MPI - Message Passing Interface

`www.mpi-forum.org` (official MPI standard documents)
[Gropp/Lusk/Skjellum'95] [Pacheco'97]

MPI core routines
Modes for point-to-point communication
Collective communication operations
Virtual processor topologies
Group concept
Communicator concept
One-sided communication (MPI-2)
Fork-join-parallelism (MPI-2)

MPI - principles

MPI standard for message passing created in 1993

- API with C and Fortran bindings
- replaced vendor-specific message passing libraries
- replaced other de-facto standards: PICL, PARMACS, PVM
- abstraction from machine-specific details
- enhanced portability (though at a low level)
- efficient implementations (avoid unnecessary copying)
- implemented on almost all parallel machines

MPI-1.1 1995
MPI-2 1997

Free implementations (e.g. for NOWs, Linux clusters):
MPICH (Argonne), LAM (Ohio), CHIMP (Edinburgh), ...

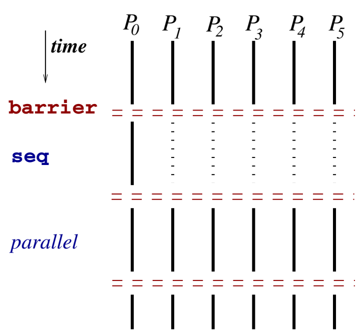
MPI - program execution

Run a MPI executable: with (platform-dependent) shell script

`mpirun` -np 6 `a.out` [args]

creates fixed set of 6 processes that execute `a.out`

- fixed set of processors
- no spawn() command
- main() executed by all started processors as one group



SPMD execution style

MPI - determinism

Message passing is generally nondeterministic:
Arrival order of two sent messages is unspecified.

MPI guarantees that two messages sent from processor A to B
will arrive in the order sent.

Messages can be distinguished by sender and a tag (integer).

User-defined nondeterminism in receive operations:

wildcard `MPI_ANY_SOURCE`

wildcard `MPI_ANY_TAG`

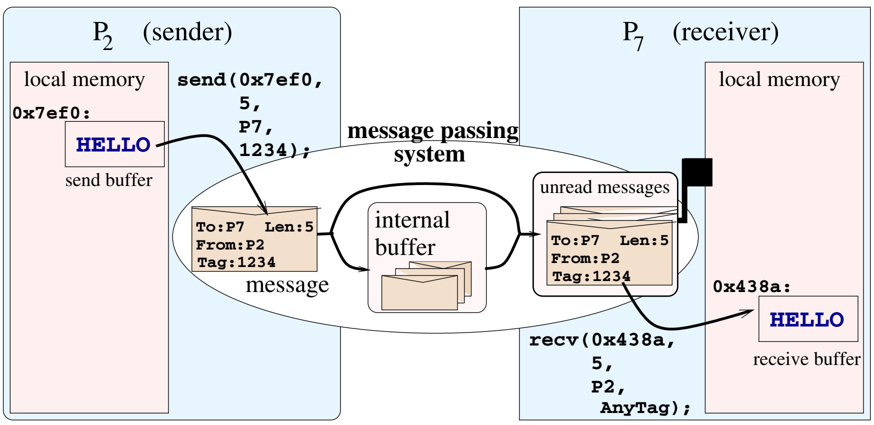
MPI core routines

```
MPI_Init( int *argc, char ***argv );

MPI_Finalize( void );

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

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

MPI_Comm_size( MPI_Comm comm, int *psize);

MPI_Comm_rank( MPI_Comm comm, int *prank);
```

Status object:

status->`MPI_SOURCE` indicates the sender of the message received;
status->`MPI_TAG` indicates the sender of the message received;
status->`MPI_ERROR` contains an error code.

Hello World (1)



Hello World (2)

```
#include <mpi.h>

void main( void )
{
 MPI_Status status;
 char *string = "xxxxx";  // receive buffer
 int myid;

 MPI_Init( NULL, NULL );
 MPI_Comm_rank( MPI_COMM_WORLD, &myid );
 if (myid==2)
   MPI_Send( "HELLO", 5, MPI_CHAR, 7, 1234, MPI_COMM_WORLD );
 if (myid==7) {
   MPI_Recv( string, 5, MPI_CHAR, 2, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status );
   printf( "Got %s from P%d, tag %d\n",
           string, status.MPI_SOURCE, status.MPI_TAG );
 }
 MPI_Finalize();
}
```


MPI predefined data types

Some predefined data types in MPI:

MPI_Datatype	Corresponding C type
MPI_CHAR	char
MPI_BYTE	—
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Recommended for program portability across platforms

MPI communication operations

An MPI communication operation (i.e., call to send or receive routine) is called

blocking if the return of program control to the calling process
means that all resources (e.g., buffers) used in the operation
can be reused immediately;

nonblocking or **incomplete** if the operation returns control
to the caller *before* it is completed, such that buffers etc.
may still be accessed afterwards by the started communication activity,
which continues running in the background.

In MPI, nonblocking operations are marked by an `I` prefix.

MPI communication modes

A MPI communication can run in the following modes:

standard mode: the default mode:

synchronicity and buffering depends on the MPI implementation.

synchronous mode:

send and receive operation are forced to work partly simultaneously:
send returns when receive has been started.

buffered mode: (the buffer can be attached by the programmer)

send returns when its send buffer has either been received
or written to a temporary buffer → decouples send and receive

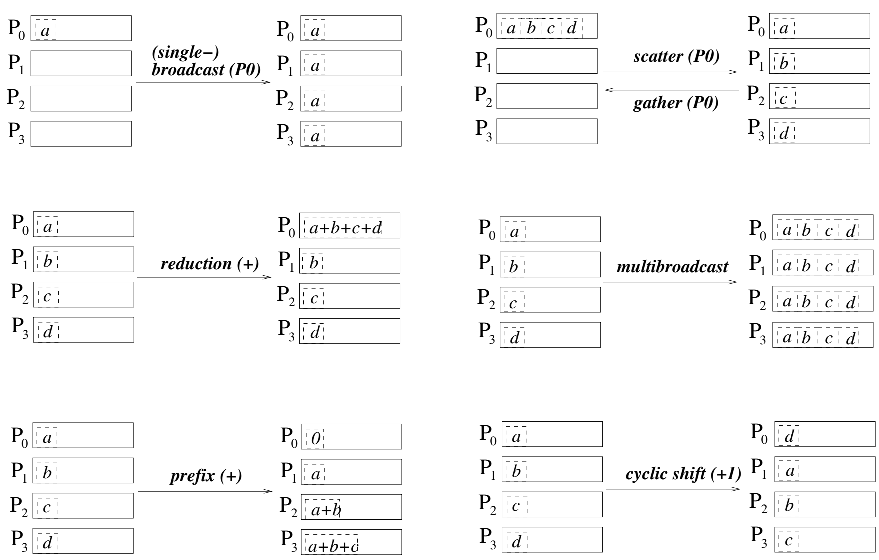
In MPI, the mode is controlled by a prefix (none, `S`, `B`) of the send operation.

Overview of some important point-to-point communication operations

Operation type	blocking		nonblocking	
Communication mode	send	receive	send	receive
standard	MPI_SEND	MPI_RECV	MPI_ISEND ... ↓ request MPI_WAIT	MPI_IRECV ... ↓ request MPI_WAIT
synchronous	MPI_SSEND		MPI_ISSEND ... ↓ request MPI_WAIT	
buffered	MPI_BSEND		MPI_IBSEND ... ↓ request MPI_WAIT	
tentative	MPI_*SEND	MPI_PROBE	MPI_I*SEND ... ↓ request MPI_WAIT	MPI_IPROBE ... ↓ request MPI_WAIT

Remarks: there are further routines, another mode "ready",
`MPI_TEST` as alternative to `MPI_WAIT`

MPI - Collective communication operations (1)



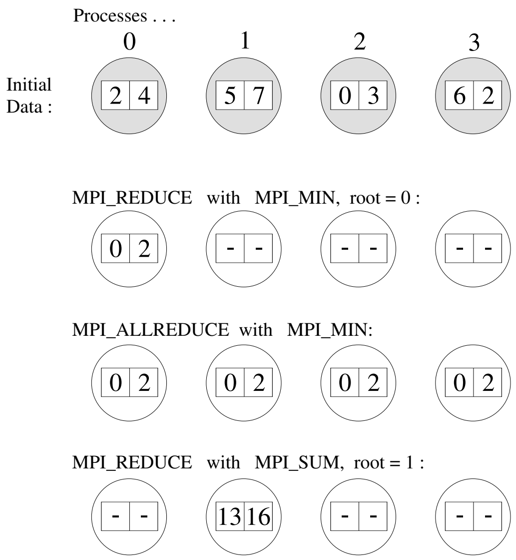
MPI - Collective communication operations (2)

```
MPI_Bcast( void *srbuf, int count, MPI_Datatype datatype,
                             int rootrank, MPI_Comm comm );

MPI_Reduce( void *sbuf, void *rbuf, int count,
              MPI_Datatype datatype, MPI_Op op, int rootrank,
                                          MPI_Comm comm );
```

with predefined $op \in \{$ `MPI_SUM`, `MPI_MAX`, ... $\}$
or user-defined by `MPI_Op_Create`.

```
MPI_Allreduce

int MPI_Barrier( MPI_Comm comm );
```


MPI - Collective communication operations (3): reductions



elementwise reductions

Source: Ian Foster: Designing and building parallel programs. ©1995 Addison Wesley.

MPI - Collective communication operations (4)

```
switch (my_rank) {
   case 0:  MPI_Bcast( buf1, count, type, 0, comm );
            MPI_Bcast( buf2, count, type, 1, comm );
            break;
   case 1:  MPI_Bcast( buf2, count, type, 1, comm );
            MPI_Bcast( buf1, count, type, 0, comm );
            break;
}
```

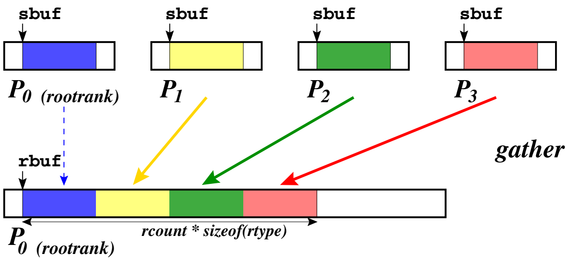
Deadlock risk!

(a) run-time system may interpret the first Bcast calls
of each processor as part of the *same* Bcast operation
$\rightarrow$ error (different roots specified)

(b) otherwise deadlock if no or too small system buffers used:
global communication operations in MPI are always blocking.

MPI - Collective communication operations (5)

```
int MPI_Scatter( void *sbuf, int scount, MPI_datatype stype,
                 void *rbuf, int rcount, MPI_datatype rtype,
                 int rootrank, MPI_Comm comm );

int MPI_Gather( void *sbuf, int scount, MPI_datatype stype,
                void *rbuf, int rcount, MPI_datatype rtype,
                int rootrank, MPI_Comm comm );
```

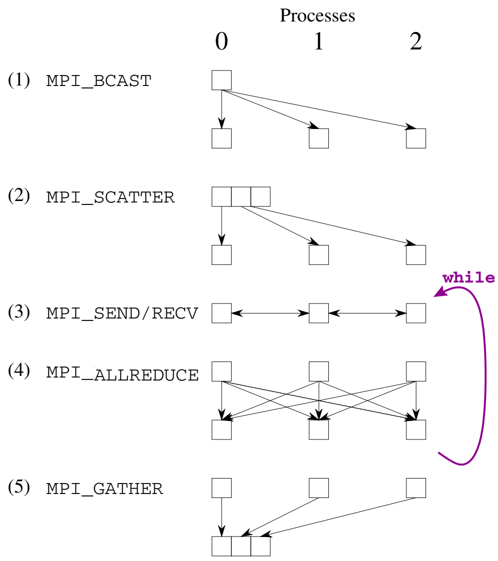


Also, `MPI_Scatterv` and `MPI_Gatherv` for variable-sized local partitions

MPI - Collective communication operations (6)

Example: Finite differences

(→ code handed out)



Source: Ian Foster: Designing and building parallel programs.

Virtual topologies in MPI

Example: arrange 12 processors in 3×4 grid

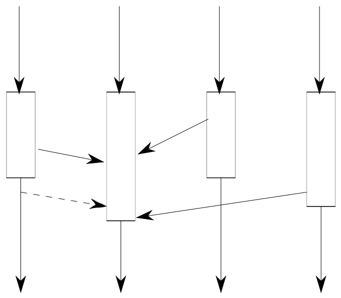
```
int dims[2], coo[2], period[2], src, dest;
period[0]=period[1]=0;  // 0=grid, !0=torus
reorder=0; // 0=use ranks in communicator,
           // !0=MPI uses hardware topology
dims[0] = 3; // extents of a virtual
dims[1] = 4; // 3X4 processor grid

// create virtual 2D grid topology:
MPI_Cart_create( comm, 2, dims, period,
                 reorder, &comm2 );

// get my coordinates in 2D grid:
MPI_Cart_coords( comm2, myrank, 2, coo );

// get rank of my grid neighbor in dim. 0
MPI_Cart_shift( comm2, 0, +1, // to south,
                &src, &dest); // from south
...
```

0 (0,0)	**1** (0,1)	**2** (0,2)	**3** (0,3)
4 (1,0)	**5** (1,1)	**6** (1,2)	**7** (1,3)
8 (2,0)	**9** (2,1)	**10** (2,2)	**11** (2,3)

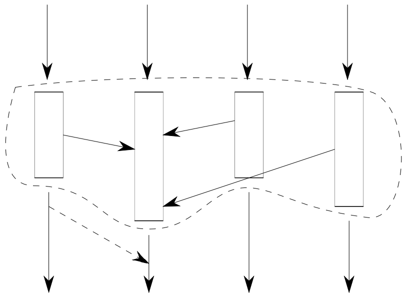
```
...
coo[0]=i; coo[1]=j;

// convert cartesian coordin.
// (i,j) to rank r:
MPI_Cart_rank(comm, coo, &r);

// and vice versa:
MPI_Cart_coords(comm,r,2,coo);
```

Communicator concept – Motivation

Communication error in a sequential composition
where a message is intercepted
by a library routine:



Source: Ian Foster: Designing and building parallel programs.
©1995 Addison Wesley.



Avoid error by using a separate context
(separate tag space for messages)

Communicator concept

Communicators provide information hiding when building modular programs.

- identify a process group and the context in which a communication occurs.
- encapsulate internal communication operations within a process group
  (e.g. through local process identifiers)

$\rightarrow$ MPI supports sequential and parallel module composition
(concurrent composition only for MPI-2)

Default communicator: `MPI_COMM_WORLD`

- includes all MPI processes
- defines default context

Communicator functions

`MPI_COMM_DUP` (*comm*, *newcomm*)

creates a new communicator with same processes as *comm*
but with a different context with different message tags.

$\rightarrow$ supports sequential composition

Furthermore:

`MPI_COMM_SPLIT` (*comm*, *color*, *key*, *newcomm*)

create a new communicator for a *subset of a group* of processes

`MPI_INTERCOMM_CREATE` (*comm*, *local_leader*, ... *remote_leader*, ...*intercomm*)

create an intercommunicator, linking processes in different groups

`MPI_COMM_FREE` (*comm*)

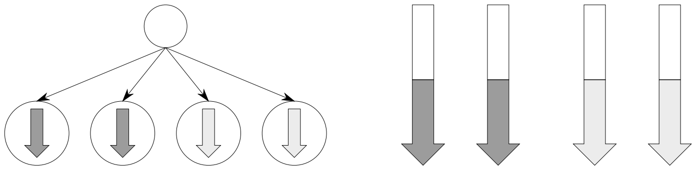
release previously created communicator *comm*

Communicators for splitting process sets

MPI_COMM_SPLIT (*comm*, *color*, *key*, *newcomm*)

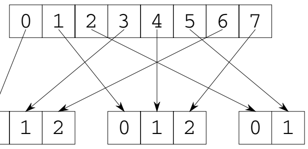
used for parallel composition of process groups.

A fixed set of processes changes character.



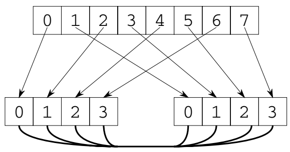
Example:

```
color = myid % 3
    // make color 0, 1, or 2
MPI_COMM_SPLIT( comm, color,
                    key, newcomm)
```



Source: Ian Foster: Designing and building parallel programs. ©1995 Addison Wesley.

Communicators for communicating between process groups

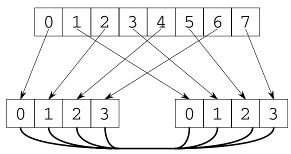


An *intercommunicator* connects two process groups

- needs a common parent process (*peercomm*)
- needs a leader process for each process group (*local_leader*, *remote_leader*)
- The local communicator *comm* denotes one of the process groups
- The created intercommunicator is placed in *intercomm*
- The *tag* is used for “safe” communication between the two leaders

MPI_INTERCOMM_CREATE (*comm*, *local_leader*,
peercomm, *remote_leader*, *tag*, *intercomm*)

Communicators for communicating between process groups (cont.)



Example:

(program fragment executing on each processor)

split into 2 groups: odd / even numbered

```
call MPI_COMM_SPLIT( MPI_COMM_WORLD, mod(myid,2), myid, comm, ierr)
...
if (mod(myid,2) .eq. 0) then
```

Group 0: create intercommunicator and send message

local leader: 0, remote leader: 1, tag = 99

```
  call MPI_INTERCOMM_CREATE( comm, 0, MPI_COMM_WORLD, 1, 99, intercomm,ierr
  ...
else
```

Group 1: create intercommunicator and send message

note that remote leader has ID 0 in MPI_COMM_WORLD:

```
  call MPI_INTERCOMM_CREATE( comm, 0, MPI_COMM_WORLD, 0, 99, intercomm,ierr
  ...
```

One-sided communication in MPI-2

"Receive Considered Harmful"

→ One-sided communication / Remote memory access (RMA)

RMA Windows

`int MPI_Win_create ( void *`*base*`, MPI_Aint `*size*`, int `*d*`,`
`MPI_Info `*info*`, MPI_Comm `*comm*`, MPI_Win *`*Win*` )`

open memory block *base* with *size* bytes for RMA by other processors
displacement unit *d* bytes (distance between neighbored elements)
additional info (typ. MPI_INFO_NULL) to runtime system
→ window descriptor *Win*

`MPI_Win_free ( MPI_Win *`*win*` )`

One-sided communication in MPI-2 (2)

3 non-blocking RMA operations:

`MPI_Put`
remote write

`MPI_Get`
remote read

`MPI_Accumulate`
remote reduction

Concurrent read and write leads to unpredictable results.

Multiple Accumulate operations on same location are possible.

One-sided communication in MPI-2 (3)

`MPI_Win_fence ( int `*assert*`, MPI_Win *`*win*` )`

global synchronization of all processors
that belong to the group that declared *win*
flushes all pending writes to *win* (→ consistency)
assert typ. 0 (tuning parameter for runtime system)

```
while (! converged( A )) {
  update ( A );
  update_buffer( A, from_buf );
  MPI_Win_fence ( 0, win );
  for (i=0; i<num_neighbors; i++)
    MPI_Put ( &from_buf[i], size[i], MPI_INT,
              neighbor[i],
              to_disp[i], size[i], MPI_INT,  win );
  MPI_Win_fence ( 0, win );
}
```

One-sided communication in MPI-2 (4)

Advanced issues

partial synchronization for a subgroup
synchronizing only the accessing and the accessed processor

lock synchronization of two processors
using a window on a third, not involved process as lock holder

Additional MPI / MPI-2 features

- Derived data types
  user can construct and register new data types in MPI type system,
  e.g. row/column vectors of certain length/stride,
  indexed vectors, aggregates of heterogeneous types
  $\rightarrow$ allows for extended type checking for incoming messages
- Process creation and management in MPI-2
- Additional global communication operations
- Environment inquiry functions

MPI Summary

SPMD style parallelism, p processes with fixed processor ID $0..p-1$
- dynamic process creation / concurrent composition possible in MPI-2

Processes interact by exchanging messages
- messages are typed (but not statically type-safe!)
- point-to-point communication in different modes
- collective communication
- probing for pending messages
- determinism / liveness not guaranteed,
  but can be achieved by careful programming

Modularity through communicators
- combine subprograms by sequential or parallel composition

One-sided communication in MPI-2