MPI - Message Passing Interface

Overview

MPI core routines
  Example: Hello World
Modes for point-to-point communication
Collective communication operations
  Example: 1D Finite Differences
Virtual processor topologies
Group concept
Communicator concept
One-sided communication (MPI-2)
Additional features
Literature

MPI - principles

MPI standard for message passing created in 1993

- API with C, 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
Extension MPI-2 1997 (MPI-2.1 2008, MPI-2.2 2009)

Free implementations (e.g. for PCs, NOWs, Linux clusters):
  MPICH (Argonne), OpenMPI (www.open-mpi.org), CHIMP (Edinburgh), ...

Commercial implementations (optimized) e.g. Scali MPI (on Neolith)

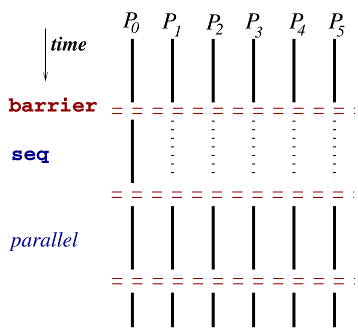
MPI - program execution

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

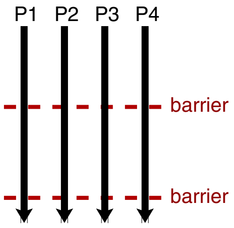


SPMD execution style

Background: SPMD execution style vs. Fork-join execution style

Parallel program execution styles

SPMD style



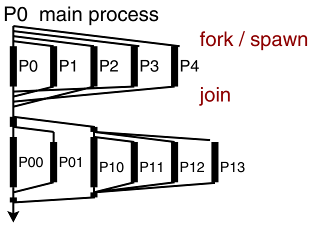
constant number of parallel activities
(processors / processes / threads)

static mapping to processors

mostly flat parallelism
(nested parallelism by group splitting)

Example: MPI, HPF, UPC, NestStep

Fork–join–style



dynamic creation and deletion
of parallel activities

needs dynamic scheduling (overhead)

naturally nested parallelism
(nested parallelism by nested spawning)

Example: pthreads, Java threads, Unix–fork,
OpenMP, PVM, MPI–2, Cilk

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 (C API)

```
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:

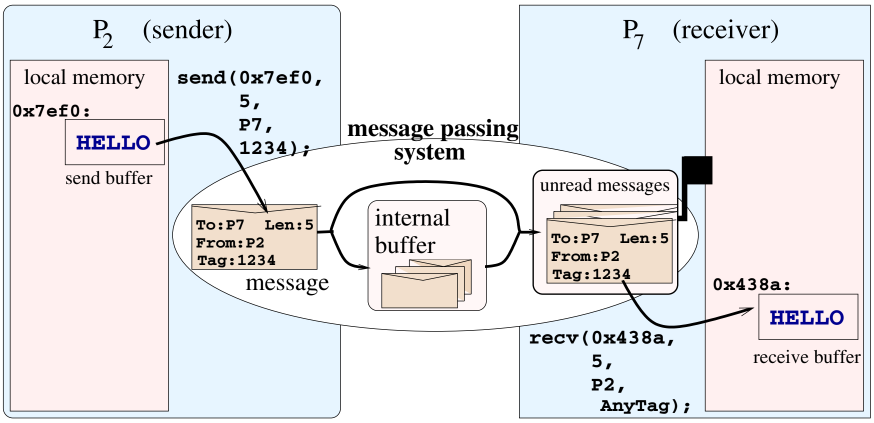
status->`MPI_SOURCE` indicates the sender of the message received;
status->`MPI_TAG` indicates the tag of the message received;
status->`MPI_ERROR` contains an error code.

MPI core routines (C++ API)

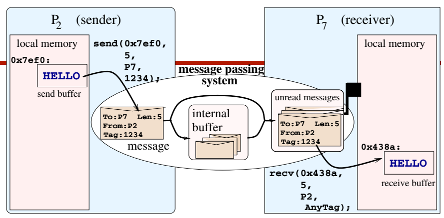
```
void MPI::Init( int &argc, char **&argv );
void MPI::Init( );

void MPI::Finalize( );

void MPI::Comm::Send( void *sbuf, int count,
                                            const Datatype& datatype,
                                                   int dest, int tag );

void int MPI::Comm::Recv( void *dbuf, int count,
                                            const Datatype& datatype,
                             int source, int tag, Status& status );

int MPI::Comm::Get_size( );

int MPI::Comm::Get_rank( );
```

Remark: The C++ API is deprecated in MPI 2.2 (2009), may disappear with MPI 3.0

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

Symbolic constants encode 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., a send or receive routine) is called

```
// ... write array A
MPI_Isend(..., A, ...);
// ... no overwriting of the
// ... sent part of A here!
MPI_Wait(...);
// ... can write over A here
```

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

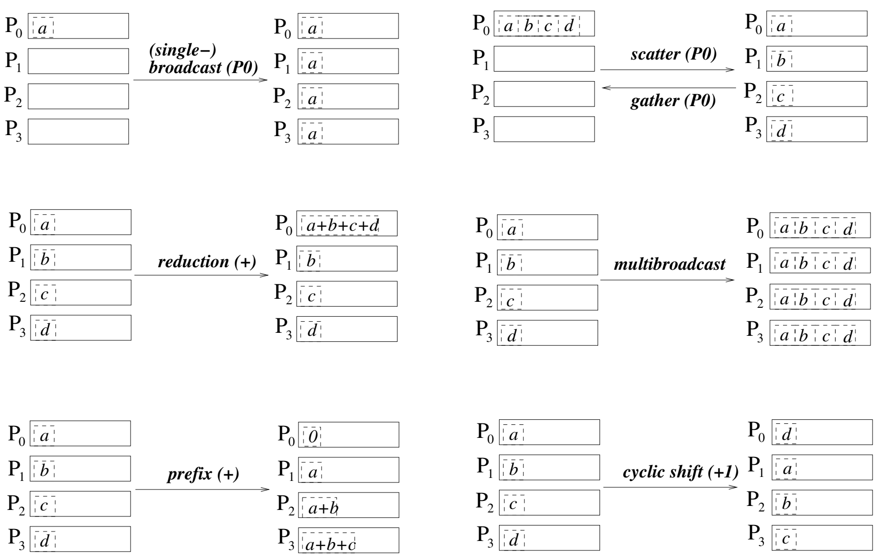
In MPI, the mode is controlled by a prefix (none, `S`, `B`) of the send operation.

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

Collective communication operations



MPI - Collective communication operations: Broadcast, Reduction

Single-Broadcast:

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

Reduction:

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

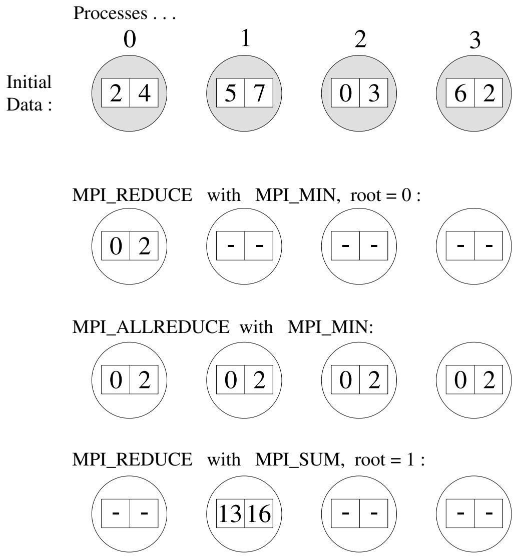
with predefined $op \in$ { MPI_SUM, MPI_MAX, ... }
or user-defined by MPI_Op_Create.

MPI_Allreduce

Barrier synchronization:

int MPI_Barrier(MPI_Comm *comm*);

MPI - Collective communication operations (3): Reductions

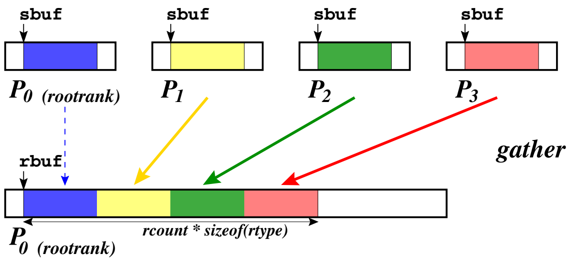


elementwise reductions

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

MPI - Collective communication operations: Scatter, Gather

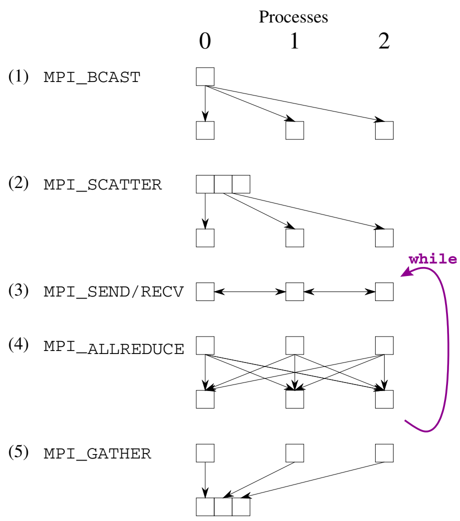
```
int MPI_Scatter( void *sbuf, int scount, MPI_datatype stype,
                 void *rbuf, int rcount, MPI_datatype rtype,
                 int rootrank, MPI_Comm comm );

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

Example: 1D Finite Differences with collective communication

```
main(int argc, char *argv[]) {
  MPI_Comm com = MPI_COMM_WORLD;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(com, &np);
  MPI_Comm_rank(com, &me);
  if (me == 0) {                  /* Read problem size at process 0 */
    read_problem_size(&size);
    buff[0] = size;
  }
  /* Global broadcast propagates this data to all processes */
  MPI_Bcast(buff, 1, MPI_INT, 0, com);
  /* Extract problem size from buff; allocate space for local data */
  lsize = buff[0]/np;
  local = malloc(lsize+2);
  /* Read input data at process 0; then distribute to processes */
  if (me == 0) { work = malloc(size); read_array(work); }
  MPI_Scatter(work, lsize, MPI_FLOAT, local+1, lsize,
              MPI_FLOAT, 0, com);
  lnbr = (me+np-1)%np;            /* Determine my neighbors in ring */
  rnbr = (me+1)%np;
  globalerr = 99999.0;
  while (globalerr > 0.1) {       /* Repeat until termination */
    /* Exchange boundary values with neighbors */
    ls = local+lsize;
    MPI_Send(local+2, 1, MPI_FLOAT, lnbr, 10, com);
    MPI_Recv(local+1, 1, MPI_FLOAT, rnbr, 10, com, &status);
    MPI_Send(ls-2, 1, MPI_FLOAT, rnbr, 20, com);
    MPI_Recv(ls-1, 1, MPI_FLOAT, lnbr, 20, com, &status);
    compute(local);
    localerr = maxerror(local); /* Determine local error */
    /* Find maximum local error, and replicate in each process */
    MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT,
                  MPI_MAX, com);
  }
  /* Collect results at process 0 */
  MPI_Gather(local, lsize, MPI_FLOAT, work, size,
             MPI_FLOAT, 0, com);
  if (me == 0) { write_array(work); free(work); }
  MPI_Finalize();
}
```



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

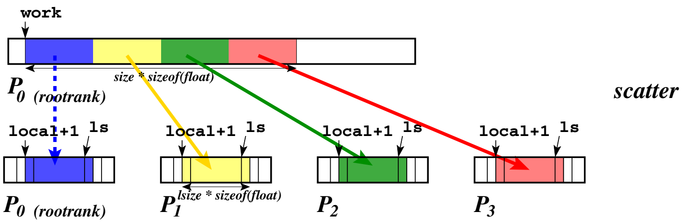
Program 8.5 : Outline of an MPI finite difference algorithm

Example, detailed look (1)

```
main ( int argc, char *argv[] )
{
  int np, me, size, lsize, buff[1];
  float *local, *work, localerr, globalerr;

  MPI_Comm com = MPI_COMM_WORLD;
  MPI_Init( &argc, &argv );
  MPI_Comm_size( com, &np );
  MPI_Comm_rank( com, &me );

  if (me == 0) { // read problem size at process 0:
     read_problem_size( &size );
     buff[0] = size;
  }
  // Single-Broadcast of size from P0 to P1...P(np-1):
  MPI_Bcast( buff, 1, MPI_INT, 0, com );
  // Extract problem size from buff; allocate space:
  lsize = buff[0] / np;   // local problem size
  local = (float *) malloc ( (lsize+2) * sizeof(float) );
  ...
```

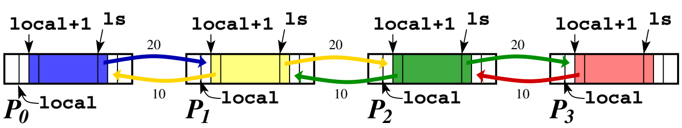

Example, detailed look (2)

```
  ...
  if (me == 0) { // read input data:
     work = (float *) malloc( size * sizeof(float) );
     read_array( work, size );
  }
  // Distribute work's contents from P0 across all processes:
  MPI_Scatter( work, lsize, MPI_FLOAT,
               local+1, lsize, MPI_FLOAT, 0, com );
  lnbr = (me + np - 1) % np;   // left neighbor rank
  rnbr = (me + 1) % np;        // right neighbor rank
  ls = local + lsize;     // points to my last local element
```



Example, detailed look (3)

```
  ...
  globalerr = 99999.0;  // large value
  while (globalerr > 0.1) {
     // Exchange boundary values with neighbors:
     MPI_Send( local+1, 1, MPI_FLOAT, lnbr, 10, com );
     MPI_Recv( ls+1, 1, MPI_FLOAT, rnbr, 10, com, &status );
     MPI_Send( ls, 1, MPI_FLOAT, rnbr, 20, com );
     MPI_Recv( local, 1, MPI_FLOAT, lnbr, 20, com, &status );
     compute( local, lsize );    // update my inner elements
     localerr = maxerror( local );
     MPI_Allreduce( &localerr, &globalerr, 1, MPI_FLOAT,
                    MPI_MAX, com );
  }
  ...
```



Questions on the example (for self-evaluation)

Draw a figure that shows in detail which elements are exchanged between neighbor processors in the `send` and `recv` operations.

Question 1:

Is it necessary (for correct execution) to use *different* tags (10, 20) in the exchange phase?

Question 2:

How could this program be improved in efficiency?

(*Hint:* try to overlap communication phases with local computation where possible.)

Question 3:

Try to construct a deadlock situation by reordering the `send` / `recv` operations.

Virtual topologies in MPI

Example: arrange 12 processors in 3×4 grid

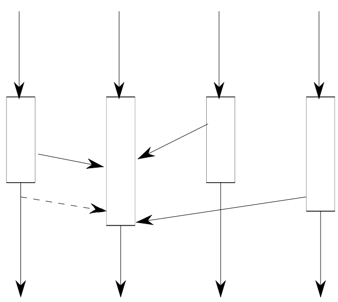
0 (0,0)	**1** (0,1)	**2** (0,2)	**3** (0,3)
4 (1,0)	**5** (1,1)	**6** (1,2)	**7** (1,3)
8 (2,0)	**9** (2,1)	**10** (2,2)	**11** (2,3)

```
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
...
```

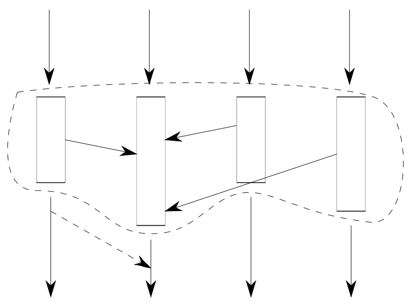
```
...
coo[0]=i; coo[1]=j;

// 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.



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)

→ 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.

→ supports sequential composition

Furthermore:

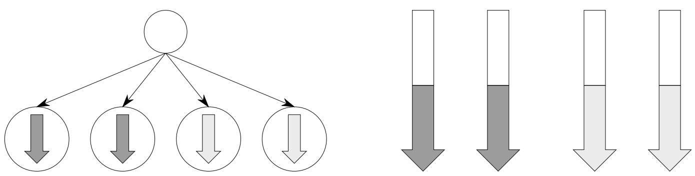
`MPI_COMM_SPLIT` (*comm*, *color*, *key*, *newcomm*)
create a new communicator for a *subset of a group* of processes

`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*

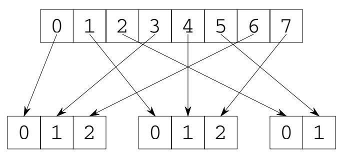
Communicators for splitting process sets

`MPI_COMM_SPLIT` (*comm*, *color*, *key*, *newcomm*)
used for parallel composition of process groups.
A fixed set of processes changes character.



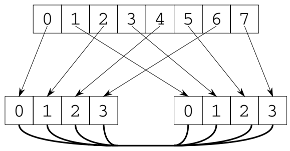
Example:

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



Source: Ian Foster: Designing and building parallel programs.

Communicators for communicating between process groups

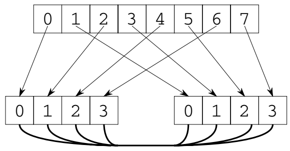


An *intercommunicator* connects two process groups

- 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 (MPI-2)

One-sided communication / Remote memory access (RMA)

- Each MPI process sets up a separate "thread" for servicing RMA requests
- Limited to a fixed memory block (RMA Window)

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* ($\rightarrow$ 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
- Dynamic 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

Literature on MPI

MPI-Forum: http://www.mpi-forum.org/
Official MPI standard documents
MPI 3.0 expected in the near future

Book series:
Gropp, Lusk, Skjellum: *Using MPI*. Second edition, MIT press, 1999
Gropp, Lusk, Thakur: *Using MPI-2*. MIT press, 1999

Chapter 8 of
Foster: Designing and Building Parallel Programs, Addison-Wesley 1995