ', rank,' heard ', rdat, ' from ', talk_to
   CALL MPI_SEND(sdat, n, MPI_CHARACTER, talk_to, 0, &
       MPI_COMM_WORLD, ierr)
END IF
RETURN
```


Deadlock - diagram
Process 0
Process 1
MPI_Recv()
MPI_Send()
MPI_Recv()
MPI_Send()
Time

Deadlock avoidance example (C++)

```
if (talk_to < size) {
    if (rank%2 == 0){
        MPI_Recv(rdat, n, MPI_CHAR, talk_to, 0,
            MPI_COMM_WORLD, &status);
        MPI_Send(sdat, n, MPI_CHAR, talk_to, 1,
            MPI_COMM_WORLD);
    }
    else {
        MPI_Send(sdat, n, MPI_CHAR, talk_to, 0,
           MPI_COMM_WORLD);
        MPI_Recv(rdat, n, MPI_CHAR, talk_to, 1,
            MPI_COMM_WORLD, &status);
    }
    cout<<"Proc "<<rank<<" heard"<<rdat<<" from "<<talk_to;
}
...
```

Deadlock avoidance example (Fortran)

```
...
IF (talk_to < size) THEN
    IF (MOD(rank,2) == 0) THEN
        CALL MPI_RECV(rdat, n, MPI_CHARACTER, talk_to, &
            0, MPI_COMM_WORLD, status, ierr)
        CALL MPI_SEND(sdat, n, MPI_CHARACTER, talk_to, &
            1, MPI_COMM_WORLD, ierr)
    ELSE
        CALL MPI_SEND(sdat, n, MPI_CHARACTER, talk_to, &
            0, MPI_COMM_WORLD, ierr)
        CALL MPI_RECV(rdat, n, MPI_CHARACTER, talk_to, &
            1, MPI_COMM_WORLD, status, ierr)
    END IF
    WRITE(*,*) 'Proc ', rank,' heard ', rdat,' from ', tall
END IF
...
```

Combined send and receive

- In this example, `MPI_Send` and `MPI_Recv` could be carefully ordered to avoid deadlocks. This can be difficult.
- MPI provides a very useful combined send and receive function, `MPI_Sendrecv`, which is "guaranteed not to deadlock".

If you read the spec. §3.10 naïvely. JHD isn't convinced (for $N > 2$). See [Squ09] for an example (using multiple tags) that can deadlock with $N = 2$.

- This routine sends a message and posts a receive, then blocks until the send data buffer is free and the receive data buffer has received its data.
- The analogy is the office worker with an envelope to send in one hand and a free hand ready to receive: these must both happen, but in either order.
- Other ways to avoid deadlocks include using buffered sends and non-blocking communication, both described later

Different types of blocking send

- Standard send: `MPI_Send` may or may not use a system buffer according to implementation and message length
- Buffered send: `MPI_Bsend` message is buffered using application buffer space, supplied by the user using `MPI_Buffer_attach`. May complete before matching receive is posted
- Synchronous send: `MPI_Ssend` won't complete until a matching receive is posted and the send buffer can be re-used. Completion implies that the receive has started.
- Ready send: `MPI_Rsend` should only be used if the matching receive has already been posted. The programmer needs to be certain that this is the case. Rarely useful. Always correct to replace with a standard send

Non-overtaking but (potentially) unfair

- Messages are dealt with in order but not necessarily fairly.
- Non-overtaking: If a sender posts two messages to the same receiver and a receive operation matches both messages, the message posted first will be chosen.
- Unfair: No matter how long a send has been pending, it can always be overtaken by a message sent from another process.

Blocking and Non-blocking Calls

So far we have only dealt with *blocking* MPI.

Blocking return from the call indicates that resources (primarily, the variables containing the data being sent/received) can safely be re-used.

Non-blocking the call may return before the operation completes, and before the user can safely re-use the resources specified in the call.

Note that non-blocking doesn't speed up the message, rather it just lets one use the latency/overheads

Non-blocking communications

- Each type of send and receive has a non-blocking counterpart:
  - standard: `MPI_Send` ⇒ `MPI_Isend`
  - buffered: `MPI_Bsend` ⇒ `MPI_Ibsend`
  - synchronous: `MPI_Ssend` ⇒ `MPI_Issend`
  - ready: `MPI_Rsend` ⇒ `MPI_Irsend`
  - receive: `MPI_Recv` ⇒ `MPI_Irecv`
- The additional 'I' in the name stands for 'immediate' (as in immediate return).
- These calls may return before the operation has completed. You cannot safely reuse resources (such as the data buffer) until you know that it has completed.
- Need to test for completion using, for example, `MPI_WAIT` or `MPI_TEST`.
- Any type of send routine can be paired with any type of receive routine

- but mixing may cause programmer confusion!

Analogy

We likened `MPI_Send` to an office worker having to stand at the door waiting for a messenger to collect the envelope. By analogy, `MPI_Isend` is rather like having an out-tray: the worker puts the envelope in the out tray and carries on working.
Similarly, `MPI_Irecv` is rather like having an in-tray: the worker puts a post-it saying "happy to receive messages about X from Y", and then the messenger can leave an envelope in the tray.
Unsolicited messages are not delivered.

Bindings of MPI_Isend/MPI_Irecv

We need to be able to refer to a call that's not completed

- MPI_Isend is identical to MPI_Send, except for one additional argument, request.
- Same for the variants
- MPI_Irecv does not have the status argument that MPI_recv has but does have a new argument, request.
- In both cases, the request argument returns a handle to the MPI_Isend/MPI_Irecv call and so provides a way to test whether that call has completed
- Is of type MPI_Request in C/C++ and INTEGER in Fortran.

Waiting for a call to complete

- `MPI_WAIT` – waits until the call has completed.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

```
SUBROUTINE MPI_WAIT(request, status, ierror)
INTEGER :: request
INTEGER :: status(MPI_STATUS_SIZE), ierror
```

- A non-blocking send immediately followed by `MPI_Wait` is functionally equivalent to a blocking send: it's the ability to do things between send and wait that's the difference with non-blocking

Testing for completion of a call

- `MPI_TEST` — tests for completion of a call and returns straight away.

```
int MPI_Test(MPI_Request *request, int *flag,
        MPI_Status *status);
```

```
SUBROUTINE MPI_TEST(request, flag, status, ierror)
INTEGER :: request
LOGICAL :: flag
INTEGER :: status(MPI_STATUS_SIZE), ierror
```

Multiple Completions

- When a number of non-blocking messages have been posted, it is often useful to wait/test for the completion of a number of these at a time.
- `MPI_Waitall`/`MPI_Testall` — waits/tests for the completion of all listed pending operations.
- `MPI_Waitany`/`MPI_Testany` — waits/tests for the completion of any one of the listed pending operations, returning the index to the handle of one completed message.
- `MPI_Waitsome`/`MPI_Testsome` — waits/tests for the completion of at least one of the listed pending operations, returning the indices to the handles of all completed messages.

"Any" can be dangerous in that you can miss completions.

Timing

C++ `(double) MPI_Wtime()`

Fortran `DOUBLE PRECISION MPI_WTIME()`

These *really are* functions, unlike practically everything else in MPI

Returns number of (elapsed) seconds since a time in the past (which is guaranteed not to change during the life of the process). Standard operation:

```
old=MPI_Wtime();
<code>
t=MPI_Wtime();
std.cout << "time taken" << t-old <<"seconds";
```

This is "per process": we'll see how to add them up.

Question 6 Pseudocode

```
<initialise>
base=MPI_Wtime();
vector<double> timestamps(size);
if (rank==0) {
    timestamps[0]=MPI_Wtime()-base;
    MPI_Send(...)
    MPI_Recv(...) }
else {
    MPI_Recv(...)
    timestamps[rank]=MPI_Wtime()-base;
    MPI_Send(...);  }
```

MPI Datatypes

MPI provides a mechanism for user-defined data types, analogous to `struct` in C or F90 derived data types.

but they needn't be contiguous in memory, that is to say that `MPI_Send` can implicitly do a "gather", and `MPI_Recv` a "scatter".

- MPI "normally" deals with the rounding rules, so a `double` followed by 3 `char` would actually have size 16 to align properly.
- There's a lot, and this talk is just scratching the surface.

Contiguous Types

This makes a new type which is count copies of an existing type, which are contiguous in memory (the easy case!)

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)
```

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

Since there are count fields in MPI_Send and MPI_Recv, this is only needed as part of more general constructions.

Strided Vector Types

e.g. "two doubles, then skip 4, then two more . . . with a total of 10 lots, i.e. 20 doubles"

```
int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYP
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERRO
```

So we would have

```
MPI_Type_vector(10,2,2+4,MPI_DOUBLE,&my_type)
```

Note the "2+4": MPI wants the stride, i.e. the distance between successive block starts.

! `stride` can be negative, and -1 does an array backwards

- `MPI_Type_create_hvector` takes the `stride` in bytes (portability warning!)

Indexed Vector Types

e.g. "3 doubles starting 4 in, then 1 double starting 0 in"

```
int MPI_Type_indexed(int count, const int array_of_blockler
const int array_of_displacements[], MPI_Datatype oldtype,
MPI_Datatype *newtype)
```

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS,OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
ARRAY_OF_DISPLACEMENTS(*),OLDTYPE, NEWTYPE, IERROR)
```

So (in pseudo-code)

```
MPI_Type_indexed(2,[3,1],[4,0],MPI_DOUBLE,&new_type);
```

- Note the `count` is the number of blocks, not items.
- `MPI_Type_create_hindexed` has the displacements (only) in bytes.
- `MPI_Type_create_indexed_block` has one constant block length

Structured Vector Types

As above, but each block can be of a different data type. e.g. "3 doubles starting 64 bytes in, then 7 char starting 0 in"

```
int MPI_Type_create_struct(int count,
const int array_of_blocklengths[],
const MPI_Aint array_of_displacements[],
const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*),
NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

So (in pseudo-code)

```
MPI_Type_indexed(2,[3,7],[64,0],[MPI_DOUBLE,MPI_CHAR],&new
```

Sub-arrays

"A $2 \times 3 \times 4$ subarray of a $20 \times 30 \times 40$ C array, starting at $(5, 6, 7)$"

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

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SU
ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

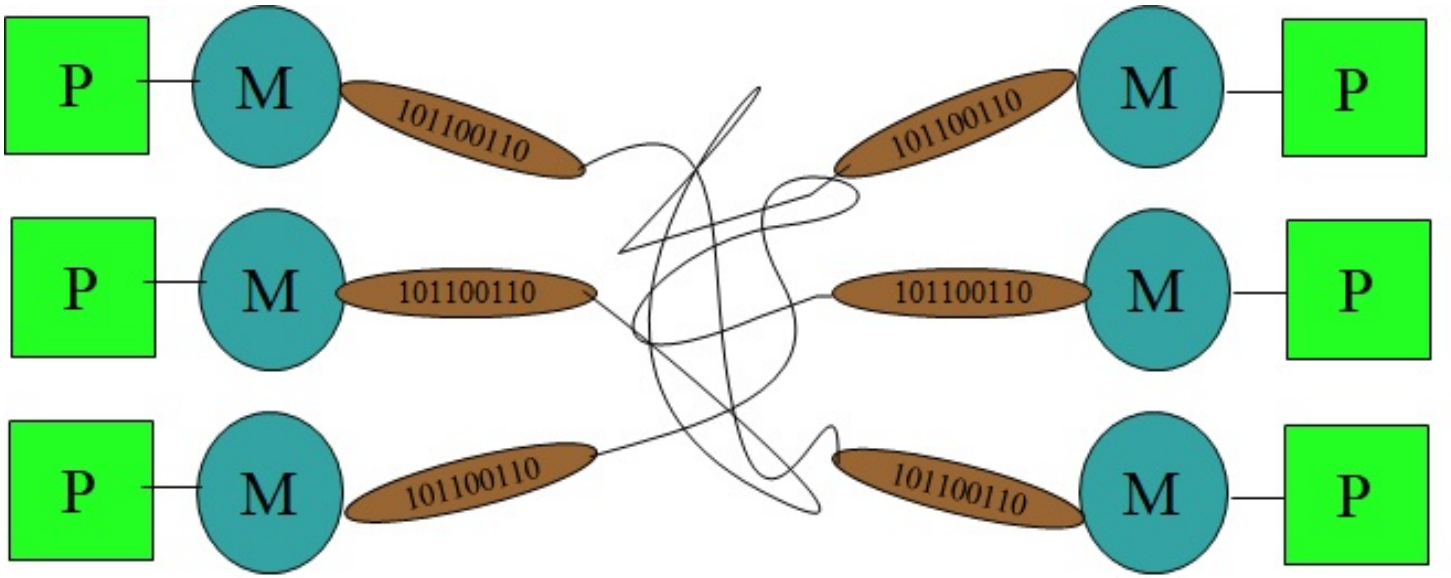
So (in pseudo-code)

```
MPI_Type_create_subarray(3,[20,30,40],[2,3,4],[5,6,7],
    MPI_ORDER_C,MPI_DOUBLE,&new_type);
```

Fortran users need to subtract 1 from the `starts`

Collective Communication

- A communication involving a group of processes is called collective.
- The group of processes involved is defined by the communicator used in the call.
- **All** collective calls must be made by **every process** in the group associated with the communicator.



Barrier Synchronization

- It may be important that all processes have finished their part of a task before any process proceeds or that all processes begin work at the same time.
- The routine `MPI_Barrier` is used to synchronize a group of processes. No data are transferred.

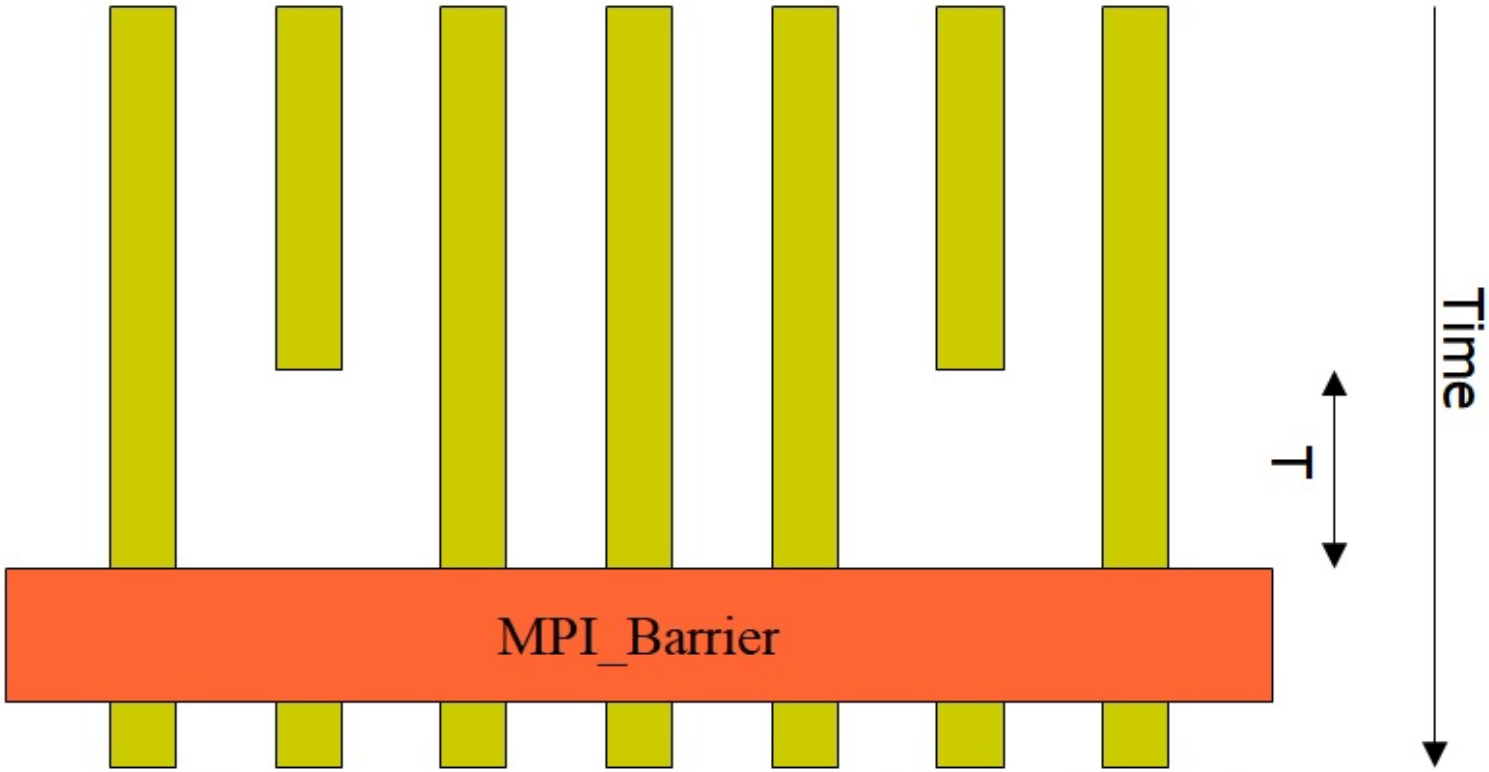
```
int MPI_Barrier(MPI_comm comm);
```

```
SUBROUTINE MPI_BARRIER(COMM, IERROR)
INTEGER :: COMM, IERROR
```

- A call to `MPI_Barrier` does not return until all processes associated with the communicator `comm` have called it.
- If they don't, then execution will deadlock.
- It is the programmer's responsibility to make sure they do.

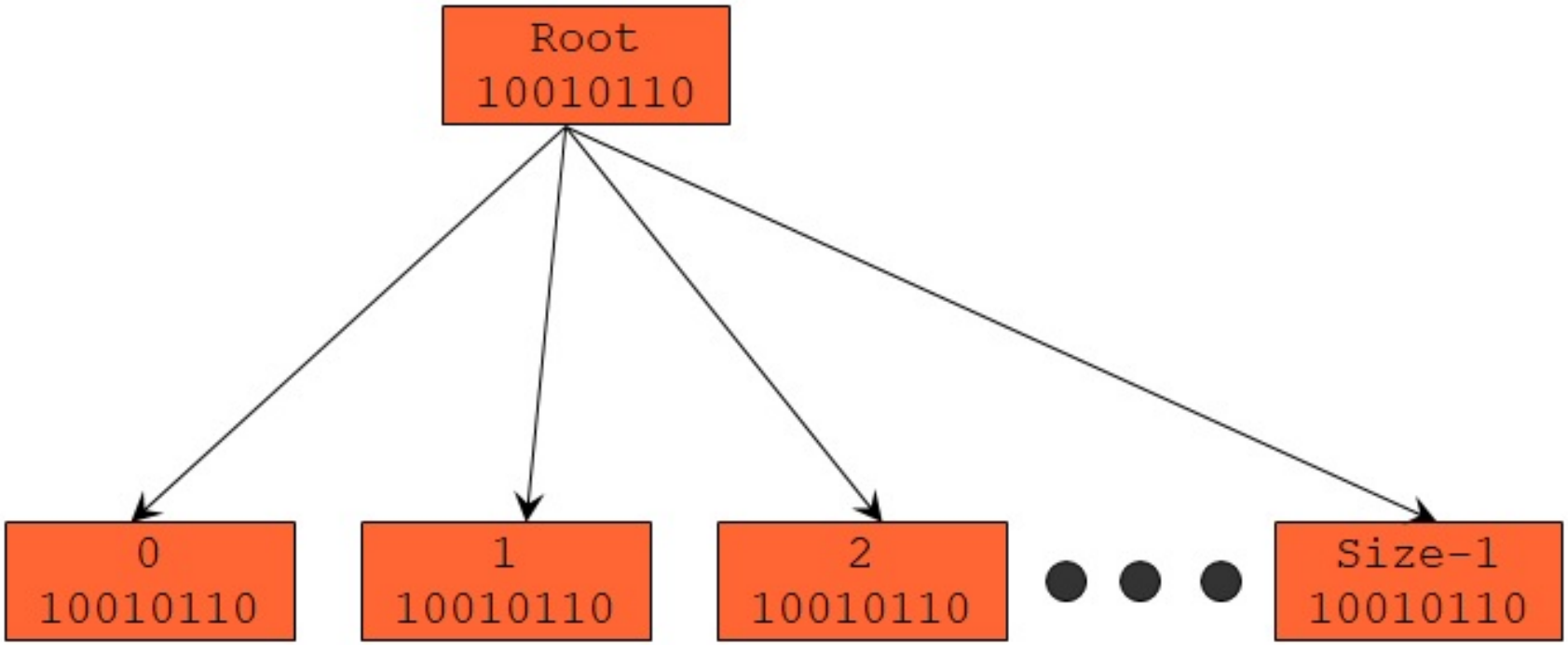
Barrier Illustrated

MPI_Barrier



Here five processes are waiting T seconds for two processes to reach the barrier.

Broadcast Illustrated



Data held in the data buffer on process root is copied into the data buffers on all processes in the group associated with comm.

Broadcasting data

- `MPI_Bcast` copies data from a specified root process to all processes in a group.
- As with all collective calls, this must be called by every process associated with the communicator.

```
int MPI_Bcast(void *buffer, int count,
        MPI_Datatype datatype, int root, MPI_Comm comm);
SUBROUTINE MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, &
         COMM, IERR)
<type> :: BUFFER(*)
INTEGER :: COUNT, DATATYPE, ROOT, COMM, IERR
```

Broadcast — C example

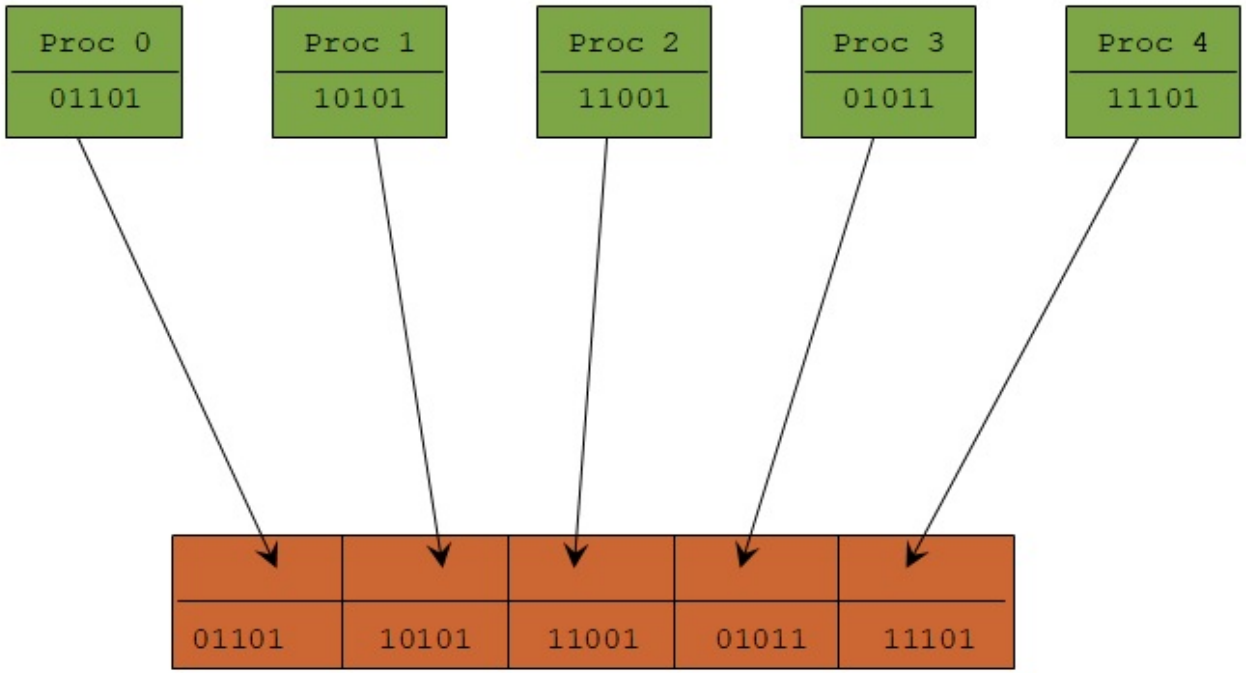
```
include <mpi.h>
include "mouse.h"
extern int size, rank;

int get_calc_type(window_t * w){
  int calc_type;

  if (rank == 0)
    get_mouse_event(w,&calc_type);

  MPI_Bcast(&calc_type, 1, MPI_INT, 0, MPI_COMM_WORLD);

  return calc_type;
}
```

Broadcast — Fortran example

```
SUBROUTINE get_calc_type(rank, size, w, calc_type)
IMPLICIT NONE
INCLUDE "mpif.h"
INCLUDE "mouse.h"
INTEGER, INTENT(IN) :: rank, size
INTEGER, INTENT(OUT) :: calc_type
TYPE (WINDOW_T), INTENT(IN) :: w
INTEGER :: ierr

IF (rank == 0) CALL get_mouse_event(w, calc_type)

CALL MPI_BCAST(calc_type, 1, MPI_INTEGER, 0, &
      MPI_COMM_WORLD, ierr)

RETURN
END SUBROUTINE get_calc_type
```

Gathering data from processes

- When each process has computed some data that needs to be gathered to give the final result, the routine `MPI_Gather` can be used.



MPI_Gather

```
int MPI_Gather(void *sendbuf, int sendcount,
      MPI_Datatype sendtype, void *recvbuf, int recvcount,
     MPI_Datatype recvtype,  int root, MPI_Comm comm);
```

```
SUBROUTINE MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,  &
    RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<sendtype> :: SENDBUF(*)
<recvtype> :: RECVBUF(*)
INTEGER :: SENDCOUNT, SENDTYPE
INTEGER :: RECVCOUNT, RECVTYPE
INTEGER :: ROOT, COMM, IERROR
```

MPI_Gather

- After a call to `MPI_Gather`, `recvbuf` on process `root` contains the data from each `sendbuf` in rank order
- includes the data from its own `sendbuf`.

Note that `recvcount` is the amount of data expected to be received from each process, not the total amount.

- `MPI_Gatherv` has an array of `recvcount` so you can work out what has been received
- If every process, rather than a single root process, requires the gathered data, then use `MPI_Allgather`.
- This is functionally equivalent to an `MPI_Gather` followed by an `MPI_Bcast`.
- There's also a converse `MPI_Scatter`.

Reduction

- Suppose each process has computed x_i, and what we want is $X = \sum_{i=0}^{\mathrm{size}-1} x_i$.
- This is reduction in MPI-speak.
- Can be done element-by-element on arrays
- Needn't be + [but must be mathematically associative — order of evaluation doesn't matter]
- The order in which the reduction is done is unspecified, so the result is guaranteed to be the same only to within the accuracy of round-off errors (see Arithmetic lecture)

MPI Reduction routines

There are two standard routines for performing global reduction operations.

- `MPI_Reduce` performs the reduction and returns the result to a specific process.
- `MPI_Allreduce` performs the reduction and returns the result to all the processes associated with the communicator.

+ Equivalent to `MPI_Reduce` followed by `MPI_Bcast`.

MPI_Reduce operations: type MPI_Op

Operator	Meaning	C	FORTRAN
MPI_MAX	Maximum		MAX(a1,a2)
MPI_MIN	Minimum		MIN(a1,a2)
MPI_SUM	Sum	+	+
MPI_PROD	Product	*	*
MPI_LAND	Logical and	&&	.AND.
MPI_BAND	Bitwise and	&	
MPI_LOR	Logical or	||	.OR.
MPI_BOR	Bitwise or	|	
MPI_LXOR	Logical xor	!=	.NEQV.
MPI_BXOR	Bitwise xor	^	

Note that all operators are usable from all languages, even if there's no language equivalent

MPI_Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm);
```

```
SUBROUTINE MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, &
      OP, ROOT, COMM, IERROR)
<type> :: SENDBUF(*), RECVBUF(*)
INTEGER :: COUNT, DATATYPE, OP
INTEGER :: ROOT, COMM, IERROR
```

MPI_Allreduce

```
int MPI_Allreduce(void *sendbuf,
void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm);
```

```
SUBROUTINE MPI_ALLREDUCE(SENDBUF, RECVBUF,
    COUNT, DATATYPE, OP, COMM, IERROR)
<type> :: SENDBUF(*), RECVBUF(*)
INTEGER :: COUNT, DATATYPE, OP, COMM, IERROR
```

Note no root argument to MPI_Alleduce.

Reduce-Scatter routines

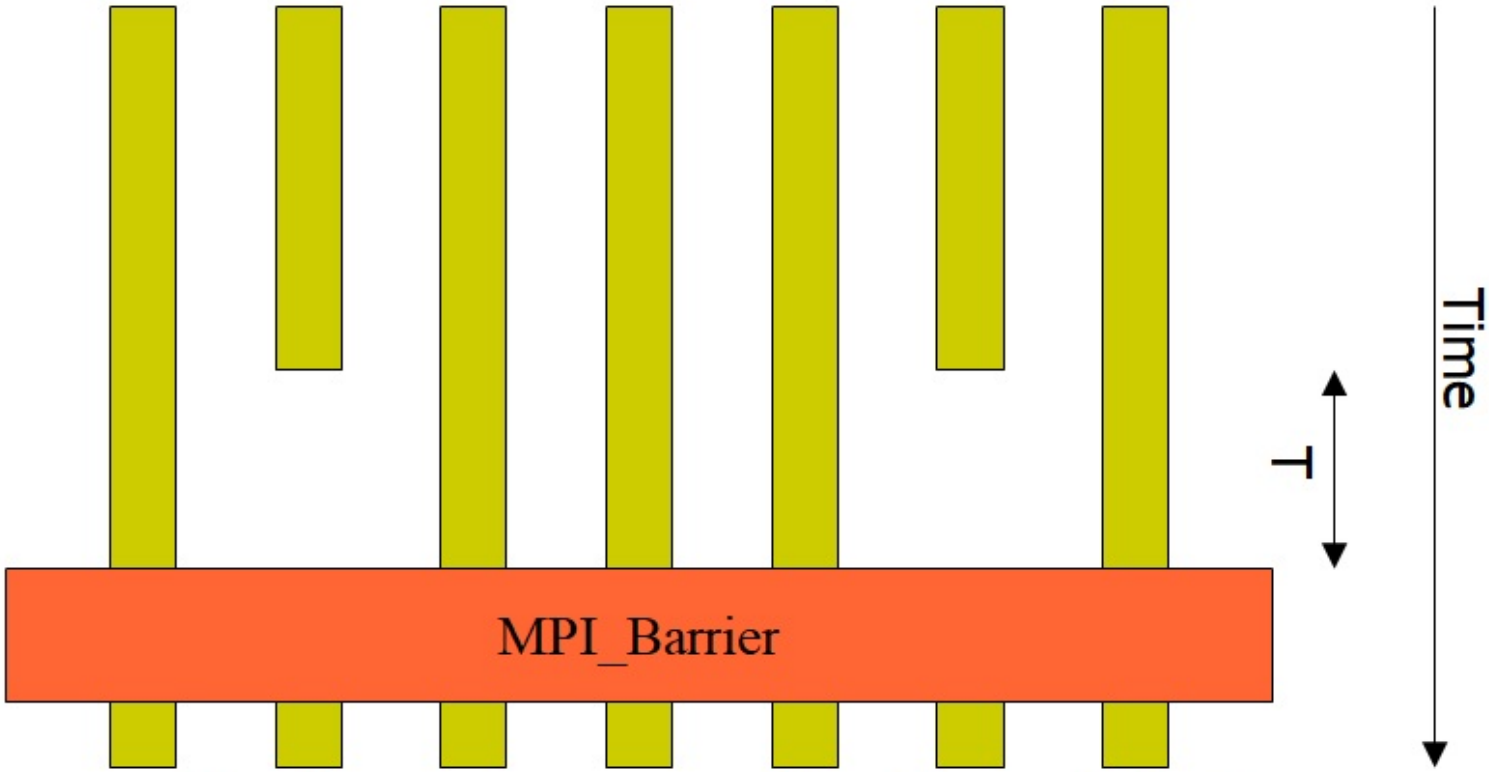
- There are also two MPI reduction routines where the result of the reduction operation is scattered amongst all the processes in the group.
- `MPI_Reduce_scatter_block` is functionally equivalent to an `MPI_Reduce` followed by an `MPI_Scatter`.
- `MPI_Reduce_scatter` is functionally equivalent to an `MPI_Reduce` followed by an `MPI_Scatterv`.
- Direct implementations may run faster than calling the reduce and scatter separately.

Non-blocking collectives

All these collective operations are blocking (as if broken down into `Send/Recv` primitives)
There are also non-blocking versions `MPI_Ireduce` etc., with `Test/Wait` needed as necessary.
Non-blocking `Barrier` might seem a bit odd!

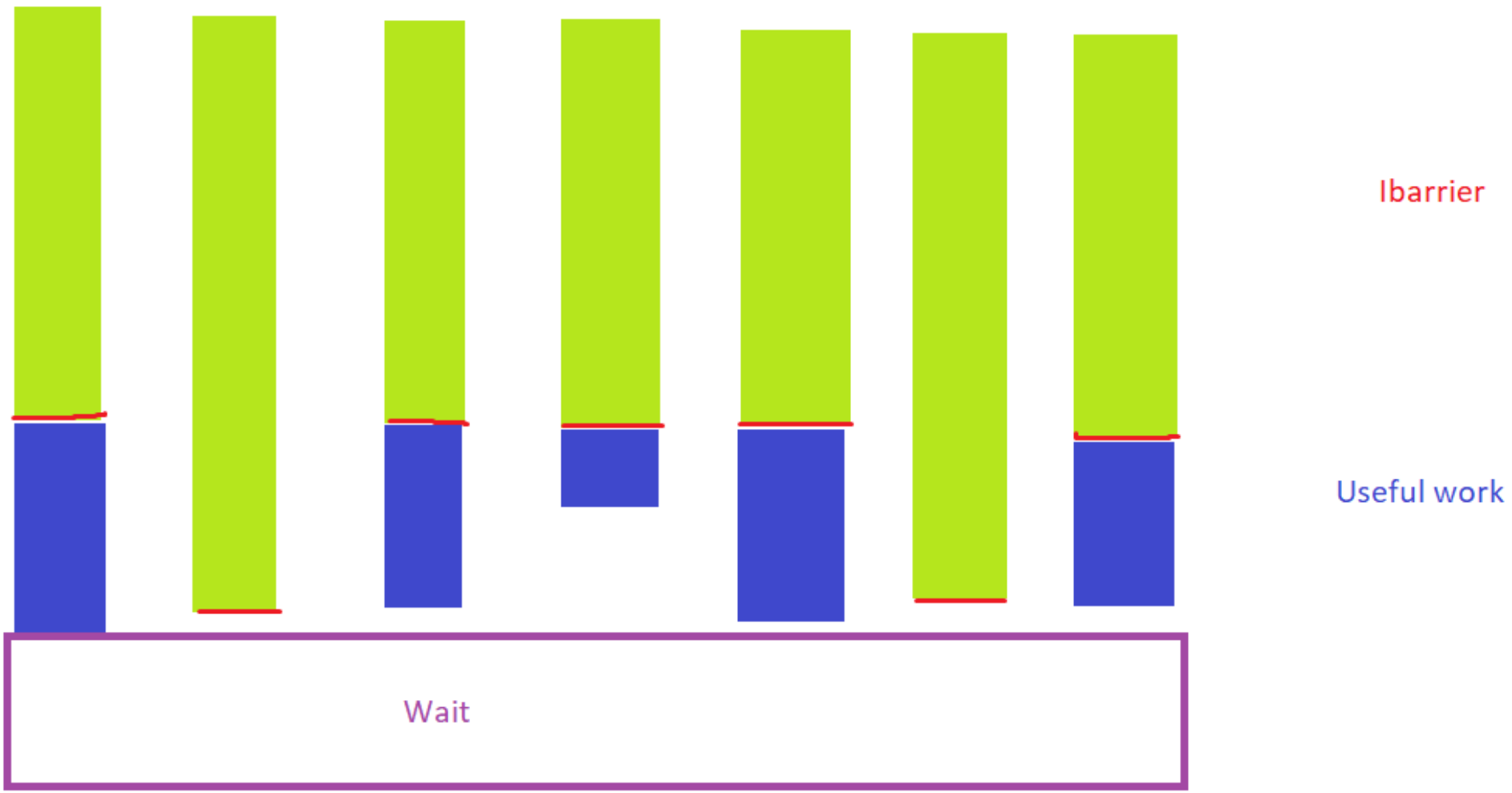
Barrier Illustrated

MPI_Barrier



Here five processes are waiting T seconds for two processes to reach the barrier.

Ibarrier Illustrated



Here five processes can do varying amounts of useful work after calling `MPI_Ibarrier`

Topologies

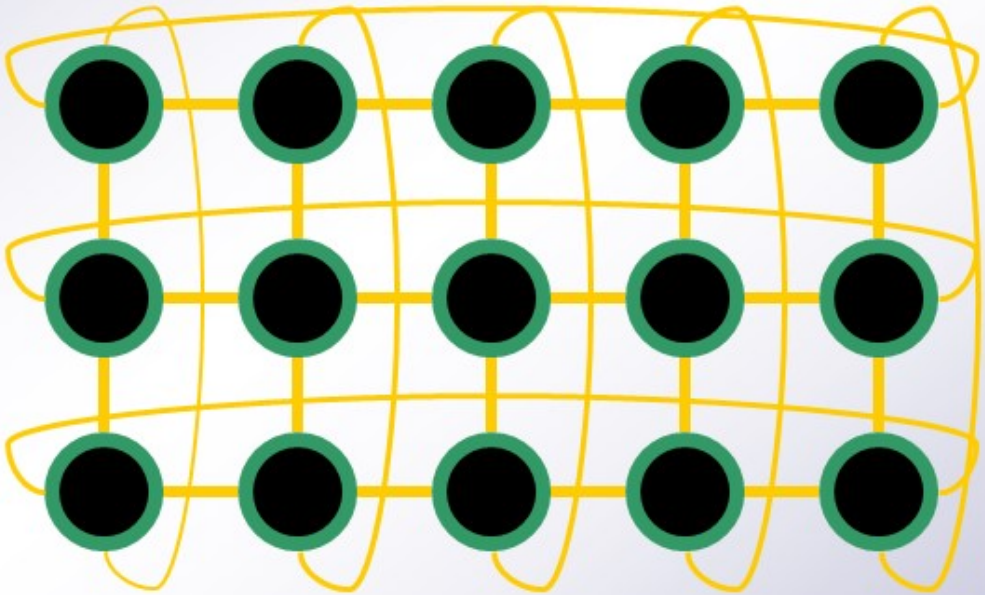
Formally, the fact that each process has a `rank` is sufficient: this lets us define any structure we want. But

- Inefficient (and possibly error-prone)
- We don't tell MPI what we want to do.

We want to define the *virtual topology* of our tasks (say a 3D grid) and let MPI map this to the *real topology* of the hardware (which isn't known until run-time and the allocated nodes are known).
All topologies can be defined by a graph, but in practice many are Cartesian, and specifying the full graph is tedious. General graph topologies are specified by `MPI_Graph_create`, or `MPI_Dist_graph_create_adjacent` if each node only knows its neighbours, rather than the full topology.

A 2D Cartesian topology

- Here, periodicity is set to true so the process with the highest coordinate has the process with the lowest co-ordinate above it, etc.



Process Coordinates

- Process coordinates within a Cartesian topology are numbered the same way in Fortran and C.
- Coordinates are numbered from 0.
- Row-major numbering is always used.
- For example, for 6 processes in a 2×3 grid:

Rank=0 Coord=(0,0)	Rank=1 Coord=(0,1)	Rank=2 Coord=(0,2)
Rank=3 Coord=(1,0)	Rank=4 Coord=(1,1)	Rank=5 Coord=(1,2)

MPI_Cart_create

Takes the old communicator comm_old and builds a Cartesian communicator.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int
const int periods[], int reorder, MPI_Comm *comm_cart)
```

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, C
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER
```

- reorder says whether the ranks in the new communicator can be different.
- periods says (for each dimension separately) whether it's periodic

processes that don't fit get MPI_COMM_NULL as the new communicator

Topology-aware MPI code checklist

1. Include the appropriate header file or module
2. Initialise the MPI environment using `MPI_Init`
3. Initialise the topology, with `reorder` true
4. Each MPI process must find out the total number of processes using `MPI_Comm_size`
5. Each MPI process must find out its own unique rank using `MPI_Comm_rank`

* Now we can do the actual work

6. Shutdown the MPI environment using `MPI_Finalize`

A topology means we have neighbours

We specify the direction (0 $\leq$direction$<$ndims, even in Fortran) and a displacement (generally 1)

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

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
     RANK_DEST, IERROR)
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERR
```

- We get back the ranks of the source and destination for a shift of that type.

`MPI_PROC_NULL` if it doesn't exist: this is legitimate as a source or destination (always giving 0 objects)

Shifting Data Example - C

```
int up, down, left, right;
double mydata, updata, downdata, leftdata, rightdata;
...
/* Create a 2D grid, as in the earlier example. */
...
/* Find the ranks of the 4 neighbouring processes */
MPI_Cart_shift(comm_cart, 0, 1, &up, &down);
MPI_Cart_shift(comm_cart, 1, 1, &left, &right);

/* Shift mydata to the right neighbour */
MPI_Sendrecv(&mydata, 1, MPI_DOUBLE, right, 0, &leftdata, 1, MPI_DOUBLE, left,
   0, comm_cart, &status);
/* Shift mydata to the left neighbour */
MPI_Sendrecv(&mydata, 1, MPI_DOUBLE, left, 0, &rightdata, 1, MPI_DOUBLE, right,
   0, comm_cart, &status);
/* Shift mydata up one */
MPI_Sendrecv(&mydata, 1, MPI_DOUBLE, up, 0, &downdata, 1, MPI_DOUBLE, down, 0,
   comm_cart, &status);
/* Shift mydata down one */
MPI_Sendrecv(&mydata, 1, MPI_DOUBLE, down, 0, &updata, 1, MPI_DOUBLE, up, 0,
   comm_cart, &status);
```

Collectives

Some of these generalise to topologies, and provide a function I can't mimic without topologies.

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

```
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE,
RECVBUF, RECVCOUNT,RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IER
```

The send buffer is sent to each neighboring process and the l-th block in the receive buffer is received from the l-th neighbor. Also `allgatherv` version, non-blocking versions, and a `MPI_Neighbor_alltoall` where different items can be sent to different neighbours.

Bibliography I

L. Anton, M. Ashworth, X. Guo, S. Pickles, A. Porter, and A. Sunderland.
Exploiting multi-core processors for scientific applications using hybrid MPI-OpenMP.
Technical Report DL-TR-2015-002, Daresbury Labs, 2015.

Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, et al.
The Tofu Interconnect D.
In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 646–654. IEEE, 2018.

Bibliography II


M. Allalen.
Best Practice Guide - Modern Interconnects.
http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Modern-Interconnects.pdf, 2019.


M. Ashworth.
Performance of Coarray Fortran vs MPI in a CFD Application.
https://www.softwareoutlook.ac.uk/sites/www.softwareoutlook.ac.uk/files/Ashworth_Poster_PGAS14_08Oct14.pdf, 2014.


J. Bachan, S.B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P.H. Hargrove, and H. Ahmed.
UPC++: A High-Performance Communication Framework for Asynchronous Computation.
In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 963–973, 2019.

Bibliography III


Message Passing Interface Forum.
MPI: A Message-Passing Interface Standard Version 3.1.
http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.


J. Squyres.
SEND, ISEND, or SENDRECV...?
https://web.archive.org/web/20160316101539/http://blogs.cisco.com/performance/send_isend_or_sendrecv, 2009.