Parallel Programming and MPI

CS717, Fall ’01

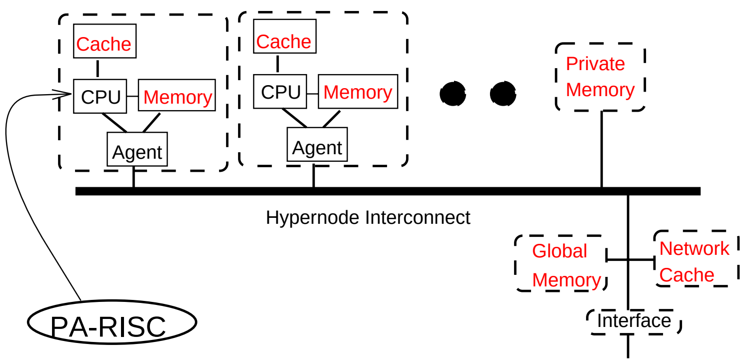
Tutorial on MPI: The Message-Passing Interface

William Gropp

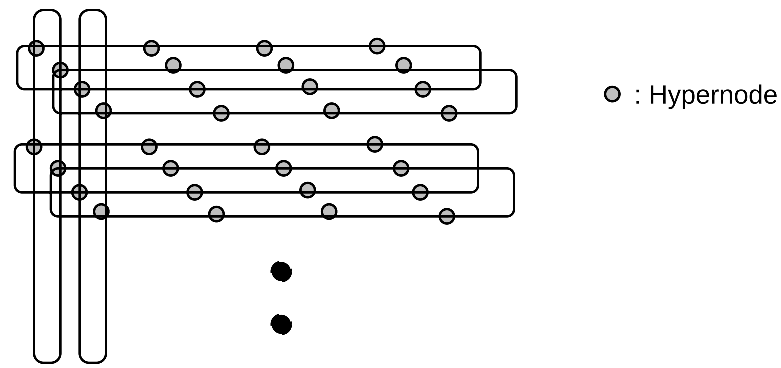
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
gropp@mcs.anl.gov

Architectural Issues in Parallel Processing

Convex Exemplar Architecture:



Hypernode



Network of hypernodes

Memory latencies:

Processor cache 10 ns

CPU private memory 500 ns

Hypernode private memory 500 ns

Network cache 500 ns

Interhypernode shared memory 2 microsec

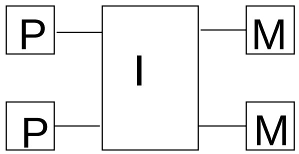
Within hypernode: SMP

Across hypernodes: NUMA

Locality of reference is extremely important!!

Physical Organization

- Uniform memory access (UMA) machines



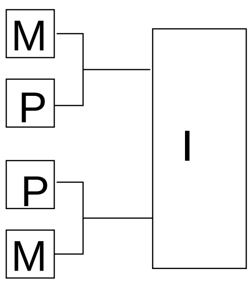
All memory is equally far away from all processors.

Early parallel processors like NYU Ultracomputer

Problem: why go across network for instructions? read-only data?

what about caches?

- Non-uniform memory access (NUMA) machines:



Access to local memory is usually 10-1000 times faster than access to non-local memory

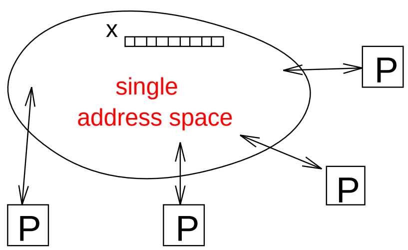
Static and dynamic locality of reference are critical for high performance.

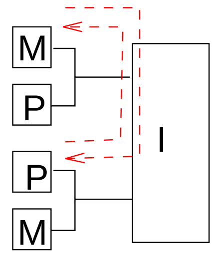
Compiler support? Architectural support?

Bus-based symmetric multiprocessors (SMP's): combine both aspects

Logical Organization

- Shared Memory Model

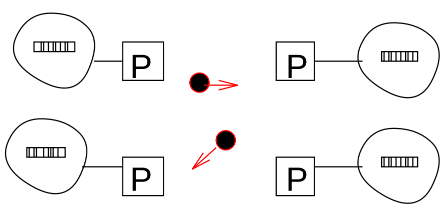


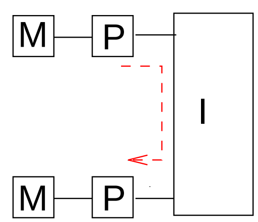


(conceptual picture)

- hardware/systems software provide single address space model to applications programmer
- some systems: distinguish between local and remote references
- communication between processors: read/write shared memory locations: put get

- Distributed Memory Model (Message Passing)





(conceptual picture)

- each processor has its own address space
- communication between processors: messages (like e-mail)
- basic message-passing commands: send receive

Key difference: In SMM, P1 can access remote memory locations w/o prearranged participation of application program on remote processor

Types of parallel computing

All use different data for each worker

Data-parallel Same operations on different data. Also called SIMD

SPMD Same program, different data

MIMD Different programs, different data

SPMD and MIMD are essentially the same because any MIMD can be made SPMD

SIMD is also equivalent, but in a less practical sense.

MPI is primarily for SPMD/MIMD. HPF is an example of a SIMD interface.

Communicating with other processes

Data must be exchanged with other workers

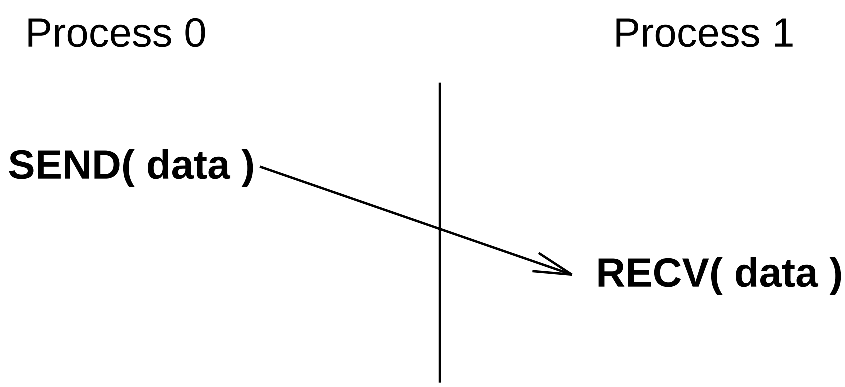
- Cooperative — all parties agree to transfer data
- One sided — one worker performs transfer of data

Cooperative operations

Message-passing is an approach that makes the exchange of data cooperative.

Data must both be explicitly sent and received.

An advantage is that any change in the *receiver's* memory is made with the receiver's participation.



So far, we have looked at point-to-point communication

Collective communication:

- patterns of group communication that can be implemented more efficiently than through long sequences of send's and receive's

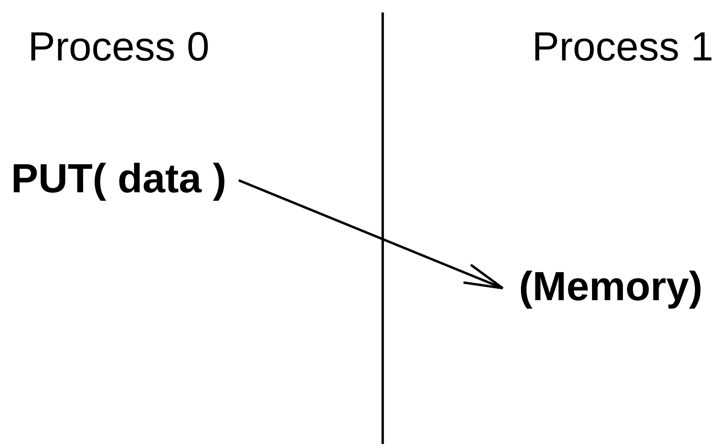
- important ones:

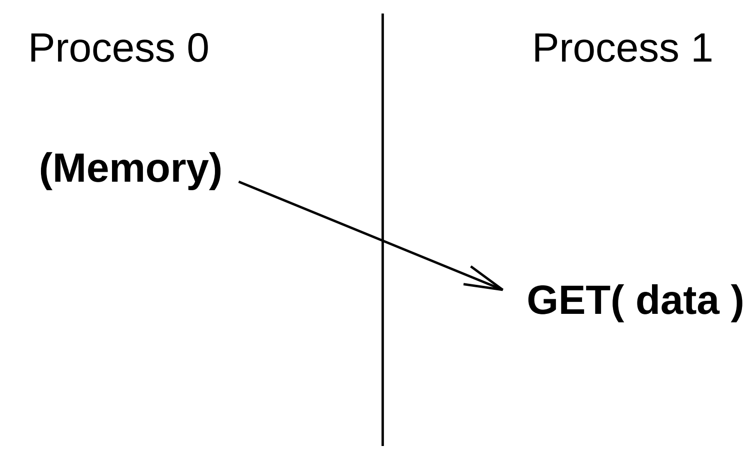
  - one-to-all broadcast
    (eg. A*x implemented by rowwise distribution: all processors need x)
  - all-to-one reduction
    (eg. adding a set of numbers distributed across all processors)
  - all-to-all broadcast
    every processor sends a piece of data to every other processor
  - one-to-all personalized communication
    one processor sends a different piece of data to all other processors
  - all-to-all personalized communication
    each processor does a one-to-all communication

One-sided operations

One-sided operations between parallel processes include remote memory reads and writes.

An advantage is that data can be accessed without waiting for another process





What is MPI?

- A *message-passing library specification*
  - message-passing model
  - not a compiler specification
  - not a specific product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to permit (unleash?) the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Features of MPI

- General
  - Communicators combine context and group for message security
  - Thread safety
- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

Features of MPI (cont.)

- Application-oriented process topologies
  - Built-in support for grids and graphs (uses groups)
- Profiling
  - Hooks allow users to intercept MPI calls to install their own tools
- Environmental
  - inquiry
  - error control

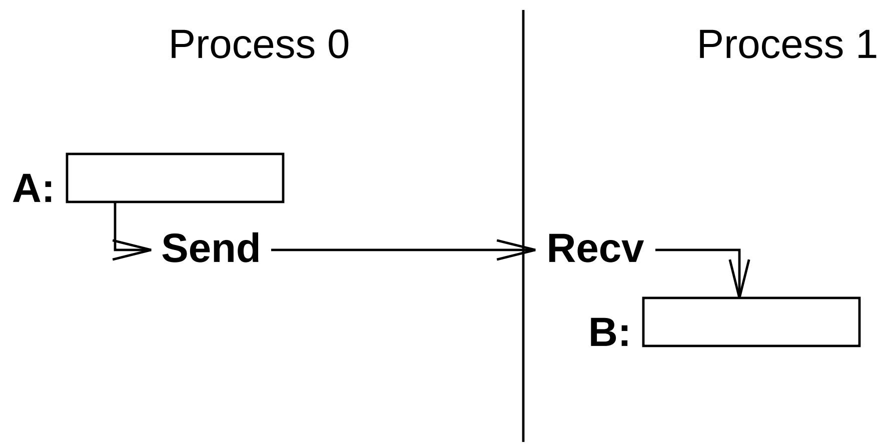
Features not in MPI

- Non-message-passing concepts not included:
  - process management
  - remote memory transfers
  - active messages
  - threads
  - virtual shared memory
- MPI does not address these issues, but has tried to remain compatible with these ideas (e.g. thread safety as a goal, intercommunicators)

A simple program

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char **argv;
{
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf( "Hello world! I'm %d of %d\n",
        rank, size );
MPI_Finalize();
return 0;
}
```

Sending and Receiving messages



Questions:

- To whom is data sent?
- What is sent?
- How does the receiver identify it?

"Primitive"

~~Current~~ Message-Passing

- A typical blocking send looks like

```
send( dest, type, address, length )
```

  where

  - **dest** is an integer identifier representing the process to receive the message.
  - **type** is a nonnegative integer that the destination can use to selectively screen messages.
  - (**address, length**) describes a contiguous area in memory containing the message to be sent.

  and

- A typical global operation looks like:

```
broadcast( type, address, length )
```

- All of these specifications are a good match to hardware, easy to understand, but too inflexible.

Limitations of Primitive Message-Passing

- Data is not always contiguous
  - data accessed by “stride”.
- heterogeneous environments
  - word size
  - endien
- “Classes” of message
  - Library A: p_1 sends int to p_2.
  - Library B: p_2 recvs int from p_1.
  - type doesn’t map to “semantics”.
- broadcast to whom?
  - divide and conquer – communicate within partition
  - matrix computations – communicate within rows and columns

Generalizing the Buffer Description

- Specified in MPI by *starting address*, *datatype*, and *count*, where datatype is:
  - elementary (all C and Fortran datatypes)
  - contiguous array of datatypes
  - strided blocks of datatypes
  - indexed array of blocks of datatypes
  - general structure
- Datatypes are constructed recursively.
- Specifications of elementary datatypes allows heterogeneous communication.
- Elimination of length in favor of count is clearer.
- Specifying application-oriented layout of data allows maximal use of special hardware.

Generalizing the Type

- A single type field is too constraining. Often overloaded to provide needed flexibility.
- Problems:
  - under user control
  - wild cards allowed (MPI_ANY_TAG)
  - library use conflicts with user and with other libraries

Sample Program using Library Calls

`Sub1` and `Sub2` are from different libraries.

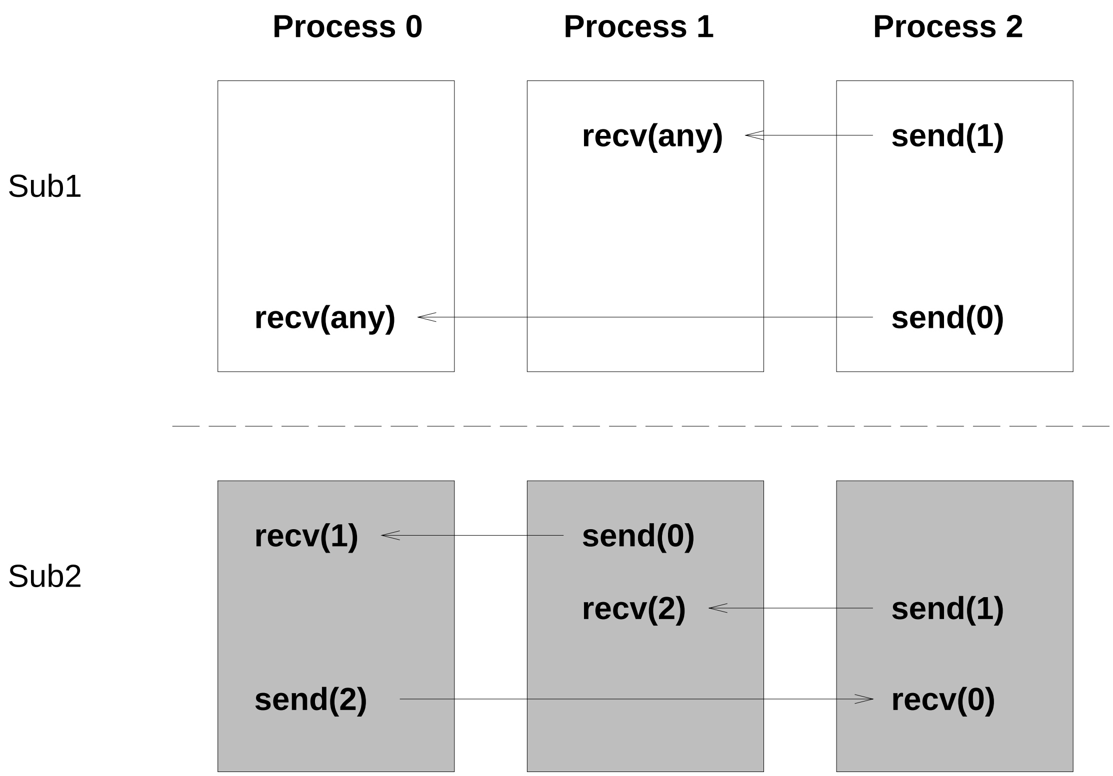
```
Sub1();
Sub2();
```

`Sub1a` and `Sub1b` are from the same library

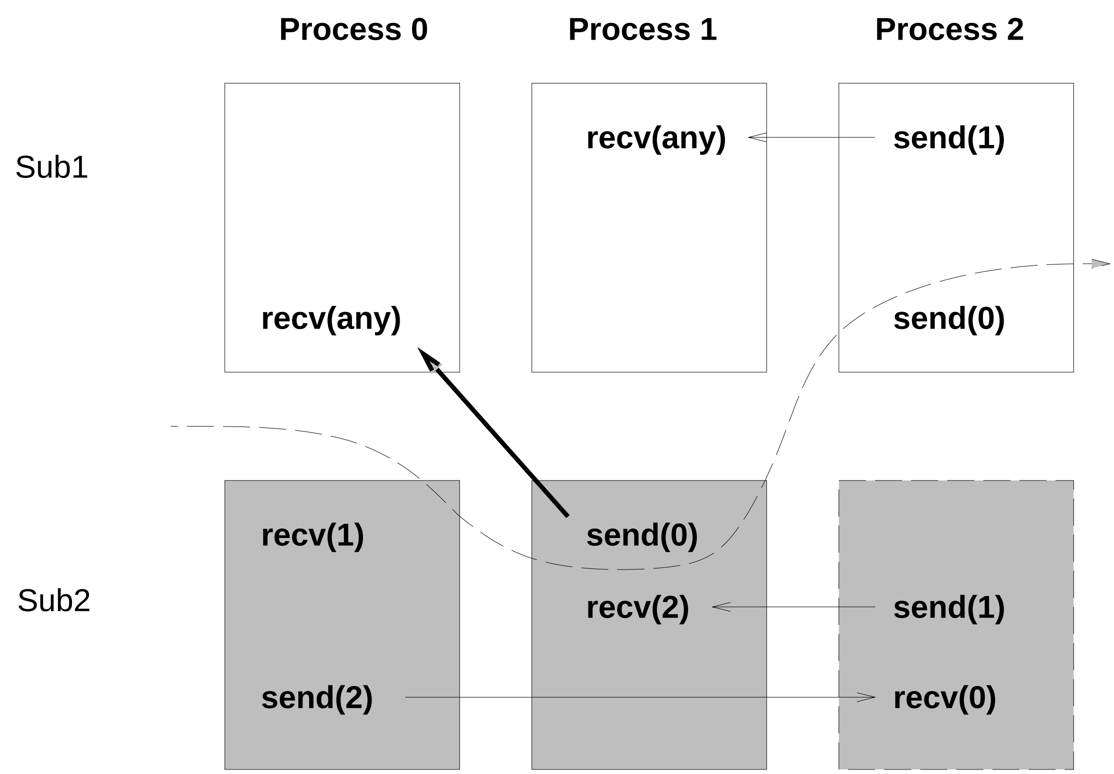
```
Sub1a();
Sub2();
Sub1b();
```

Thanks to Marc Snir for the following four examples

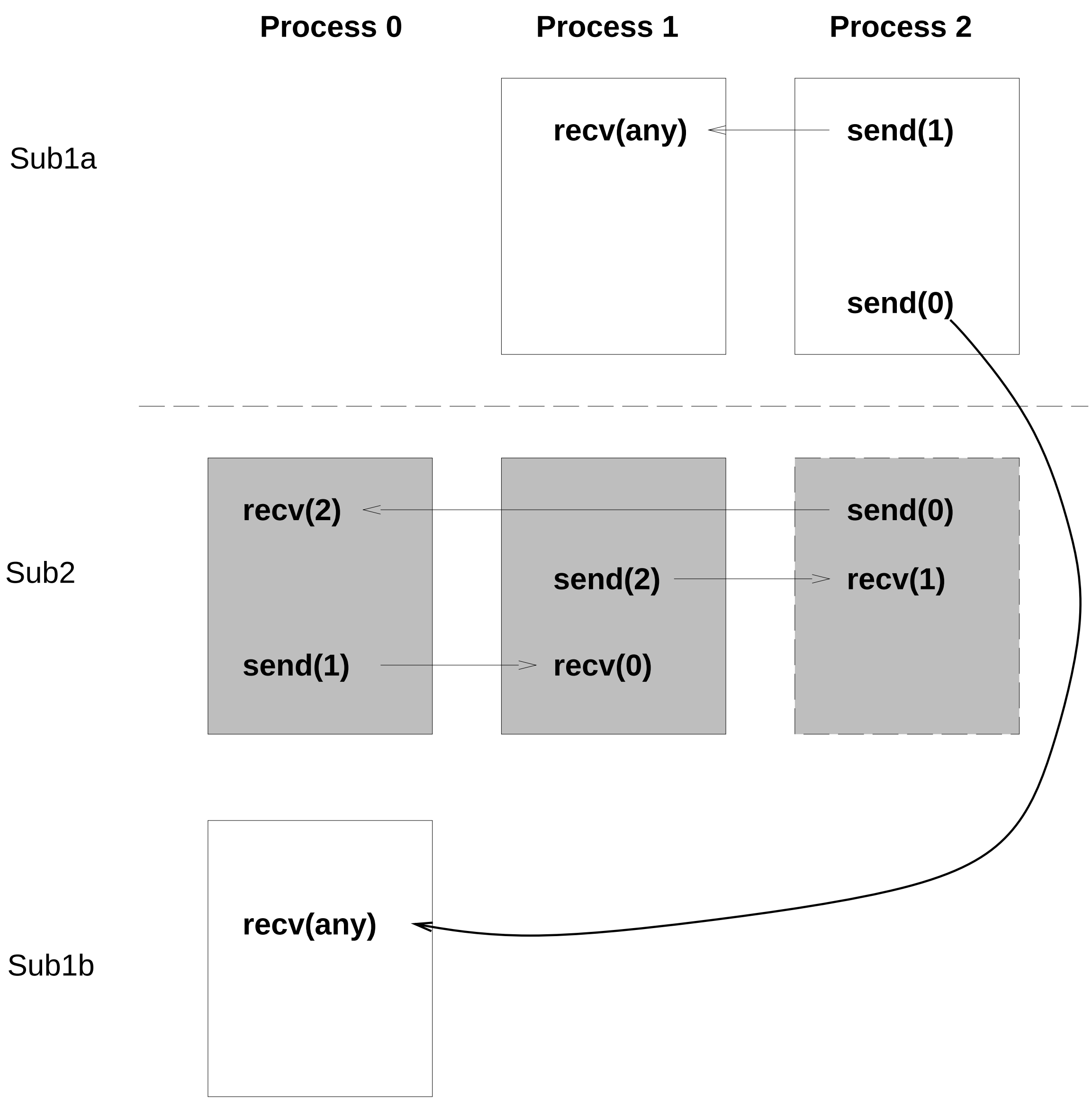
Correct Execution of Library Calls



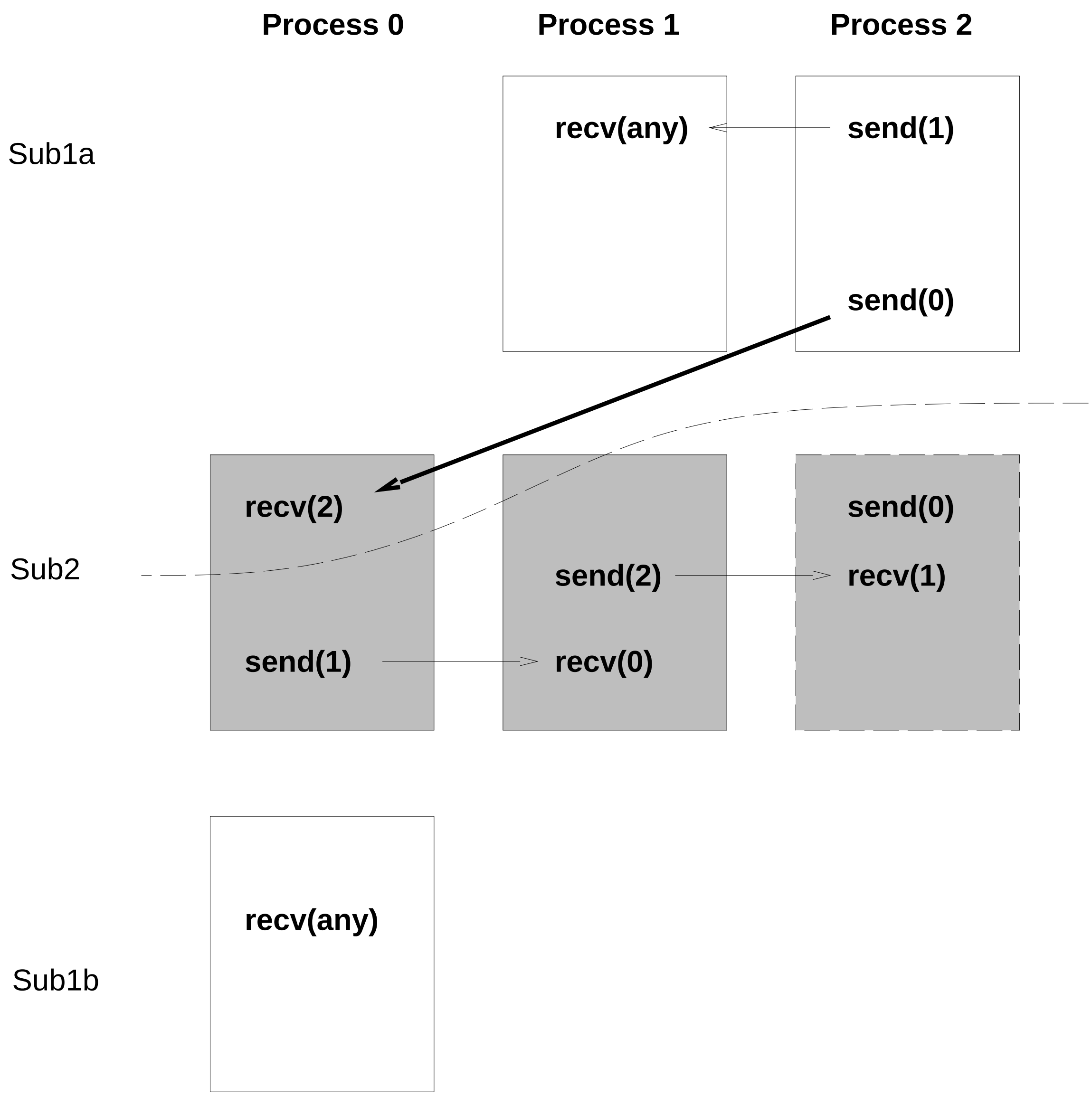
Incorrect Execution of Library Calls



Correct Execution of Library Calls with Pending Communcication



Incorrect Execution of Library Calls with Pending Communication



Solution to the type problem

- A separate communication *context* for each family of messages, used for queueing and matching. (This has often been simulated in the past by overloading the tag field.)

- No wild cards allowed, for security

- Allocated by the system, for security

- Types (*tags*, in MPI) retained for normal use (wild cards OK)

Delimiting Scope of Communication

- Separate groups of processes working on subproblems
  - Merging of process name space interferes with modularity
  - "Local" process identifiers desirable
- Parallel invocation of parallel libraries
  - Messages from application must be kept separate from messages internal to library.
  - Knowledge of library message types interferes with modularity.
  - Synchronizing before and after library calls is undesirable.

Generalizing the Process Identifier

- Collective operations typically operated on all processes (although some systems provide subgroups).
- This is too restrictive (e.g., need minimum over a column or a sum across a row, of processes)
- MPI provides *groups* of processes
  - initial "all" group
  - group management routines (build, delete groups)
- All communication (not just collective operations) takes place in groups.
- A group and a context are combined in a *communicator*.
- Source/destination in send/receive operations refer to *rank* in group associated with a given communicator. `MPI_ANY_SOURCE` permitted in a receive.

MPI Basic Send/Receive

Thus the basic (blocking) send has become:

```
MPI_Send( start, count, datatype, dest, tag,
              comm )
```

and the receive:

```
MPI_Recv(start, count, datatype, source, tag,
              comm, status)
```

The source, tag, and count of the message actually received can be retrieved from `status`.

Two simple collective operations:

```
MPI_Bcast(start, count, datatype, root, comm)
MPI_Reduce(start, result, count, datatype,
           operation, root, comm)
```

Getting information about a message

```
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG;
... status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
```

`MPI_TAG` and `MPI_SOURCE` primarily of use when `MPI_ANY_TAG` and/or `MPI_ANY_SOURCE` in the receive.

`MPI_Get_count` may be used to determine how much data of a particular type was received.

Simple Fortran example

```
      program main
      include 'mpif.h'

      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
      integer status(MPI_STATUS_SIZE)
      double precision data(100)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src = 0
C
      if (rank .eq. src) then
         to     = dest
         count  = 10
         tag    = 2001
         do 10 i=1, 10
 10         data(i) = i
         call MPI_SEND( data, count, MPI_DOUBLE_PRECISION, to,
     +                   tag, MPI_COMM_WORLD, ierr )
      else if (rank .eq. dest) then
         tag   = MPI_ANY_TAG
         count = 10
         from  = MPI_ANY_SOURCE
         call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
     +                  tag, MPI_COMM_WORLD, status, ierr )
```

Simple Fortran example (cont.)

```
         call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION,
     +                       st_count, ierr )
         st_source = status(MPI_SOURCE)
         st_tag    = status(MPI_TAG)
C
         print *, 'Status info: source = ', st_source,
     +            ' tag = ', st_tag, ' count = ', st_count
         print *, rank, ' received', (data(i),i=1,10)
      endif

      call MPI_FINALIZE( ierr )
      end
```

FIFO revisited

- MPI guarantees that messages are between “matching” sends and receives are delivered in order.
- Does this mean that a program always receives messages in order?

FIFO revisited (cont.)

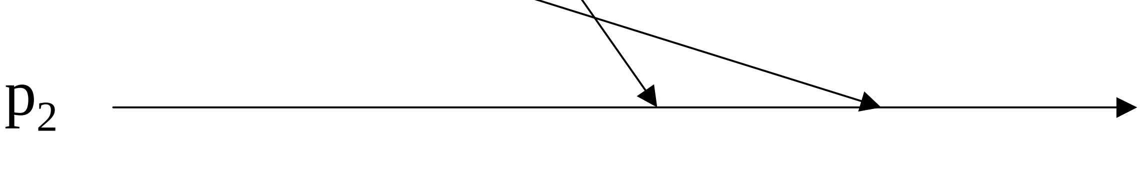
NO! For instance -

<u>Processors p_1:</u>

```
MPI_ISEND(data,count,MPI_INT,p2,tag1,
   MPI_COMM_WORLD);
MPI_ISEND(data,count,MPI_INT,p2,tag2,
   MPI_COMM_WORLD);
```

<u>Processor p_2:</u>

```
MPI_RECV(data,count,MPI_INT,p1,tag2,
   MPI_COMM_WORLD);
MPI_RECV(data,count,MPI_INT,p1,tag1,
   MPI_COMM_WORLD);
```



Broadcast and Reduction

The routine `MPI_Bcast` sends data from one process to all others.

The routine `MPI_Reduce` combines data from all processes (by adding them in this case), and returning the result to a single process.

C example: PI

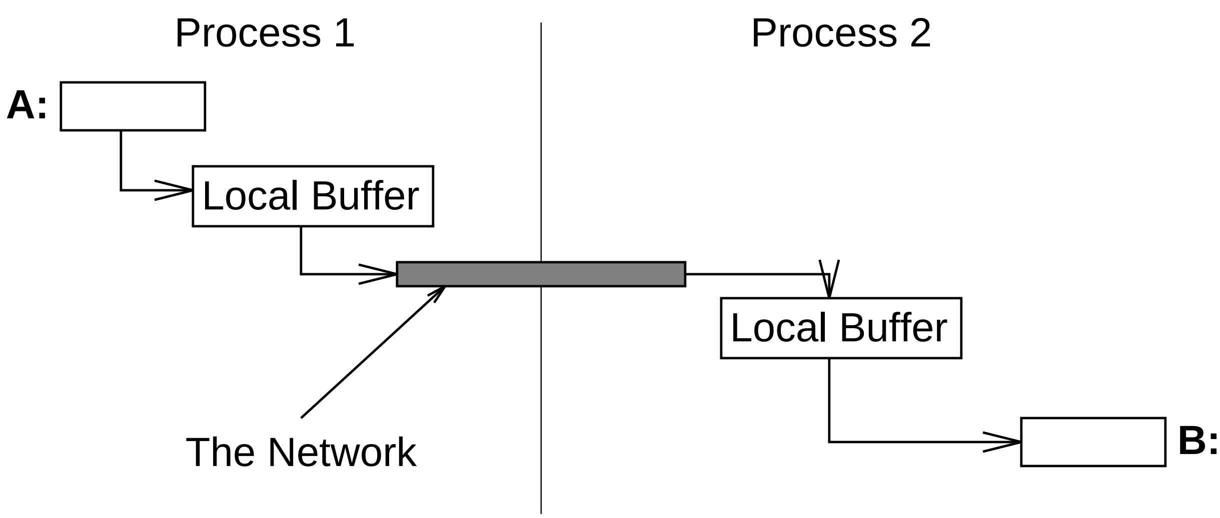
```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

C example (cont.)

```
  while (!done)
  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```

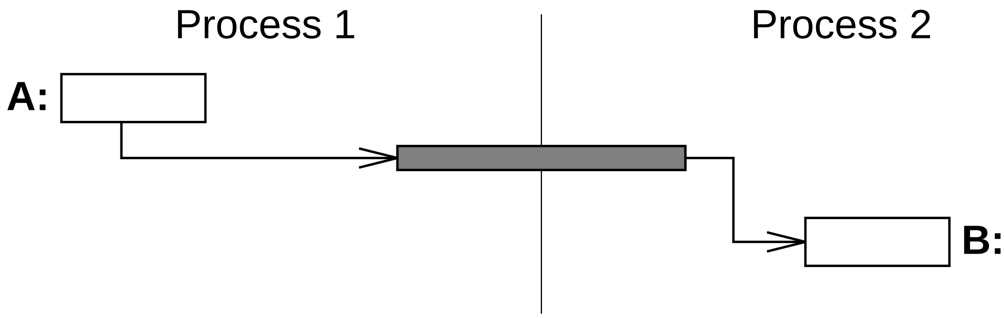
Buffering issues

Where does data go when you send it? One possibility is:



Better buffering

This is not very efficient. There are three copies in addition to the exchange of data between processes. We prefer



But this requires that either that `MPI_Send` not return until the data has been delivered *or* that we allow a send operation to return before completing the transfer. In this case, we need to test for completion later.

Blocking and Non-Blocking communication

- So far we have used **blocking** communication:
  - MPI_Send does not complete until buffer is empty (available for reuse).
  - MPI_Recv does not complete until buffer is full (available for use).
- Simple, but can be "unsafe":

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

  Completion depends in general on size of message and amount of system buffering.

Send works for small enough messages but fails when messages get too large. Too large ranges from zero bytes to 100's of Megabytes.

Some Solutions to the "Unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	Recv(0)
Recv(1)	Send(0)

- Supply receive buffer at same time as send, with `MPI_Sendrecv`:

Process 0	Process 1
Sendrecv(1)	Sendrecv(0)

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	Isend(0)
Irecv(1)	Irecv(0)
Waitall	Waitall

- Use `MPI_Bsend`

MPI's Non-Blocking Operations

Non-blocking operations return (immediately) "request handles" that can be waited on and queried:

- `MPI_Isend(start, count, datatype, dest, tag, comm, request)`
- `MPI_Irecv(start, count, datatype, dest, tag, comm, request)`
- `MPI_Wait(request, status)`

One can also test without waiting: `MPI_Test( request, flag, status)`

Multiple completions

It is often desirable to wait on multiple requests. An example is a master/slave program, where the master waits for one or more slaves to send it a message.

- `MPI_Waitall(count, array_of_requests, array_of_statuses)`
- `MPI_Waitany(count, array_of_requests, index, status)`
- `MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`

There are corresponding versions of test for each of these.

The `MPI_WAITSOME` *and* `MPI_TESTSOME` *may be used to implement master/slave algorithms that provide fair access to the master by the slaves.*

More on nonblocking communication

In applications where the time to send data between processes is large, it is often helpful to cause communication and computation to overlap. This can easily be done with MPI's non-blocking routines.

For example, in a 2-D finite difference mesh, moving data needed for the boundaries can be done at the same time as computation on the interior.

```
MPI_Irecv( ... each ghost edge ... );
MPI_Isend( ... data for each ghost edge ... );
... compute on interior
while (still some uncompleted requests) {
    MPI_Waitany( ... requests ... )
    if (request is a receive)
        ... compute on that edge ...
    }
```

Note that we call `MPI_Waitany` several times. This exploits the fact that after a request is satisfied, it is set to `MPI_REQUEST_NULL`, and that this is a valid request object to the wait and test routines.

Communication Modes

MPI provides mulitple *modes* for sending messages:

- Synchronous mode (`MPI_Ssend`): the send does not complete until a matching receive has begun. (Unsafe programs become incorrect and usually deadlock within an `MPI_Ssend`.)

- Buffered mode (`MPI_Bsend`): the user supplies the buffer to system for its use. (User supplies enough memory to make unsafe program safe).

- Ready mode (`MPI_Rsend`): user guarantees that matching receive has been posted.
  - allows access to fast protocols
  - undefined behavior if the matching receive is not posted

Non-blocking versions:
`MPI_Issend`, `MPI_Irsend`, `MPI_Ibsend`

Note that an `MPI_Recv` may receive messages sent with *any* send mode.

Buffered Send

MPI provides a send routine that may be used when `MPI_Isend` is awkward to use (e.g., lots of small messages).

`MPI_Bsend` makes use of a *user-provided* buffer to save any messages that can not be immediately sent.

```
int  bufsize;
char *buf = malloc(bufsize);
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( ... same as MPI_Send ... );
...
MPI_Buffer_detach( &buf, &bufsize );
```

The `MPI_Buffer_detach` call does not complete until all messages are sent.

The performance of `MPI_Bsend` *depends on the implementation of MPI and may also depend on the size of the message. For example, making a message one byte longer may cause a significant drop in performance.*

Reusing the same buffer

Consider a loop

```
MPI_Buffer_attach( buf, bufsize );
while (!done) {
    ...
    MPI_Bsend( ... );
    }
```

where the `buf` is large enough to hold the message in the `MPI_Bsend`. This code may *fail* because the

```
{
void *buf; int bufsize;
MPI_Buffer_detach( &buf, &bufsize );
MPI_Buffer_attach( buf, bufsize );
}
```

Other Point-to-Point Features

- MPI_SENDRECV, MPI_SENDRECV_REPLACE
- MPI_CANCEL
- Persistent communication requests

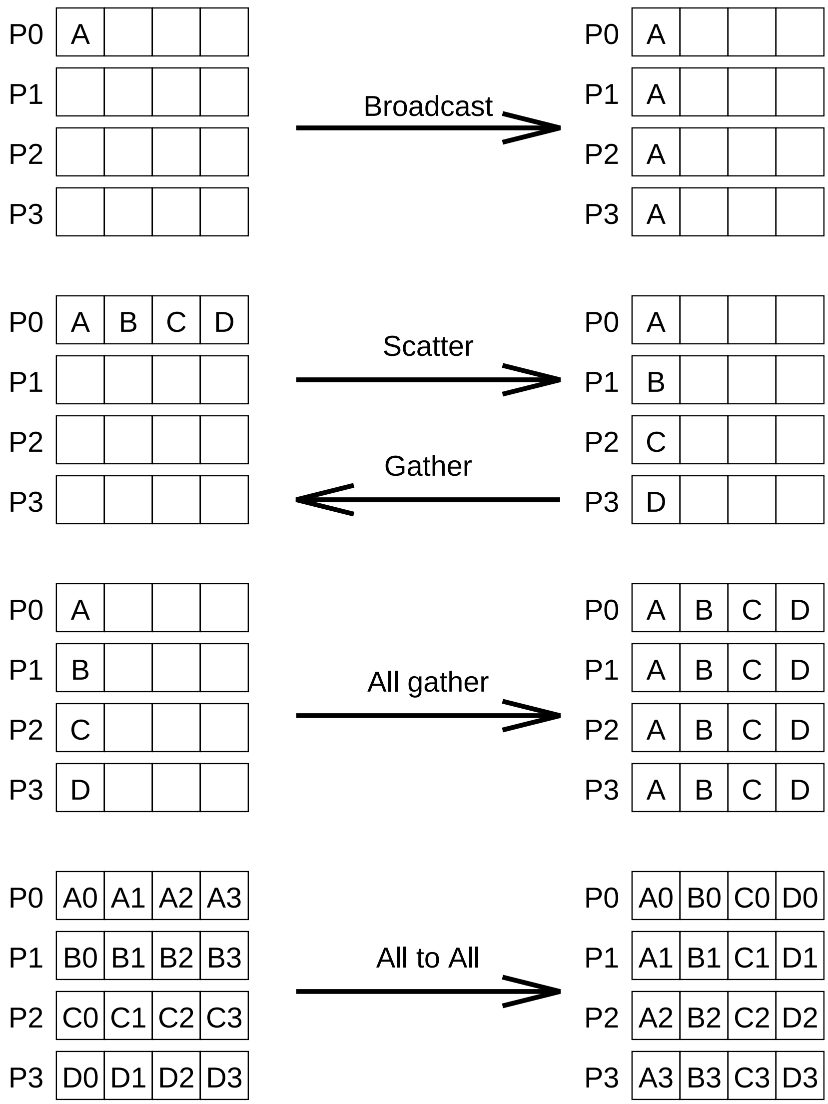
Collective Communications in MPI

- Communication is coordinated among a group of processes.
- Groups can be constructed “by hand” with MPI group-manipulation routines or by using MPI topology-definition routines.
- Message tags are not used. Different communicators are used instead.
- No non-blocking collective operations.
- Three classes of collective operations:
  - synchronization
  - data movement
  - collective computation

Synchronization

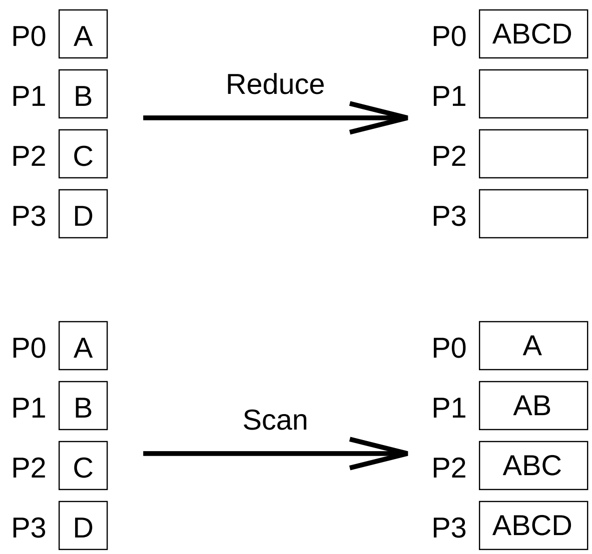
- `MPI_Barrier(comm)`
- Function blocks untill all processes in `comm` call it.

Available Collective Patterns



Schematic representation of collective data movement in MPI

Available Collective Computation Patterns



Schematic representation of collective data movement in MPI

MPI Collective Routines

- Many routines:

Allgather	Allgatherv	Allreduce
Alltoall	Alltoallv	Bcast
Gather	Gatherv	Reduce
ReduceScatter	Scan	Scatter
Scatterv		

- `All` versions deliver results to all participating processes.

- `V` versions allow the chunks to have different sizes.

- Allreduce, Reduce, ReduceScatter, and Scan take both built-in and user-defined combination functions.

Defining groups

All MPI communication is relative to a *communicator* which contains a *context* and a *group*. The group is just a set of processes.

Private communicators

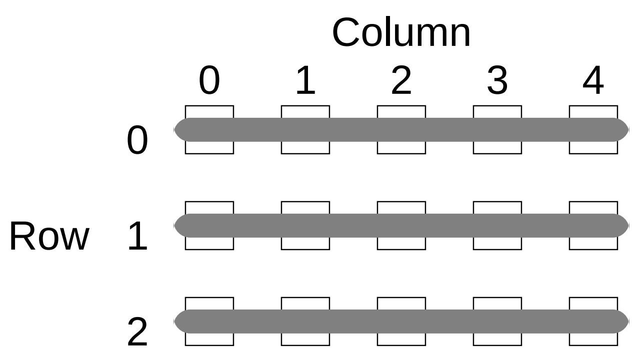
One of the first thing that a library should normally do is create private communicator. This allows the library to send and receive messages that are known only to the library.

```
MPI_Comm_dup( old_comm, &new_comm );
```

Subdividing a communicator

The easiest way to create communicators with new groups is with `MPI_COMM_SPLIT`.

For example, to form groups of rows of processes



use

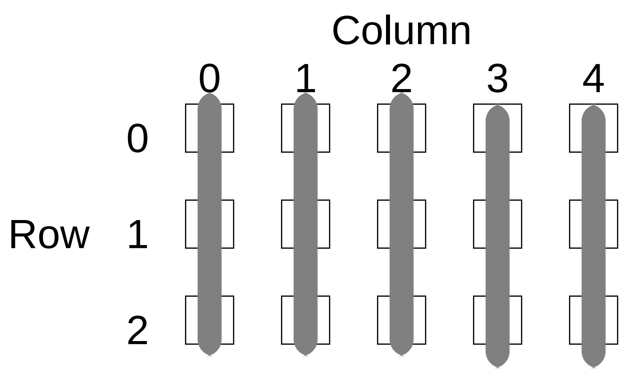
```
MPI_Comm_split( oldcomm, row, 0, &newcomm );
```

To maintain the order by rank, use

```
MPI_Comm_rank( oldcomm, &rank );
MPI_Comm_split( oldcomm, row, rank, &newcomm );
```

Subdividing (con't)

Similarly, to form groups of columns,



use

```
MPI_Comm_split( oldcomm, column, 0, &newcomm2 );
```

To maintain the order by rank, use

```
MPI_Comm_rank( oldcomm, &rank );
MPI_Comm_split( oldcomm, column, rank, &newcomm2 );
```

Manipulating Groups

Another way to create a communicator with specific members is to use `MPI_Comm_create`.

```
MPI_Comm_create( oldcomm, group, &newcomm );
```

The group can be created in many ways:

Creating Groups

All group creation routines create a group by specifying the members to take from an existing group.

- `MPI_Group_incl` specifies specific members
- `MPI_Group_excl` excludes specific members
- `MPI_Group_range_incl` and `MPI_Group_range_excl` use ranges of members
- `MPI_Group_union` and `MPI_Group_intersection` creates a new group from two existing groups.

To get an existing group, use

```
MPI_Comm_group( oldcomm, &group );
```

Free a group with

```
MPI_Group_free( &group );
```

Datatypes and Heterogenity

MPI datatypes have two main purposes

- Heterogenity — parallel programs between different processors
- Noncontiguous data — structures, vectors with non-unit stride, etc.

Basic datatype, corresponding to the underlying language, are predefined.

The user can construct new datatypes at run time; these are called *derived datatypes*.

Datatypes in MPI

Elementary: Language-defined types (e.g., MPI_INT or MPI_DOUBLE_PRECISION)

Vector: Separated by constant "stride"

Contiguous: Vector with stride of one

Hvector: Vector, with stride in bytes

Indexed: Array of indices (for scatter/gather)

Hindexed: Indexed, with indices in bytes

Struct: General mixed types (for C structs etc.)

The MPI Timer

The elapsed (wall-clock) time between two points in an MPI program can be computed using `MPI_Wtime`:

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "Elapsed time is %f\n", t2 - t1 );
```

The value returned by a single call to `MPI_Wtime` has little value.

The times are local; the attribute `MPI_WTIME_IS_GLOBAL` *may be used to determine if the times are* also *synchronized with each other for all processes in* `MPI_COMM_WORLD`.

Sharable MPI Resources

- The Standard itself:
  - As a Technical report: U. of Tennessee. report
  - As postscript for ftp: at `info.mcs.anl.gov` in `pub/mpi/mpi-report.ps`.
  - As hypertext on the World Wide Web: `http://www.mcs.anl.gov/mpi`
  - As a journal article: in the Fall issue of the Journal of Supercomputing Applications
- MPI Forum discussions
  - The MPI Forum email discussions and both current and earlier versions of the Standard are available from `netlib`.
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994
  - *MPI Annotated Reference Manual*, by Otto, et al., in preparation.

Sharable MPI Resources, continued

- Newsgroup:
  - `comp.parallel.mpi`
- Mailing lists:
  - `mpi-comm@mcs.anl.gov`: the MPI Forum discussion list.
  - `mpi-impl@mcs.anl.gov`: the implementors' discussion list.
- Implementations available by `ftp`:
  - MPICH is available by anonymous `ftp` from `info.mcs.anl.gov` in the directory `pub/mpi/mpich`, file `mpich.tar.Z`.
  - LAM is available by anonymous `ftp` from `tbag.osc.edu` in the directory `pub/lam`.
  - The CHIMP version of MPI is available by anonymous `ftp` from `ftp.epcc.ed.ac.uk` in the directory `pub/chimp/release`.
- Test code repository:
  - `ftp://info.mcs.anl.gov/pub/mpi/mpi-test`

MPI-2

- The MPI Forum (with old and new participants) has begun a follow-on series of meetings.
- Goals
  - clarify existing draft
  - provide features users have requested
  - make extensions, not changes
- Major Topics being considered
  - dynamic process management
  - client/server
  - real-time extensions
  - "one-sided" communication (put/get, active messages)
  - portable access to MPI system state (for debuggers)
  - language bindings for C++ and Fortran-90
- Schedule
  - Dynamic processes, client/server by SC '95
  - MPI-2 complete by SC '96

Providing Transparent FT within MPI

1. Modify an existing MPI implementation.
2. Write a “thin” layer on top of MPI
   - Lack of FIFO properties.
   - After failure, reposting send and receive buffers.
   - No process management in MPI-1.
   - A lot of bookkeeping has to be recovered…

The MPI Objects

`MPI_Request` Handle for nonblocking communication, normally freed by MPI in a test or wait

`MPI_Datatype` MPI datatype. Free with `MPI_Type_free`.

`MPI_Op` User-defined operation. Free with `MPI_Op_free`.

`MPI_Comm` Communicator. Free with `MPI_Comm_free`.

`MPI_Group` Group of processes. Free with `MPI_Group_free`.

`MPI_Errhandler` MPI errorhandler. Free with `MPI_Errhandler_free`.