Advanced Cluster Programming with MPI

Bill Gropp

Mathematics and Computer Science Division

Argonne National Laboratory

Thanks to

- Rusty Lusk
  Rob Ross
  Rajeev Thakur

Tutorial Outline

- Background
- Tuning for Performance on Clusters
- Parallel I/O in MPI-2
- Introduction to dynamic process management in MPI-2
- MPI and threads

About This Tutorial

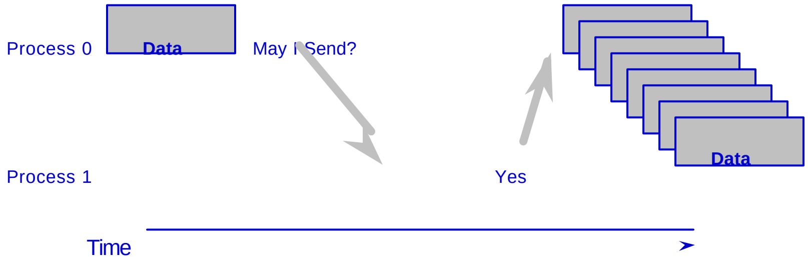
- Some of this material is taken from *Using MPI-2*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- Assumes familiarity with MPI but no deep knowledge.
- Most of the time is spent on those features most widely implemented at this time.
- Language-independent, but most examples are in C.
- Complete MPI-2 specification is on the web http://www.mpi-forum.org and in *MPI: The Complete Reference*, 2 volumes from MIT Press.

Contents of MPI-2

- Extensions to the message-passing model
  - **Parallel I/O**
  - One-sided operations
  - **Dynamic process management**
- Making MPI more robust and convenient
  - C++ and Fortran 90 bindings
  - External interfaces, handlers
  - Extended collective operations
  - Language interoperability
  - **MPI interaction with threads**

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Quick review of MPI Message passing

- Basic terms
  - nonblocking - Operation does not wait for completion
  - synchronous - Completion of send *requires* initiation (but not completion) of receive
  - ready - *Correct* send requires a matching receive
  - asynchronous - communication and computation take place simultaneously, ***not*** an MPI concept (implementations *may* use asynchronous methods)

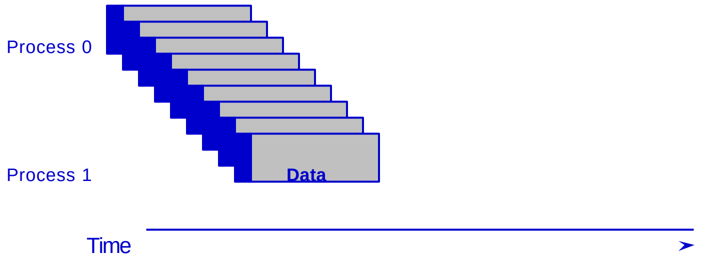
Message protocols

- Message consists of “envelope” and data
  - Envelope contains tag, communicator, length, source information, plus impl. private data
- Short
  - Message data (message for short) sent with envelope
- Eager
  - Message sent assuming destination can store
- Rendezvous
  - Message not sent until destination oks

Message Protocol Details

- Eager is not Rsend, rendezvous is not Ssend resp., but related
- User versus system buffer space
- Packetization
- Collective operations
- Datatypes, particularly non-contiguous
  - Handling of important special cases
    - Constant stride
    - Contiguous structures

Eager Protocol

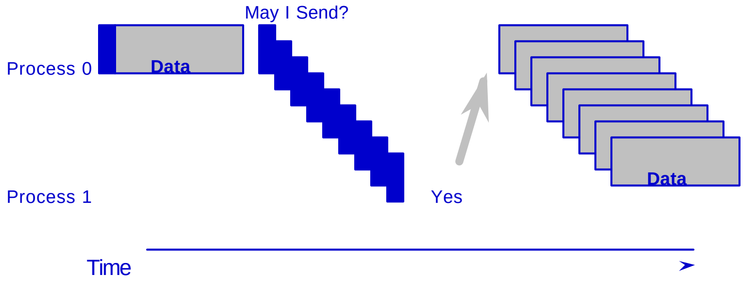


- Data delivered to process 1
  - No matching receive may exist; process 1 must then buffer and copy.

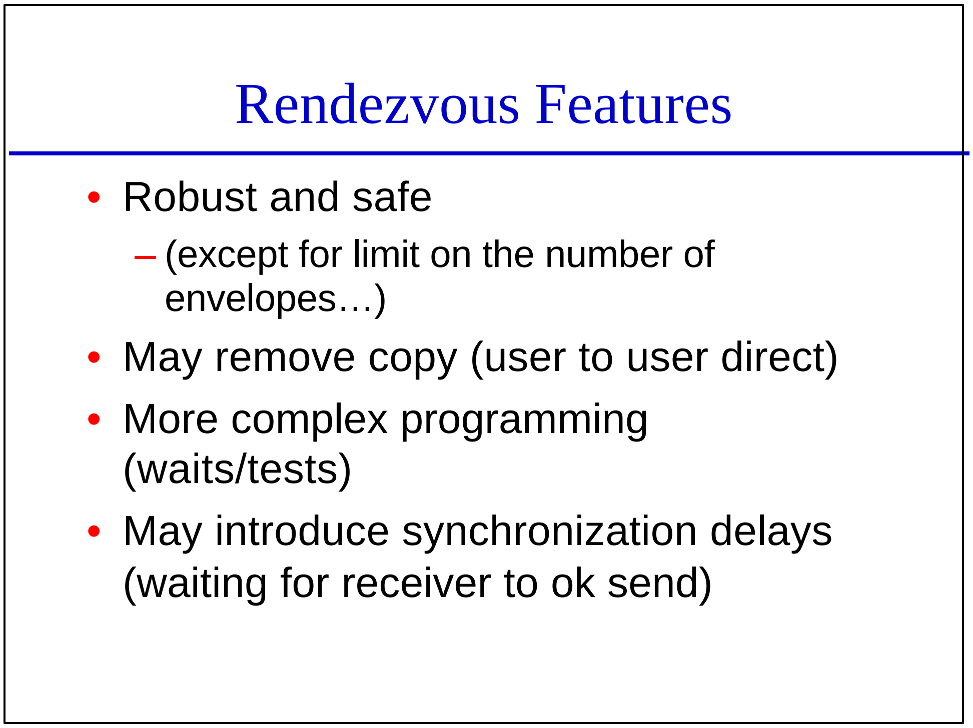
Eager Features

- Reduces synchronization delays
- Simplifies programming (just MPI_Send)
- Requires significant buffering
- May require active involvement of CPU to drain network at receiver's end
- May introduce additional copy (buffer to final destination)

Rendezvous Protocol

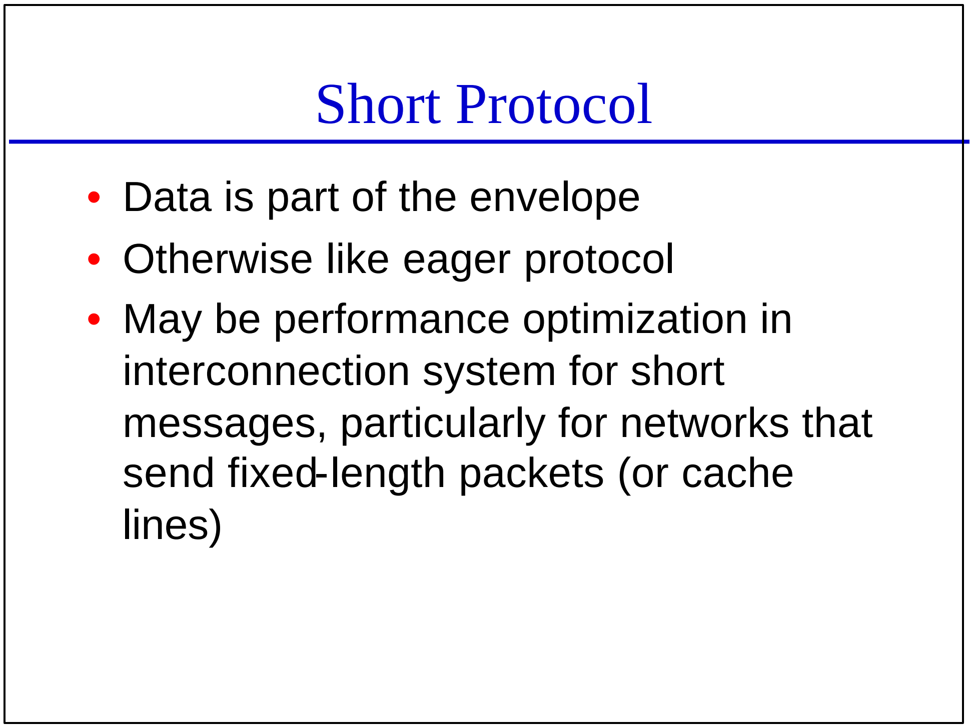


- Envelope delivered first
- Data delivered when user-buffer available
  - Only buffering of envelopes required



Rendezvous Features

- Robust and safe
  - (except for limit on the number of envelopes…)
- May remove copy (user to user direct)
- More complex programming (waits/tests)
- May introduce synchronization delays (waiting for receiver to ok send)



Short Protocol

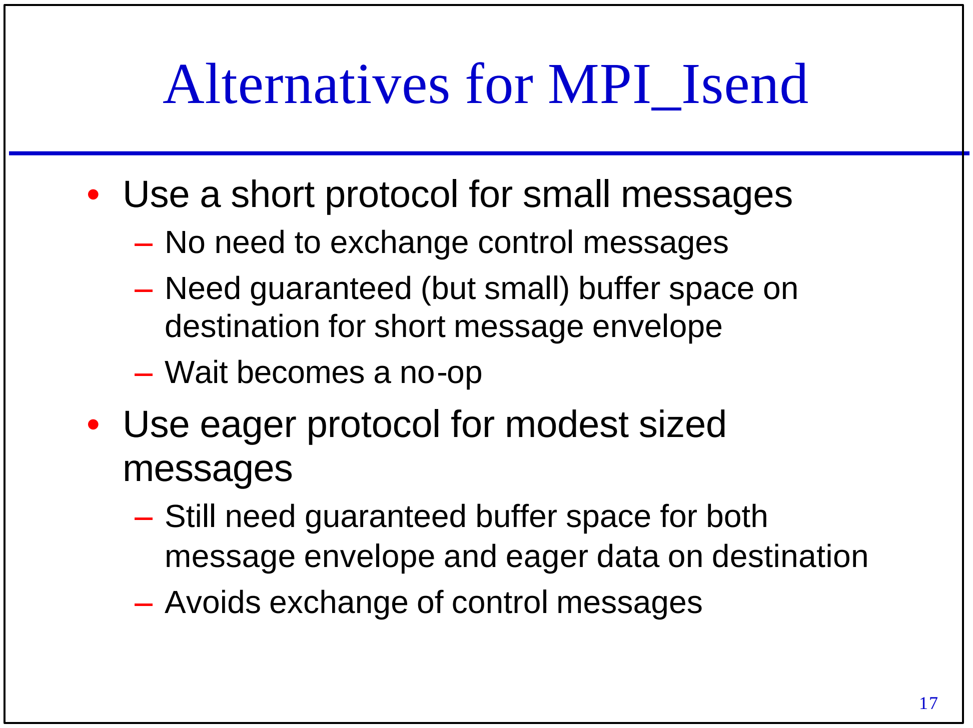
- Data is part of the envelope
- Otherwise like eager protocol
- May be performance optimization in interconnection system for short messages, particularly for networks that send fixed-length packets (or cache lines)

User and System Buffering

- Where is data stored (or staged) while being sent?
  - User's memory
    - Allocated on the fly
    - Preallocated
  - System memory
    - May be limited
    - Special memory may be faster

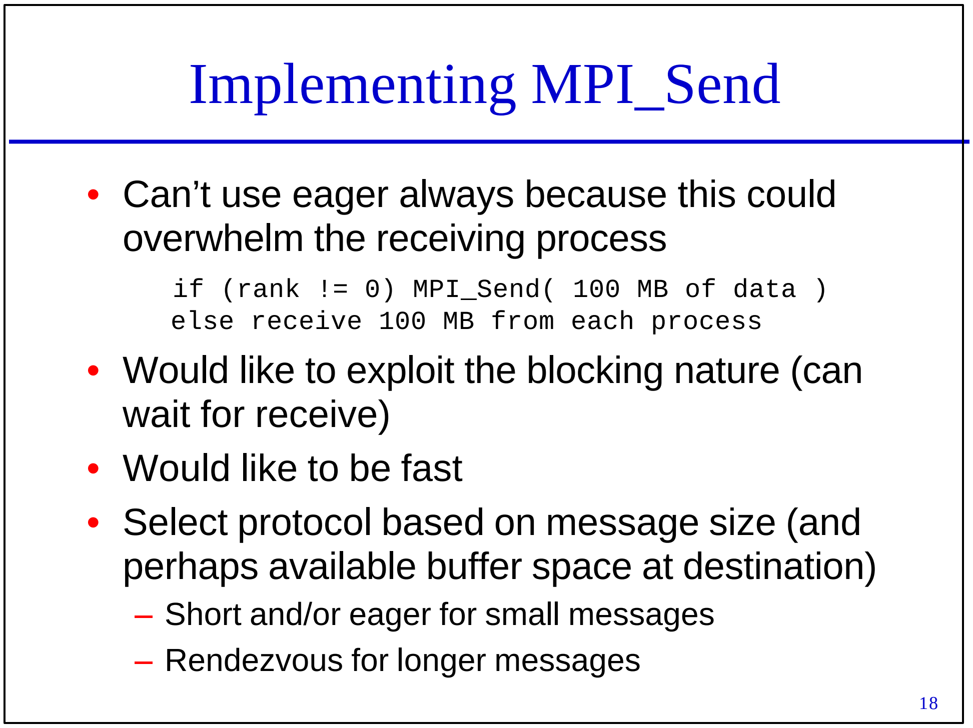
Implementing MPI_Isend

- Simplest implementation is to always use rendezvous protocol:
  - MPI_Isend delivers a request-to-send control message to receiver
  - Receiving process responds with an ok-to-send
    - May or may not have matching MPI receive; only needs buffer space to store incoming message
  - Sending process transfers data
- Wait for MPI_Isend request
  - wait for ok-to-send message from receiver
  - wait for data transfer to be complete on sending side



Alternatives for MPI_Isend

- Use a short protocol for small messages
  - No need to exchange control messages
  - Need guaranteed (but small) buffer space on destination for short message envelope
  - Wait becomes a no-op
- Use eager protocol for modest sized messages
  - Still need guaranteed buffer space for both message envelope and eager data on destination
  - Avoids exchange of control messages

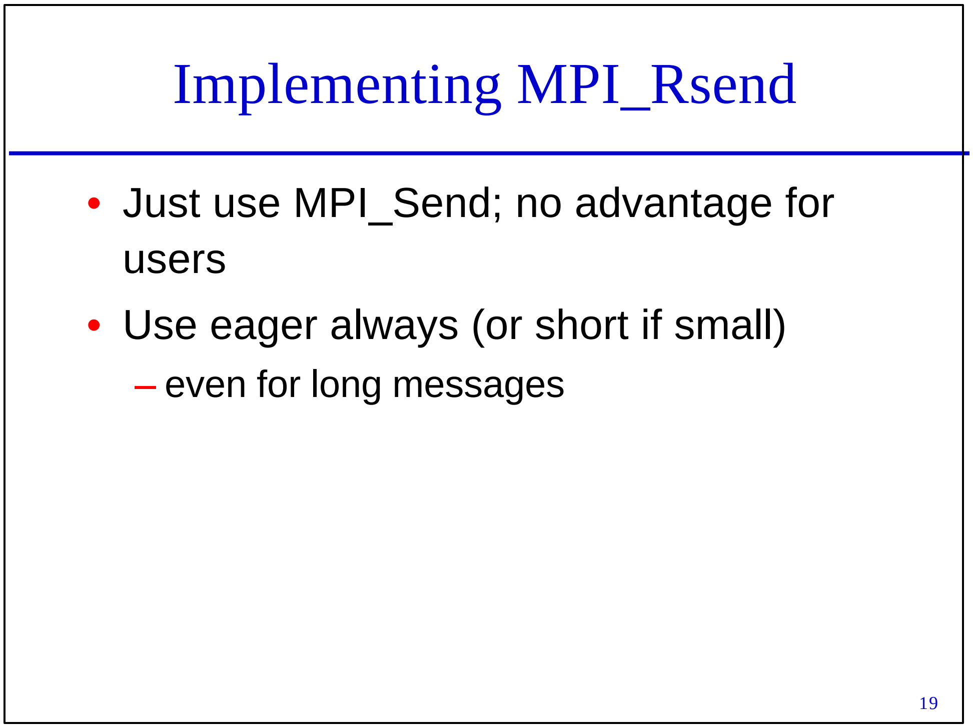


Implementing MPI_Send

- Can’t use eager always because this could overwhelm the receiving process

```
if (rank != 0) MPI_Send( 100 MB of data )
else receive 100 MB from each process
```

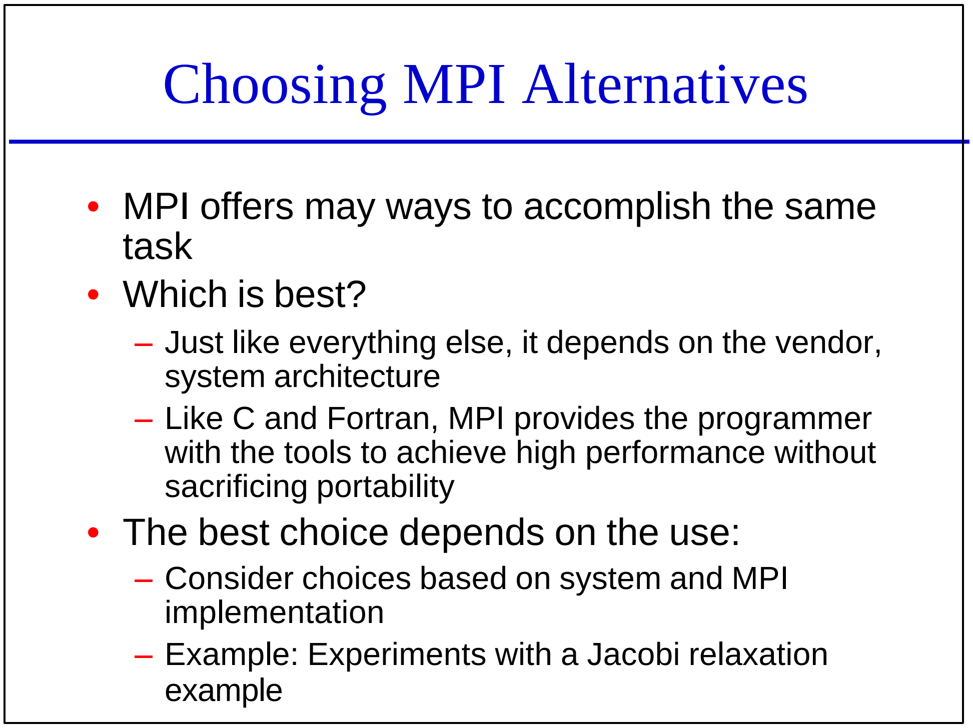
- Would like to exploit the blocking nature (can wait for receive)
- Would like to be fast
- Select protocol based on message size (and perhaps available buffer space at destination)
  - Short and/or eager for small messages
  - Rendezvous for longer messages



Implementing MPI_Rsend

- Just use MPI_Send; no advantage for users
- Use eager always (or short if small)
  - even for long messages

19



Choosing MPI Alternatives

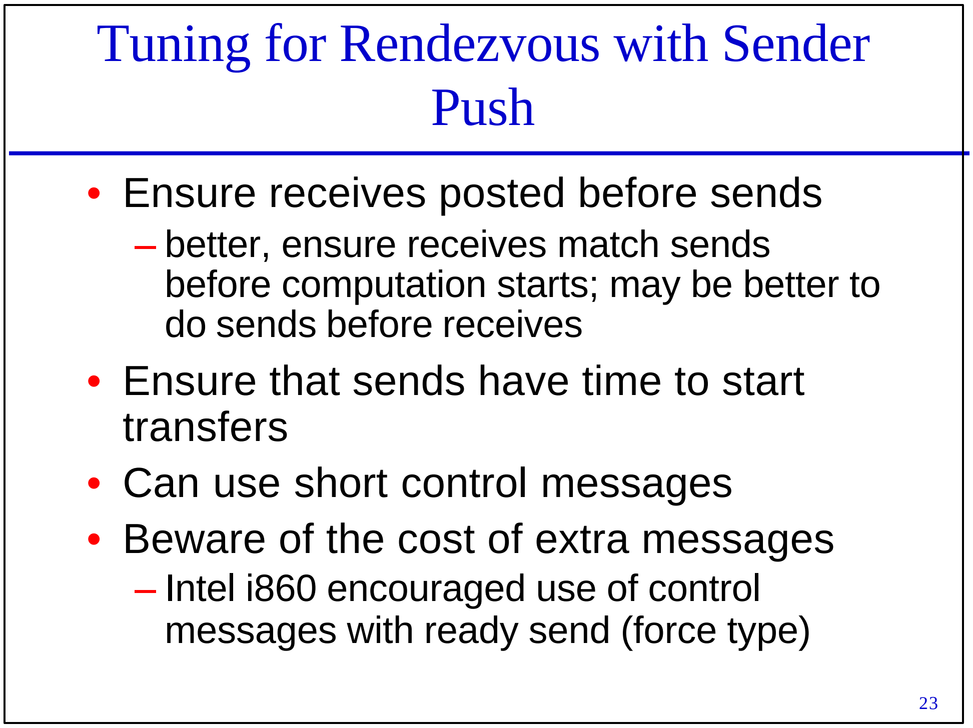
- MPI offers may ways to accomplish the same task
- Which is best?
  - Just like everything else, it depends on the vendor, system architecture
  - Like C and Fortran, MPI provides the programmer with the tools to achieve high performance without sacrificing portability
- The best choice depends on the use:
  - Consider choices based on system and MPI implementation
  - Example: Experiments with a Jacobi relaxation example

Tuning for MPI's Send/Receive Protocols

- Aggressive Eager
  - Performance problem: extra copies
  - Possible deadlock for inadequate eager buffering
  - Ensure that receives are posted before sends
  - MPI_Issend can be used to express "wait until receive is posted"
- Rendezvous with sender push
  - Extra latency
  - Possible delays while waiting for sender to begin
- Rendezvous with receiver pull
  - Possible delays while waiting for receiver to begin

Rendezvous Blocking

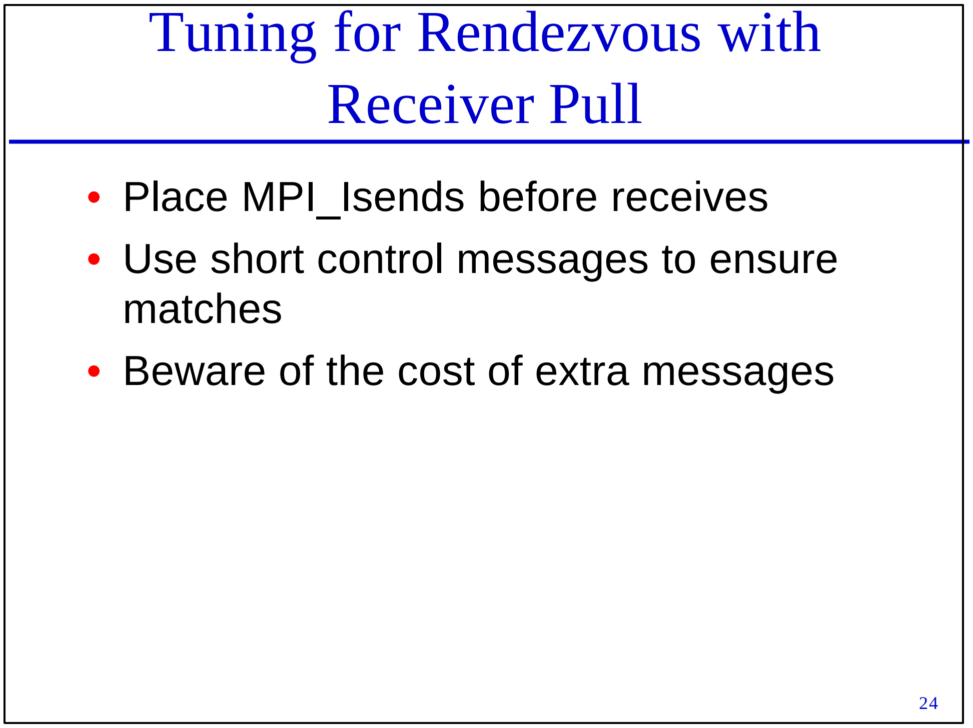
- What happens once sender and receiver rendezvous?
  - Sender (push) or receiver (pull) may complete operation
  - May block other operations while completing
- Performance tradeoff
  - If operation does *not* block (by checking for other requests), it adds latency or reduces bandwidth.
- Can reduce performance if a receiver, having acknowledged a send, must wait for the sender to complete a separate operation that it has started.



Tuning for Rendezvous with Sender Push

- Ensure receives posted before sends
  - better, ensure receives match sends before computation starts; may be better to do sends before receives
- Ensure that sends have time to start transfers
- Can use short control messages
- Beware of the cost of extra messages
  - Intel i860 encouraged use of control messages with ready send (force type)

23



Tuning for Rendezvous with Receiver Pull

- Place MPI_Isends before receives
- Use short control messages to ensure matches
- Beware of the cost of extra messages

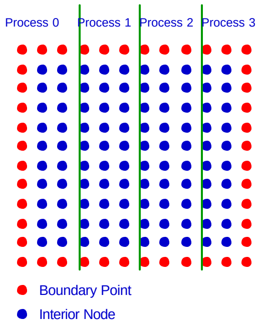
24

Experiments with MPI Implementations

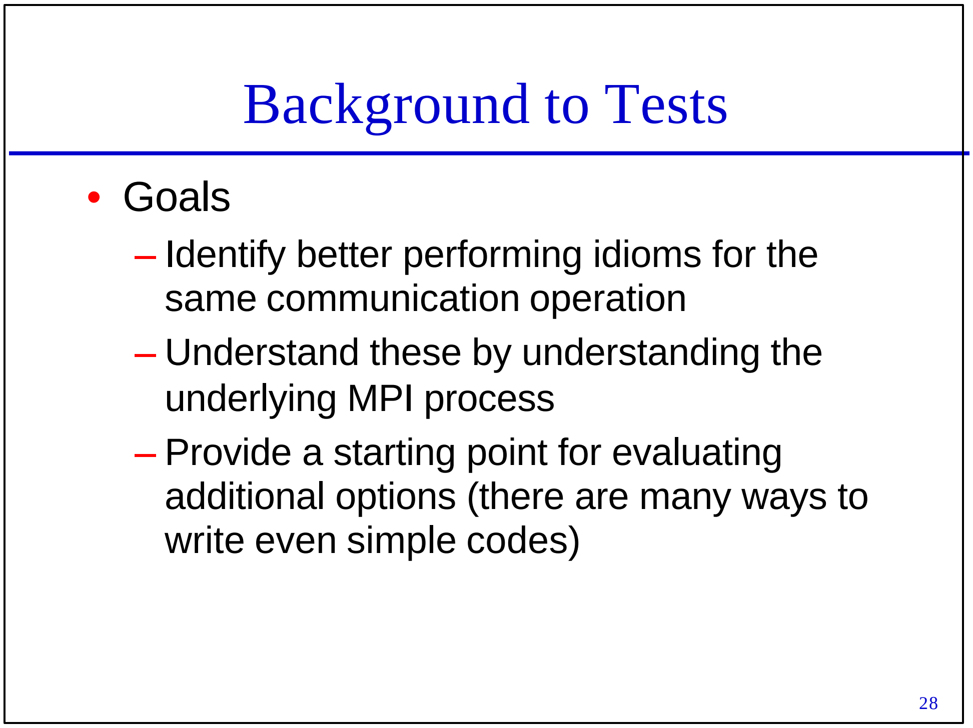
- Multiparty data exchange
- Jacobi iteration in 2 dimensions
  - Model for PDEs, Sparse matrix-vector products, and algorithms with surface/volume behavior
  - Issues are similar to unstructured grid problems (but harder to illustrate)
- Others at http://www.mcs.anl.gov/mpi/tutorials/perf

Jacobi Iteration (Fortran Ordering)

- Simple parallel data structure

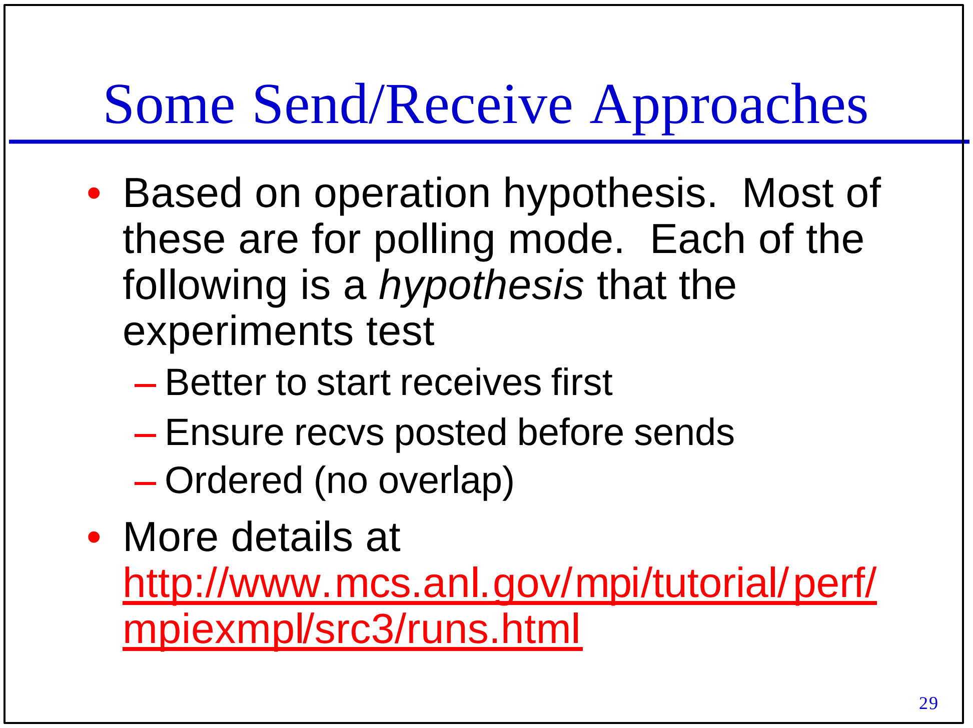


- Processes exchange columns with neighbors
- Local part declared as xlocal(m,0:n+1)



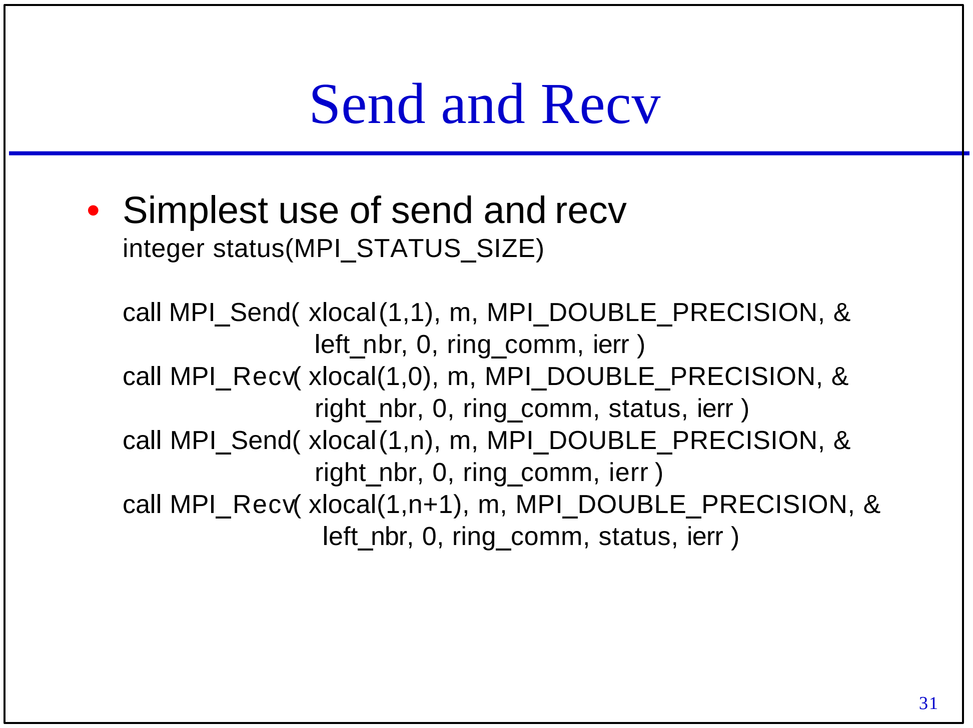
Background to Tests

- Goals
  - Identify better performing idioms for the same communication operation
  - Understand these by understanding the underlying MPI process
  - Provide a starting point for evaluating additional options (there are many ways to write even simple codes)



Some Send/Receive Approaches

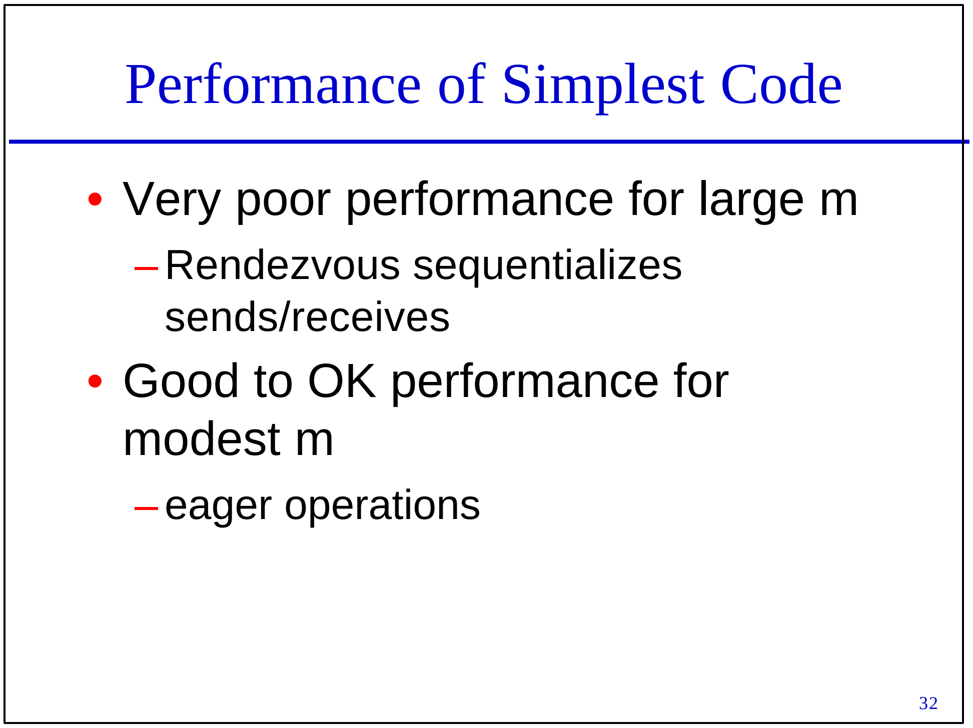
- Based on operation hypothesis. Most of these are for polling mode. Each of the following is a *hypothesis* that the experiments test
  - Better to start receives first
  - Ensure recvs posted before sends
  - Ordered (no overlap)
- More details at http://www.mcs.anl.gov/mpi/tutorial/perf/mpiexmpl/src3/runs.html



Send and Recv

- Simplest use of send and recv

```
integer status(MPI_STATUS_SIZE)

call MPI_Send( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
                  left_nbr, 0, ring_comm, ierr )
call MPI_Recv( xlocal(1,0), m, MPI_DOUBLE_PRECISION, &
                  right_nbr, 0, ring_comm, status, ierr )
call MPI_Send( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
                  right_nbr, 0, ring_comm, ierr )
call MPI_Recv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION, &
                  left_nbr, 0, ring_comm, status, ierr )
```



Performance of Simplest Code

- Very poor performance for large m
  - Rendezvous sequentializes sends/receives
- Good to OK performance for modest m
  - eager operations

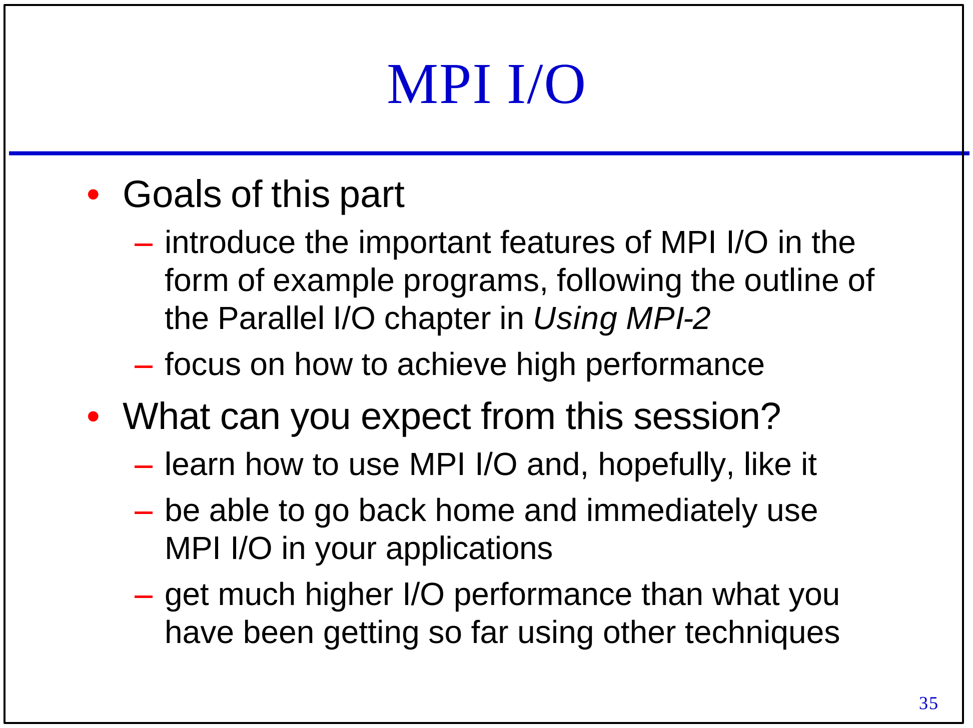
Better to start sends first

- Irecv, Isend, Waitall - ok performance

```
integer statuses(MPI_STATUS_SIZE,4), requests(4)

call MPI_Isend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
                right_nbr, ring_comm, requests(1), ierr )
call MPI_Isend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
                left_nbr, ring_comm, requests(3), ierr )
call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION,&
                left_nbr, ring_comm, requests(2), ierr )
call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION,&
                right_nbr, ring_comm, requests(4), ierr )
call MPI_Waitall( 4, requests, statuses, ierr )
```

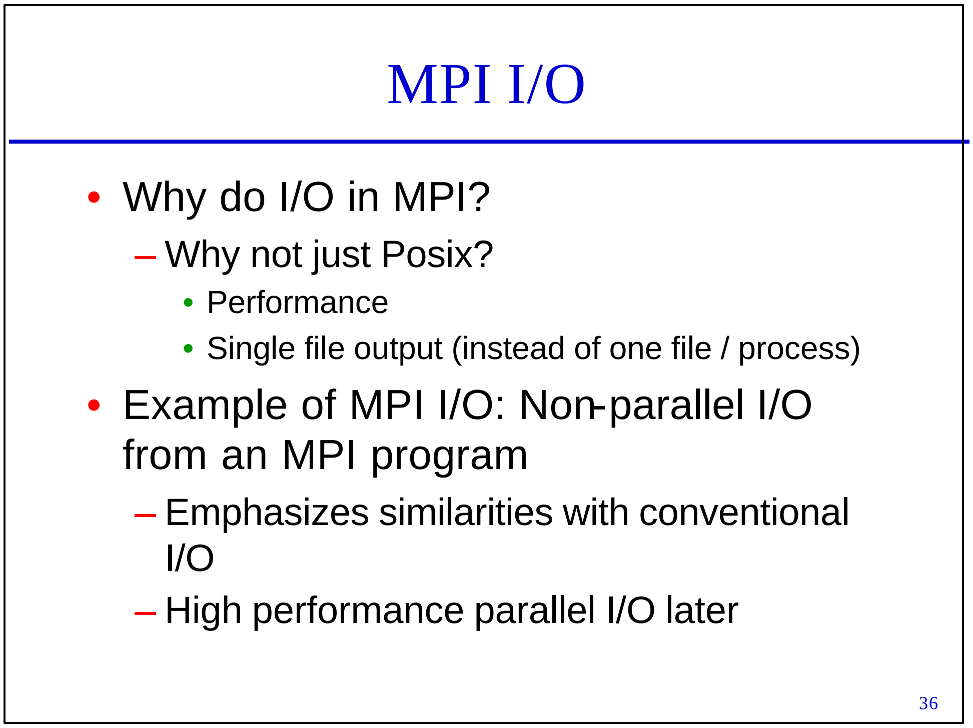
Summary of Results

- Better to start sends before receives
  - Most implementations use rendezvous protocols for long messages (Cray, IBM, SGI, MPICH, LAM)
  - Sending the envelope immediately is the best way to improve performance
- Short messages even better
  - Eager messages have lowest latency



MPI I/O

- Goals of this part
  - introduce the important features of MPI I/O in the form of example programs, following the outline of the Parallel I/O chapter in *Using MPI-2*
  - focus on how to achieve high performance
- What can you expect from this session?
  - learn how to use MPI I/O and, hopefully, like it
  - be able to go back home and immediately use MPI I/O in your applications
  - get much higher I/O performance than what you have been getting so far using other techniques



MPI I/O

- Why do I/O in MPI?
  - Why not just Posix?
    - Performance
    - Single file output (instead of one file / process)
- Example of MPI I/O: Non-parallel I/O from an MPI program
  - Emphasizes similarities with conventional I/O
  - High performance parallel I/O later

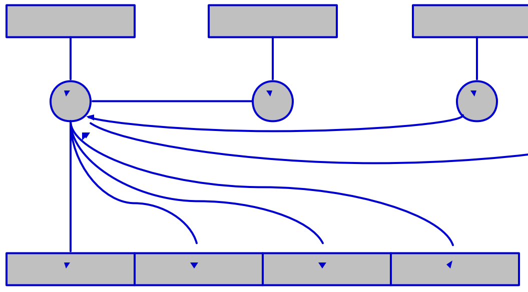
Why MPI is a Good Setting for Parallel I/O

- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations
- I.e., lots of MPI-like machinery

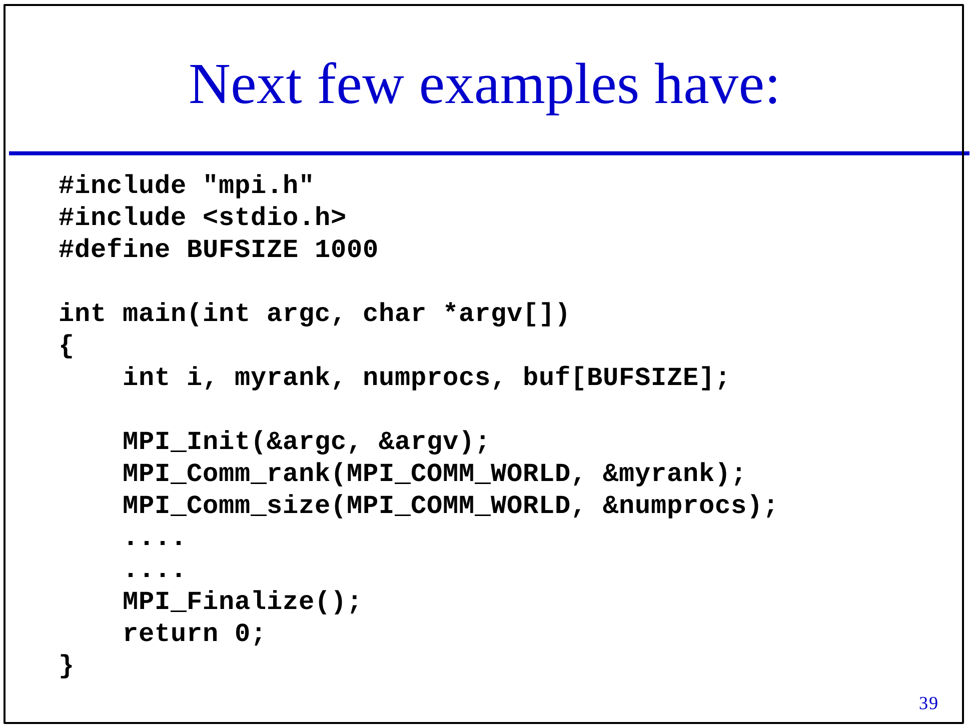
37

Introduction to Parallel I/O

- First example: non-parallel I/O from an MPI program

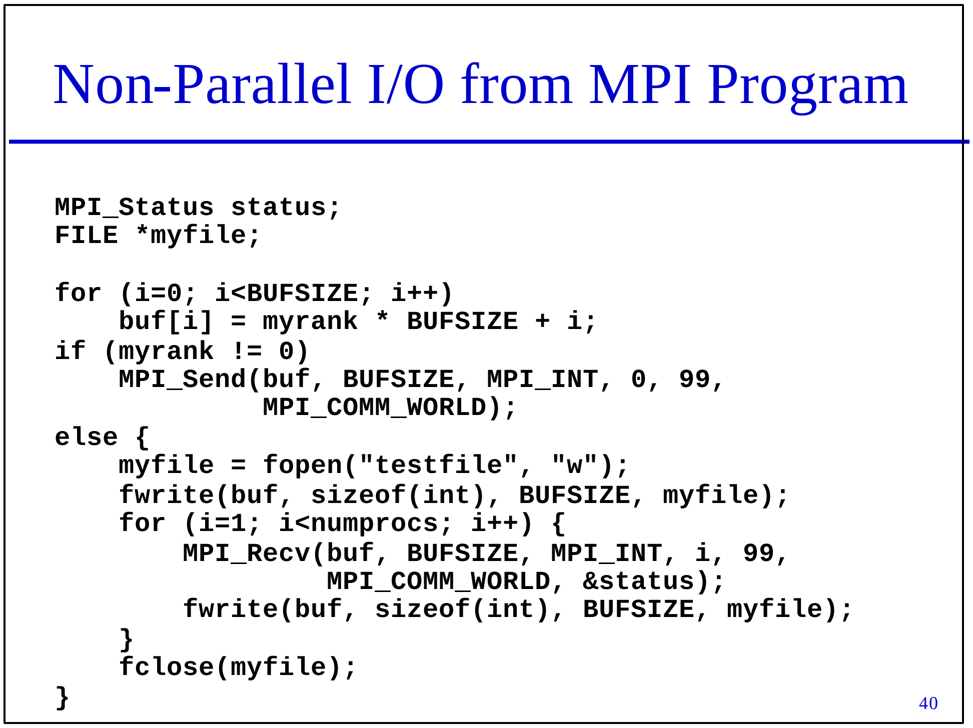


38



Next few examples have:

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 1000

int main(int argc, char *argv[])
{
    int i, myrank, numprocs, buf[BUFSIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    ....
    ....
    MPI_Finalize();
    return 0;
}
```



Non-Parallel I/O from MPI Program

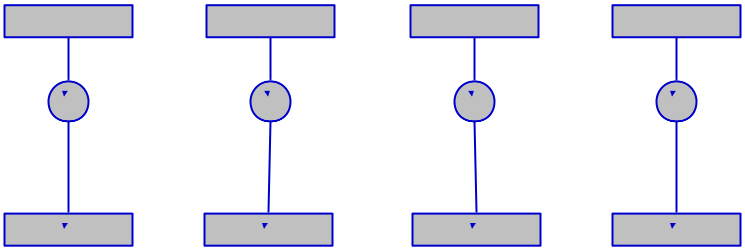
```
MPI_Status status;
FILE *myfile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
if (myrank != 0)
    MPI_Send(buf, BUFSIZE, MPI_INT, 0, 99,
             MPI_COMM_WORLD);
else {
    myfile = fopen("testfile", "w");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    for (i=1; i<numprocs; i++) {
        MPI_Recv(buf, BUFSIZE, MPI_INT, i, 99,
                 MPI_COMM_WORLD, &status);
        fwrite(buf, sizeof(int), BUFSIZE, myfile);
    }
    fclose(myfile);
}
```

Pros and Cons of Sequential I/O

- Pros:
  - parallel machine may support I/O from only one process
  - I/O libraries (e.g. HDF-4, SILO, etc.) not parallel
  - resulting single file is handy for **`ftp`**, **`mv`**
  - big blocks improve performance
  - short distance from original, serial code
- Cons:
  - lack of parallelism limits scalability, performance

Non-MPI Parallel I/O

- Each process writes to a separate file



- Pro: parallelism
- Con: lots of small files to manage

Non-MPI Parallel I/O

```
char filename[128];
FILE *myfile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
sprintf(filename, "testfile.%d", myrank);
myfile = fopen(filename, "w");
fwrite(buf, sizeof(int), BUFSIZE, myfile);
fclose(myfile);
```

MPI I/O to Separate Files

- Same pattern as previous example
- MPI I/O replaces Unix I/O in a straightforward way
- Easy way to start with MPI I/O
- Does not exploit advantages of MPI I/O
    - in producing single file
    - in allowing collective operations
- Note files cannot be read conveniently by a different number of processes

MPI I/O to Separate Files

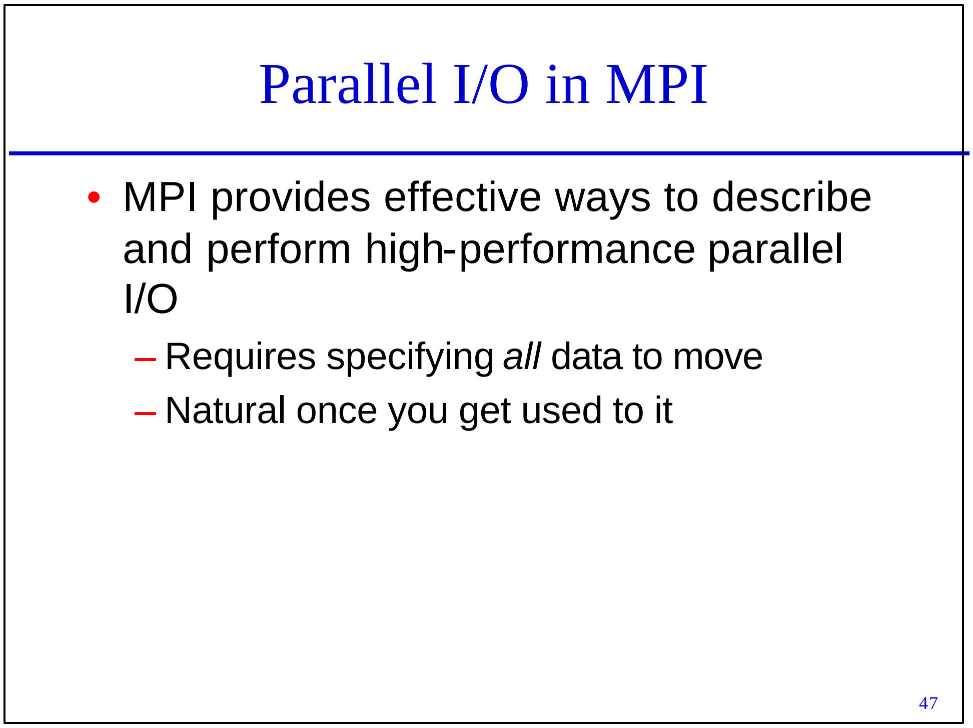
```
char filename[128];
MPI_File myfile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
sprintf(filename, "testfile.%d", myrank);
MPI_File_open(MPI_COMM_SELF, filename,
              MPI_MODE_WRONLY | MPI_MODE_CREATE,
              MPI_INFO_NULL, &myfile);
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&myfile);
```

MPI Versions of Unix I/O

• Unix	• MPI
`FILE myfile;`	`MPI_File myfile;`
`myfile = fopen(...)`	`MPI_File_open(...)` takes info, comm args

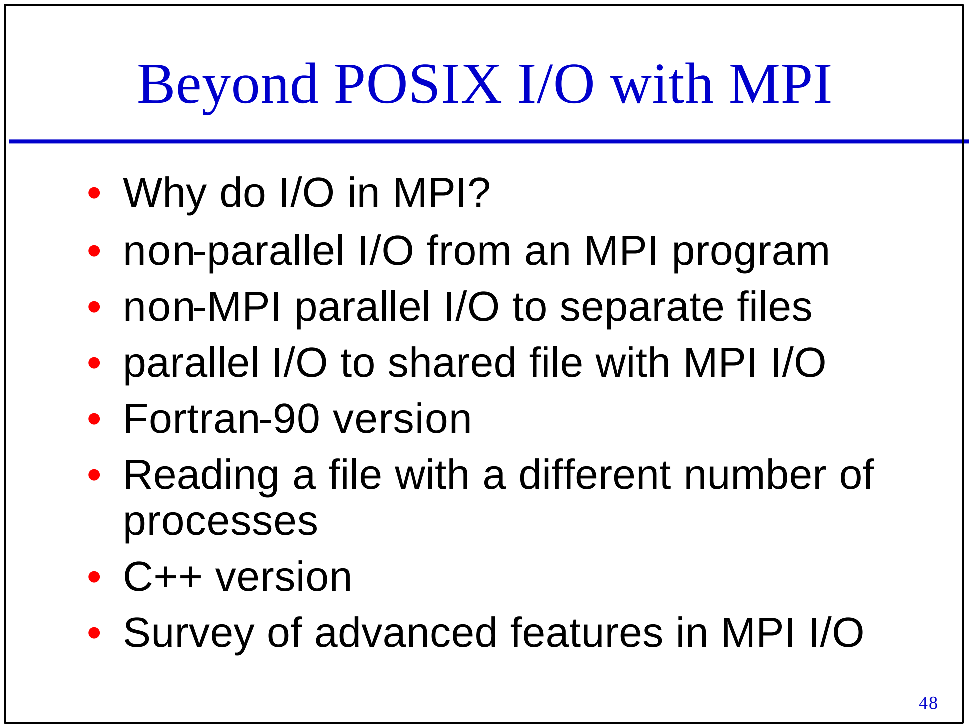
`fread(...)`
`fwrite(...)`
	`MPI_File_read/write(...)` take (addr, count, datatype)

`fclose`	`MPI_File_close`



Parallel I/O in MPI

- MPI provides effective ways to describe and perform high-performance parallel I/O
  - Requires specifying *all* data to move
  - Natural once you get used to it



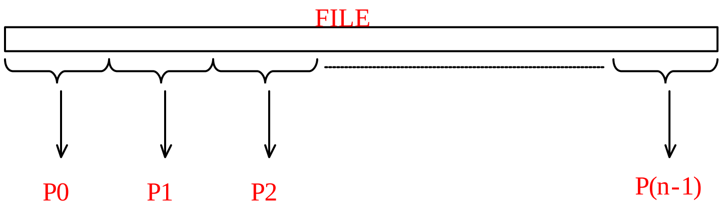
Beyond POSIX I/O with MPI

- Why do I/O in MPI?
- non-parallel I/O from an MPI program
- non-MPI parallel I/O to separate files
- parallel I/O to shared file with MPI I/O
- Fortran-90 version
- Reading a file with a different number of processes
- C++ version
- Survey of advanced features in MPI I/O

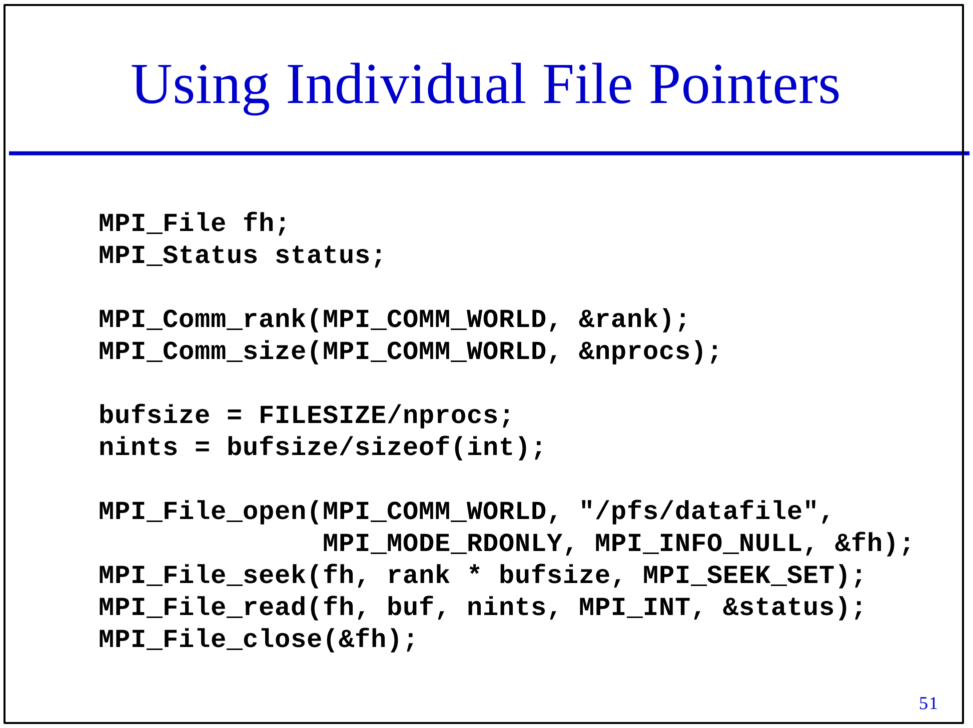
Parallel I/O with MPI

- Nonparallel I/O shown earlier
  - Simple but
    - Poor performance (single process writes to one file) or
    - Awkward and not interoperable with other tools (each process writes a separate file)
- Parallel I/O
  - Provides high performance
  - Can provide a single file that can be used with other tools (such as visualization programs)

Using MPI for Simple I/O

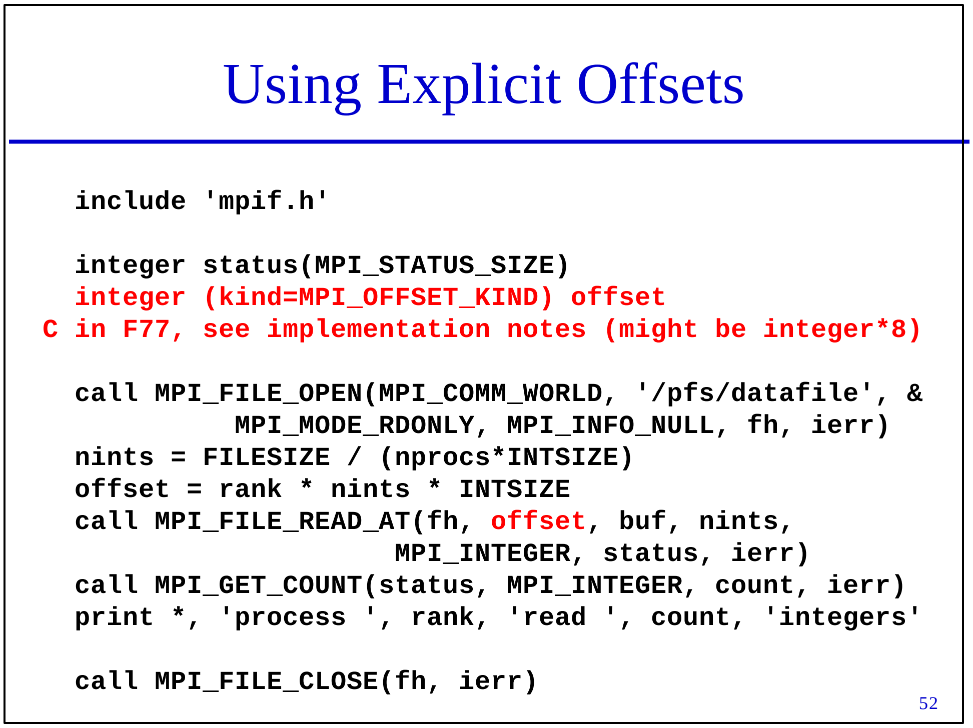


Each process needs to read a chunk of data from a common file



Using Individual File Pointers

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```



Using Explicit Offsets

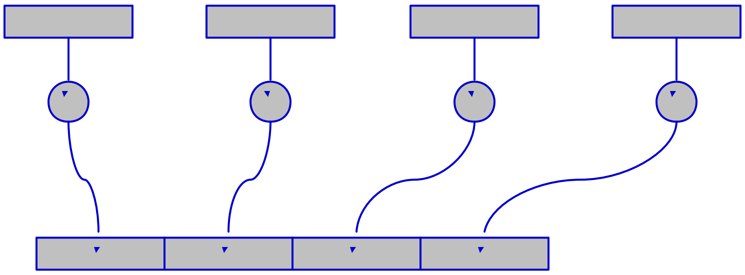
```
  include 'mpif.h'

  integer status(MPI_STATUS_SIZE)
  integer (kind=MPI_OFFSET_KIND) offset
C in F77, see implementation notes (might be integer*8)

  call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
            MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
  nints = FILESIZE / (nprocs*INTSIZE)
  offset = rank * nints * INTSIZE
  call MPI_FILE_READ_AT(fh, offset, buf, nints,
                        MPI_INTEGER, status, ierr)
  call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
  print *, 'process ', rank, 'read ', count, 'integers'

  call MPI_FILE_CLOSE(fh, ierr)
```

Writing to a File

- Use **`MPI_File_write`** or **`MPI_File_write_at`**
- Use **`MPI_MODE_WRONLY`** or **`MPI_MODE_RDWR`** as the flags to **`MPI_File_open`**
- If the file doesn't exist previously, the flag **`MPI_MODE_CREATE`** must also be passed to **`MPI_File_open`**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

MPI Parallel I/O to Single File

- Processes write to shared file



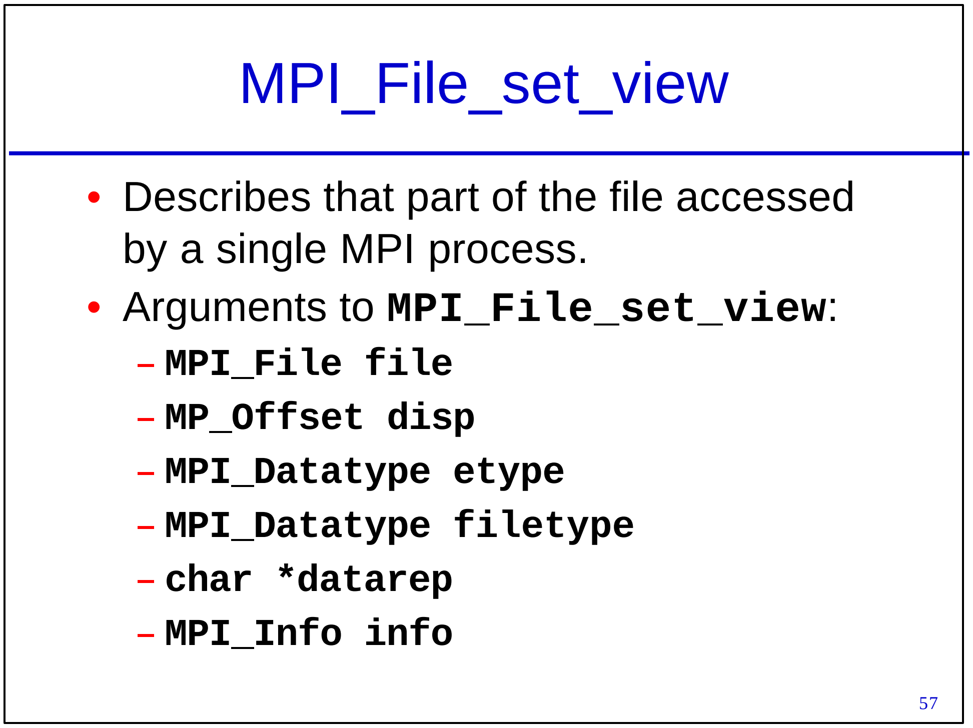
- **`MPI_File_set_view`** assigns regions of the file to separate processes

File Views

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

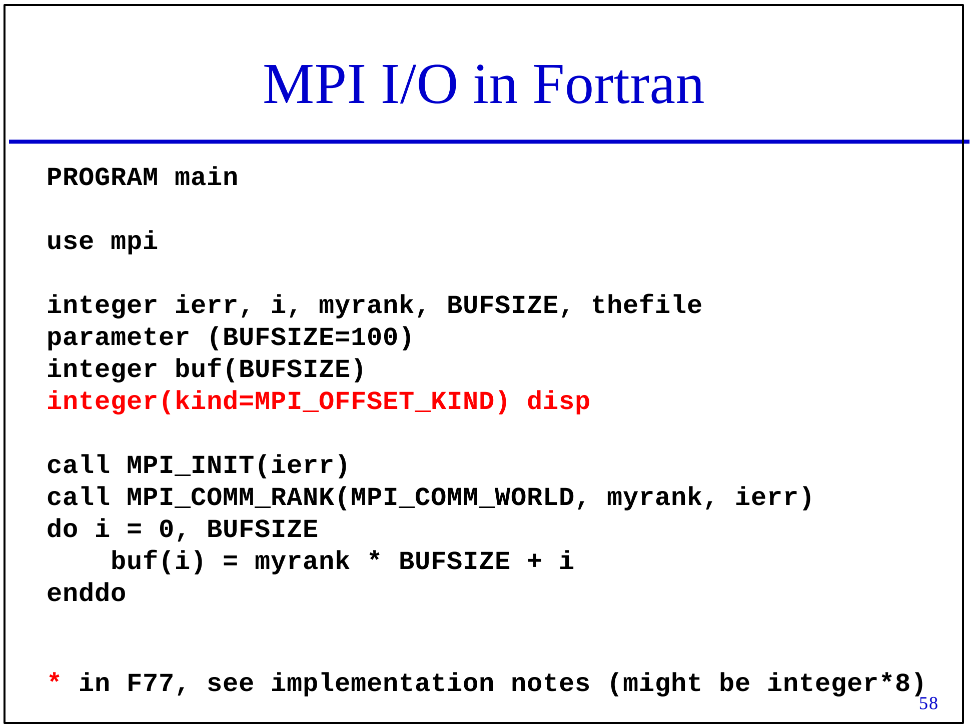
MPI Parallel I/O to Single File

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                  MPI_INT, MPI_INT, "native",
                  MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```



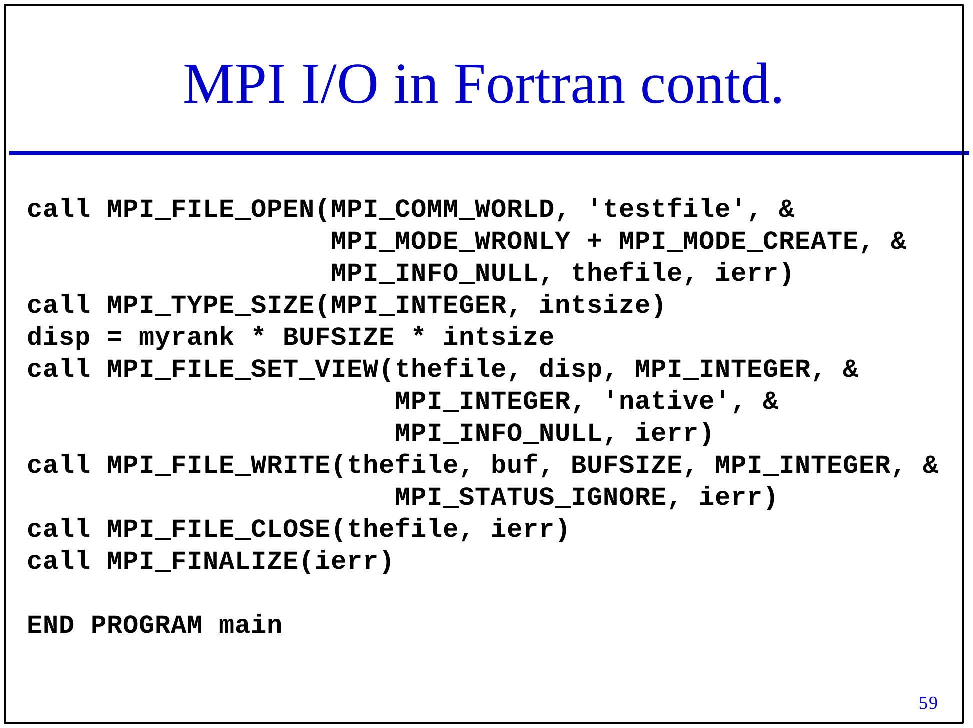
MPI_File_set_view

- Describes that part of the file accessed by a single MPI process.
- Arguments to **`MPI_File_set_view`**:
  - **`MPI_File file`**
  - **`MP_Offset disp`**
  - **`MPI_Datatype etype`**
  - **`MPI_Datatype filetype`**
  - **`char *datarep`**
  - **`MPI_Info info`**



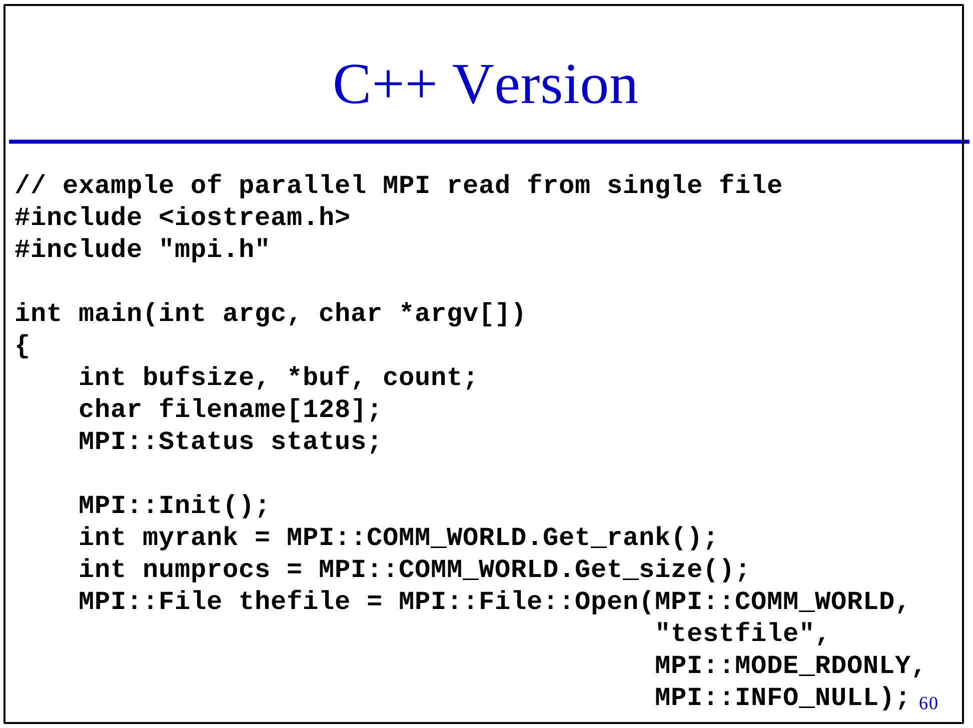
MPI I/O in Fortran

```
PROGRAM main

use mpi

integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
integer(kind=MPI_OFFSET_KIND) disp

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
do i = 0, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
enddo


* in F77, see implementation notes (might be integer*8)
```



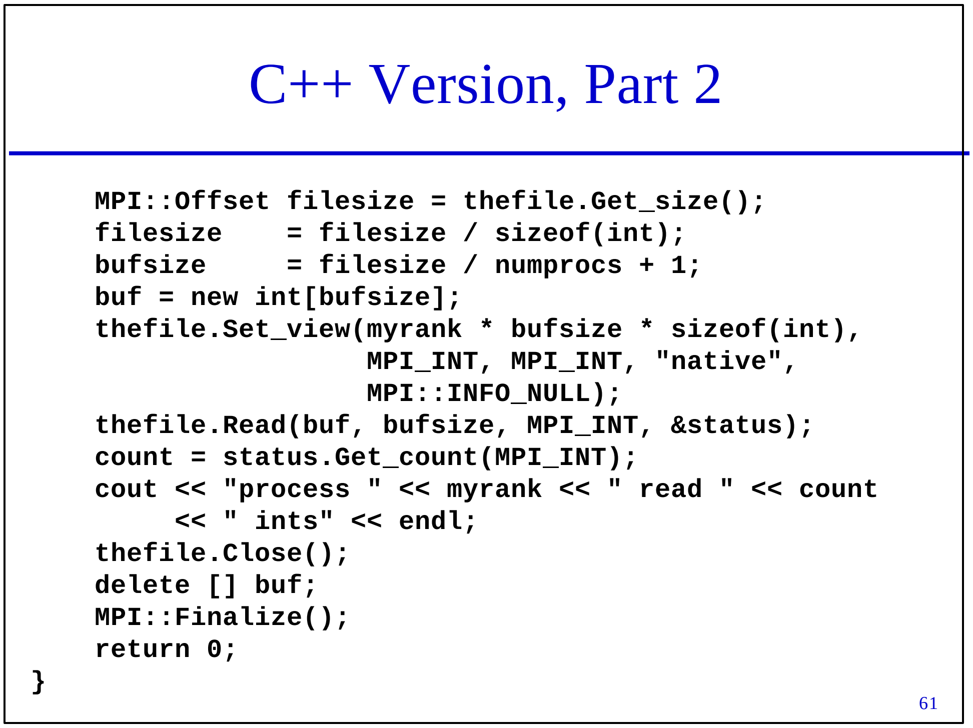
MPI I/O in Fortran contd.

```
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
                   MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                   MPI_INFO_NULL, thefile, ierr)
call MPI_TYPE_SIZE(MPI_INTEGER, intsize)
disp = myrank * BUFSIZE * intsize
call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
                       MPI_INTEGER, 'native', &
                       MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER, &
                       MPI_STATUS_IGNORE, ierr)
call MPI_FILE_CLOSE(thefile, ierr)
call MPI_FINALIZE(ierr)

END PROGRAM main
```



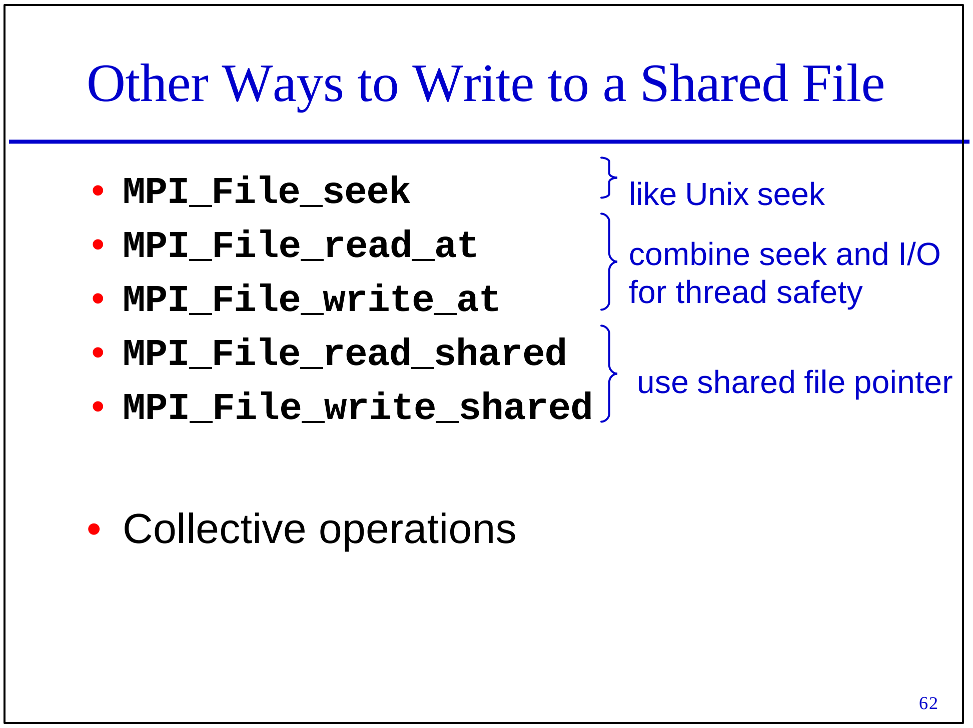
C++ Version

```
// example of parallel MPI read from single file
#include <iostream.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int bufsize, *buf, count;
    char filename[128];
    MPI::Status status;

    MPI::Init();
    int myrank = MPI::COMM_WORLD.Get_rank();
    int numprocs = MPI::COMM_WORLD.Get_size();
    MPI::File thefile = MPI::File::Open(MPI::COMM_WORLD,
                                        "testfile",
                                        MPI::MODE_RDONLY,
                                        MPI::INFO_NULL);
```



C++ Version, Part 2

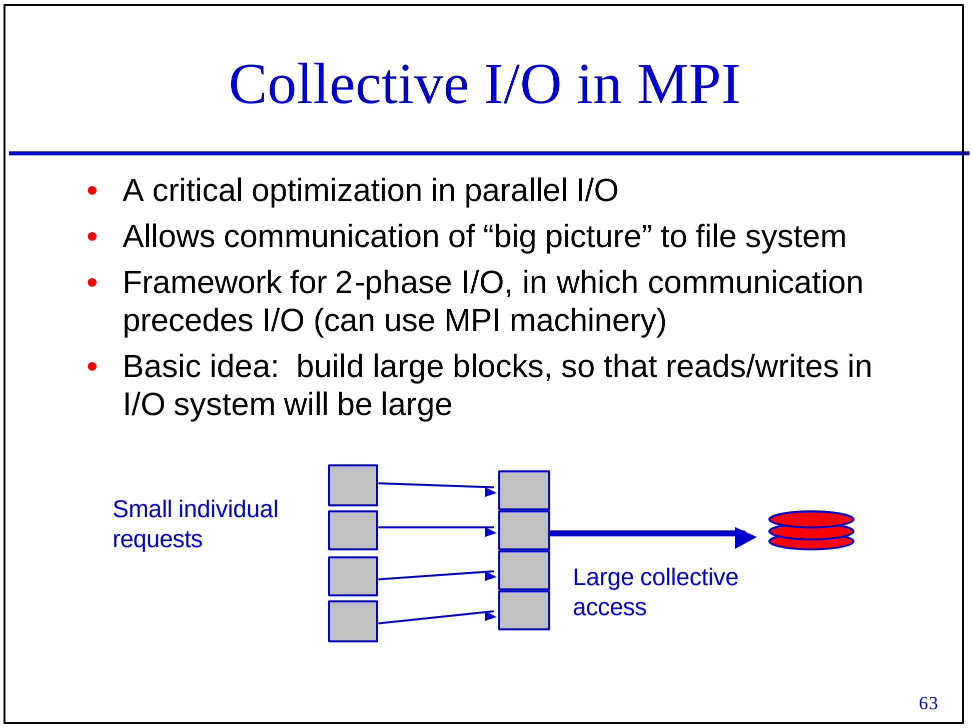
```
    MPI::Offset filesize = thefile.Get_size();
    filesize    = filesize / sizeof(int);
    bufsize     = filesize / numprocs + 1;
    buf = new int[bufsize];
    thefile.Set_view(myrank * bufsize * sizeof(int),
                     MPI_INT, MPI_INT, "native",
                     MPI::INFO_NULL);
    thefile.Read(buf, bufsize, MPI_INT, &status);
    count = status.Get_count(MPI_INT);
    cout << "process " << myrank << " read " << count
         << " ints" << endl;
    thefile.Close();
    delete [] buf;
    MPI::Finalize();
    return 0;
}
```



Other Ways to Write to a Shared File

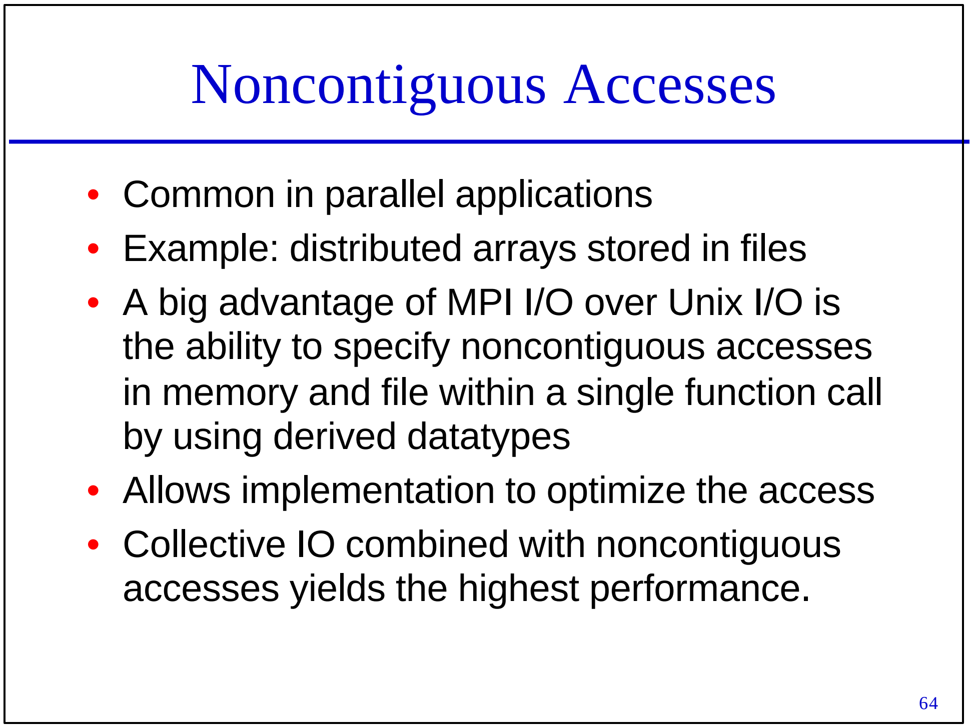
- **MPI_File_seek** } like Unix seek
- **MPI_File_read_at**
- **MPI_File_write_at** } combine seek and I/O for thread safety
- **MPI_File_read_shared**
- **MPI_File_write_shared** } use shared file pointer

- Collective operations



Collective I/O in MPI

- A critical optimization in parallel I/O
- Allows communication of "big picture" to file system
- Framework for 2-phase I/O, in which communication precedes I/O (can use MPI machinery)
- Basic idea: build large blocks, so that reads/writes in I/O system will be large

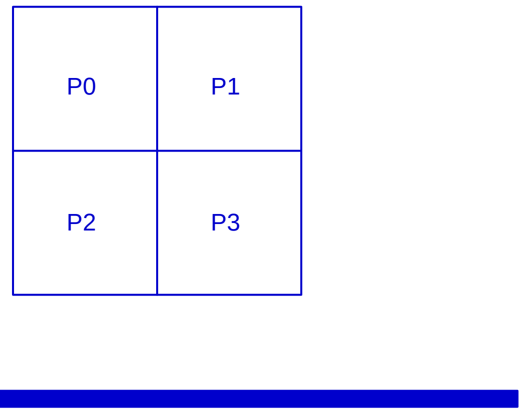


Noncontiguous Accesses

- Common in parallel applications
- Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory and file within a single function call by using derived datatypes
- Allows implementation to optimize the access
- Collective IO combined with noncontiguous accesses yields the highest performance.

Example:
Distributed Array Access

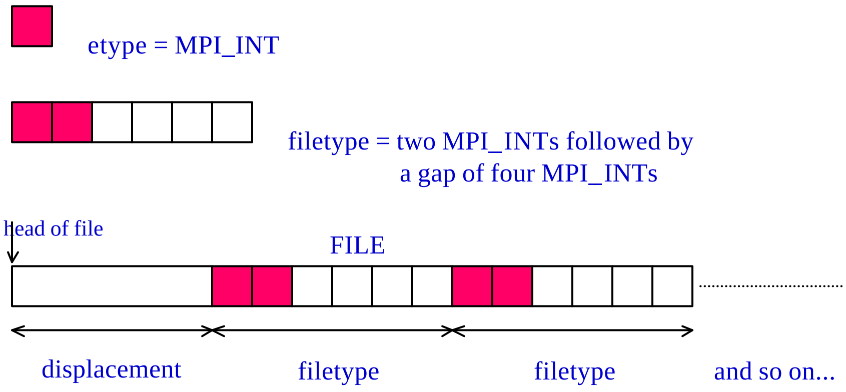
2D array distributed among four processes



File containing the global array in row-major order

65

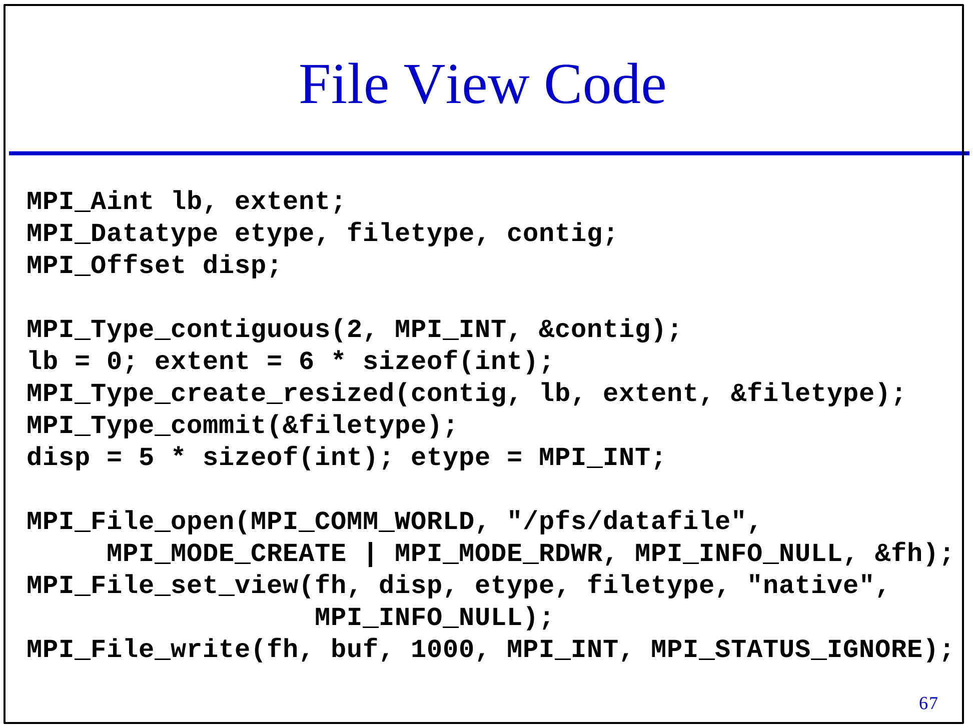
A Simple File View Example



etype = MPI_INT

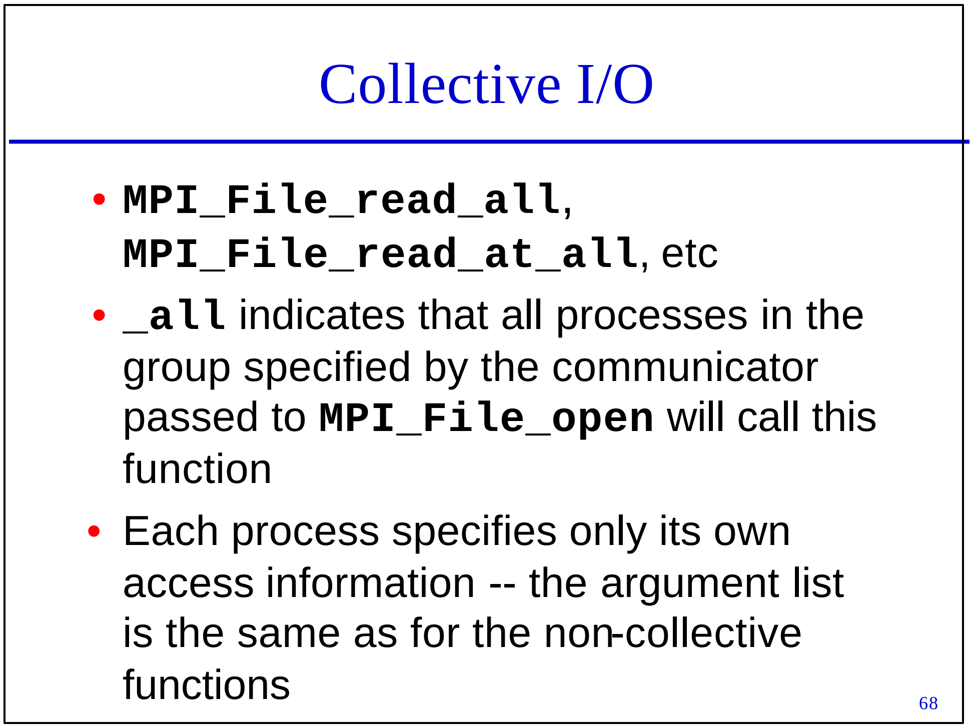
filetype = two MPI_INTs followed by
a gap of four MPI_INTs

66



File View Code

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```



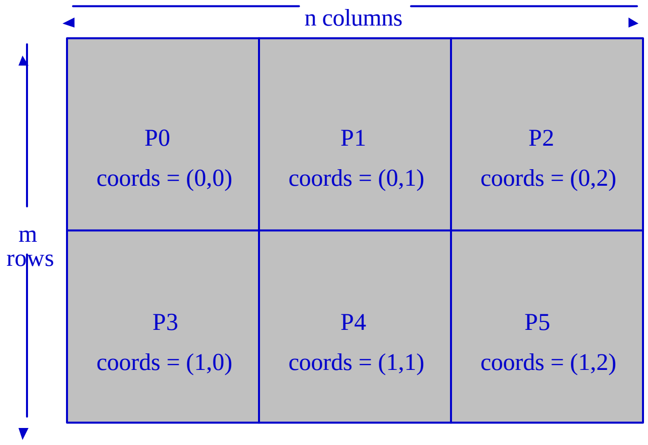
Collective I/O

- **`MPI_File_read_all`**, **`MPI_File_read_at_all`**, etc
- **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
- Each process specifies only its own access information -- the argument list is the same as for the non-collective functions

Collective I/O

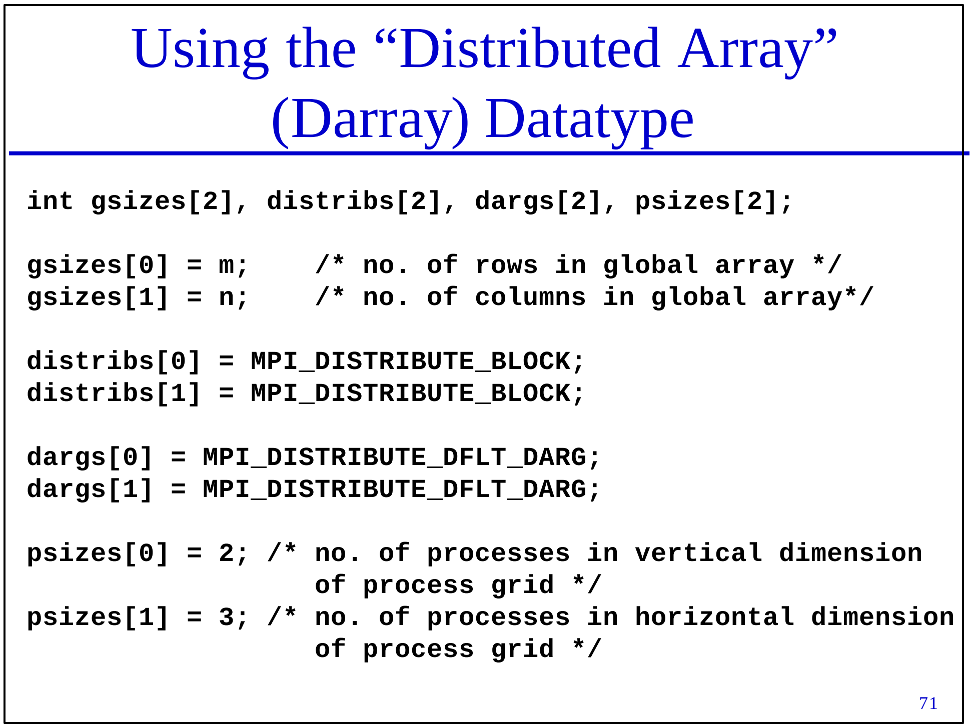
- By calling the collective I/O functions, the user allows an implementation to optimize the request based on the combined request of all processes
- The implementation can merge the requests of different processes and service the merged request efficiently
- Particularly effective when the accesses of different processes are noncontiguous and interleaved

Accessing Arrays Stored in Files



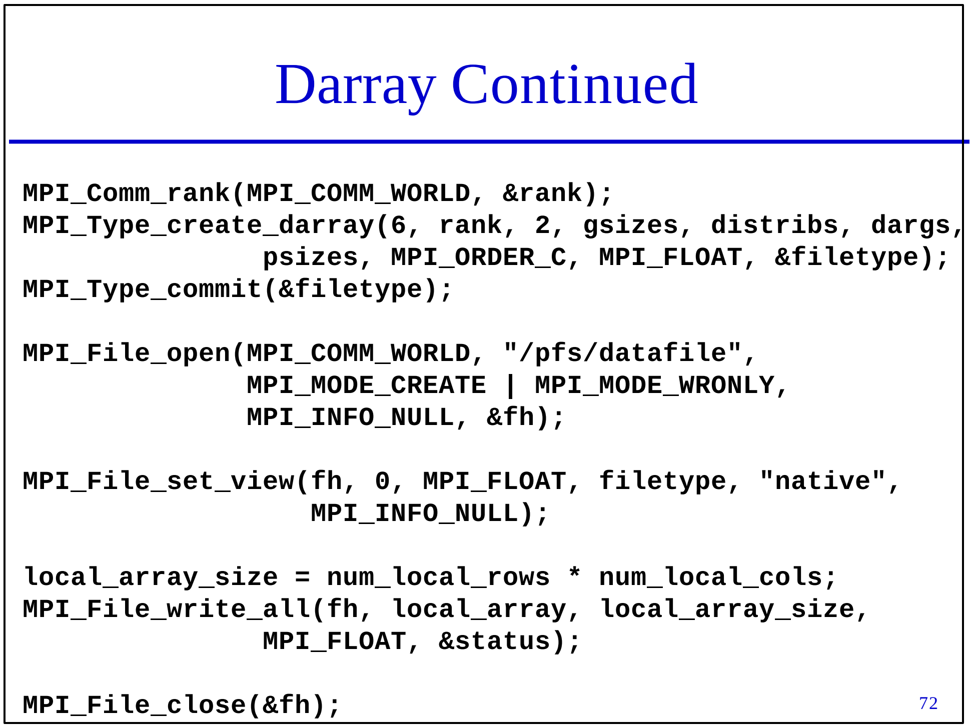
P0 coords = (0,0)	P1 coords = (0,1)	P2 coords = (0,2)
P3 coords = (1,0)	P4 coords = (1,1)	P5 coords = (1,2)

nproc(1) = 2, nproc(2) = 3



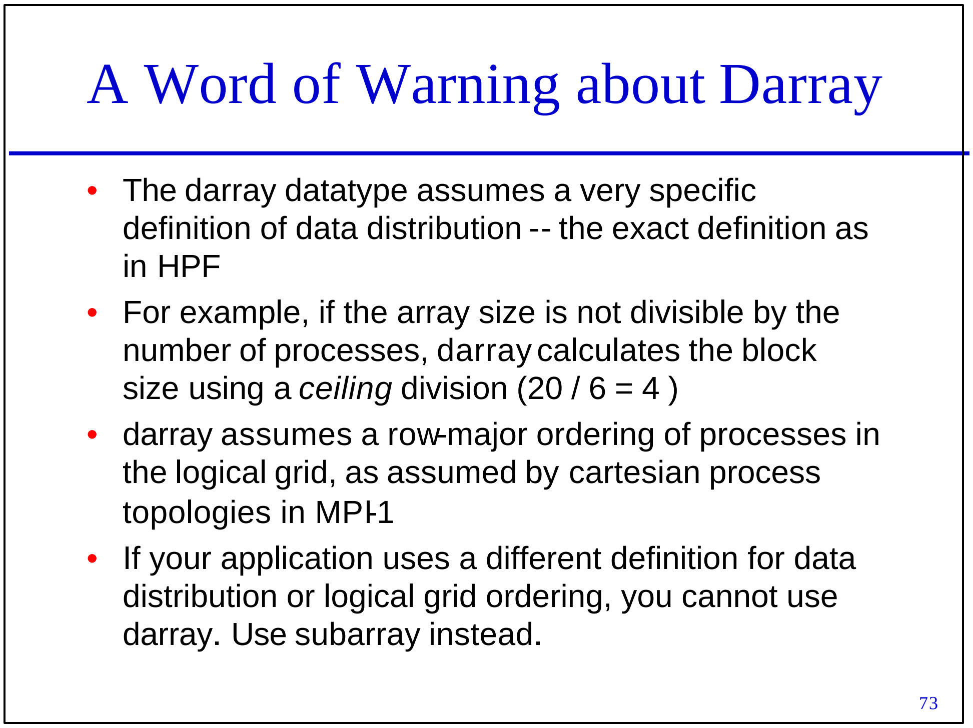
Using the “Distributed Array” (Darray) Datatype

```
int gsizes[2], distribs[2], dargs[2], psizes[2];

gsizes[0] = m;    /* no. of rows in global array */
gsizes[1] = n;    /* no. of columns in global array*/

distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;

dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;

psizes[0] = 2; /* no. of processes in vertical dimension
                  of process grid */
psizes[1] = 3; /* no. of processes in horizontal dimension
                  of process grid */
```



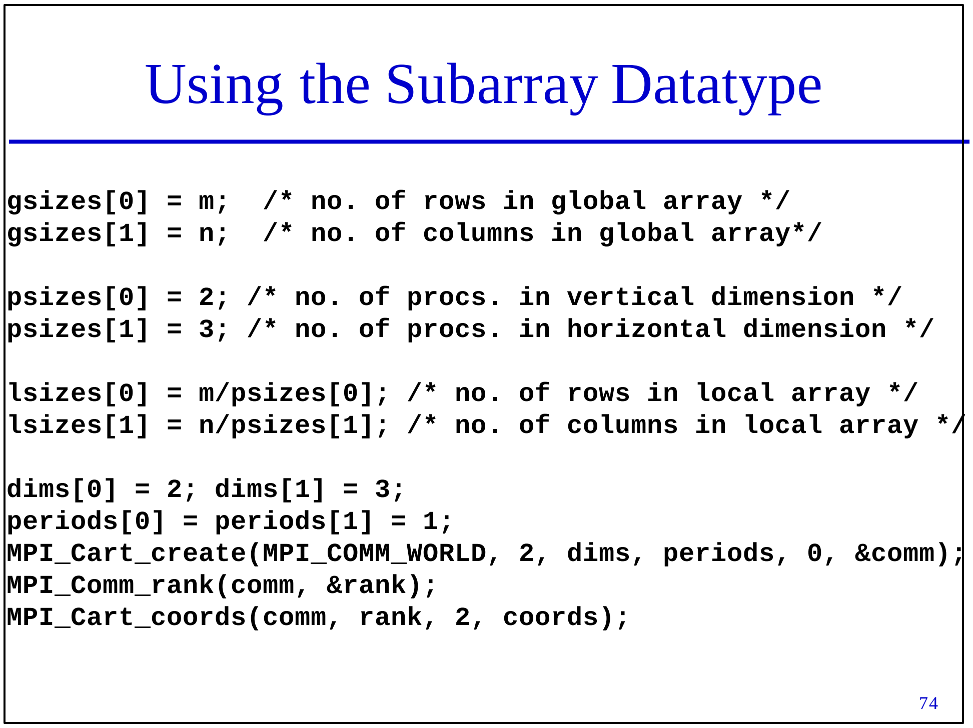
Darray Continued

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,
               psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                  MPI_INFO_NULL);

local_array_size = num_local_rows * num_local_cols;
MPI_File_write_all(fh, local_array, local_array_size,
               MPI_FLOAT, &status);

MPI_File_close(&fh);
```



A Word of Warning about Darray

- The darray datatype assumes a very specific definition of data distribution -- the exact definition as in HPF
- For example, if the array size is not divisible by the number of processes, darray calculates the block size using a *ceiling* division (20 / 6 = 4)
- darray assumes a row-major ordering of processes in the logical grid, as assumed by cartesian process topologies in MPI-1
- If your application uses a different definition for data distribution or logical grid ordering, you cannot use darray. Use subarray instead.



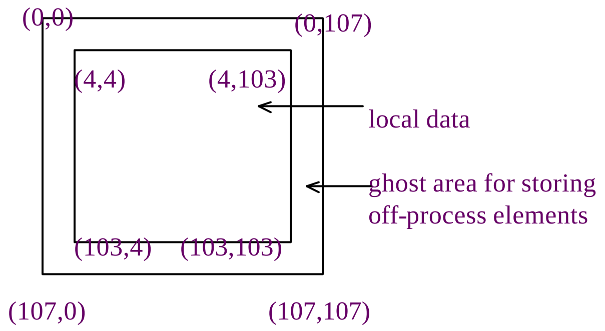
Using the Subarray Datatype

```
gsizes[0] = m;  /* no. of rows in global array */
gsizes[1] = n;  /* no. of columns in global array*/

psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

lsizes[0] = m/psizes[0]; /* no. of rows in local array */
lsizes[1] = n/psizes[1]; /* no. of columns in local array */

dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
```

Subarray Datatype contd.

```
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
              MPI_INFO_NULL);
local_array_size = lsizes[0] * lsizes[1];
MPI_File_write_all(fh, local_array, local_array_size,
               MPI_FLOAT, &status);
```

Local Array with Ghost Area in Memory

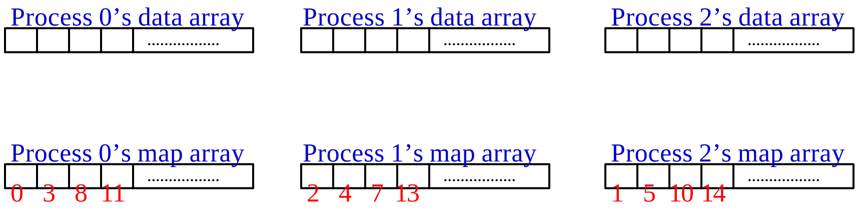


- Use a subarray datatype to describe the noncontiguous layout in memory
- Pass this datatype as argument to `MPI_File_write_all`

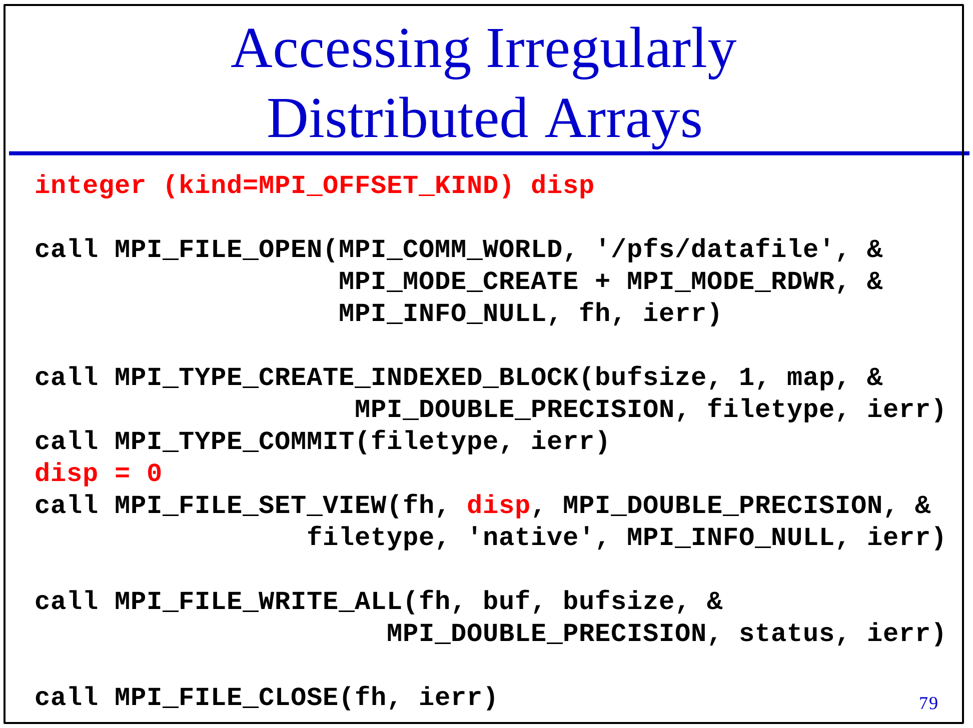
Local Array with Ghost Area

```
memsizes[0] = lsizes[0] + 8;
   /* no. of rows in allocated array */
memsizes[1] = lsizes[1] + 8;
   /* no. of columns in allocated array */
start_indices[0] = start_indices[1] = 4;
   /* indices of the first element of the local array
      in the allocated array */

MPI_Type_create_subarray(2, memsizes, lsizes,
         start_indices, MPI_ORDER_C, MPI_FLOAT, &memtype);
MPI_Type_commit(&memtype);

/* create filetype and set file view exactly as in the
   subarray example */

MPI_File_write_all(fh, local_array, 1, memtype, &status);
```

Accessing Irregularly Distributed Arrays

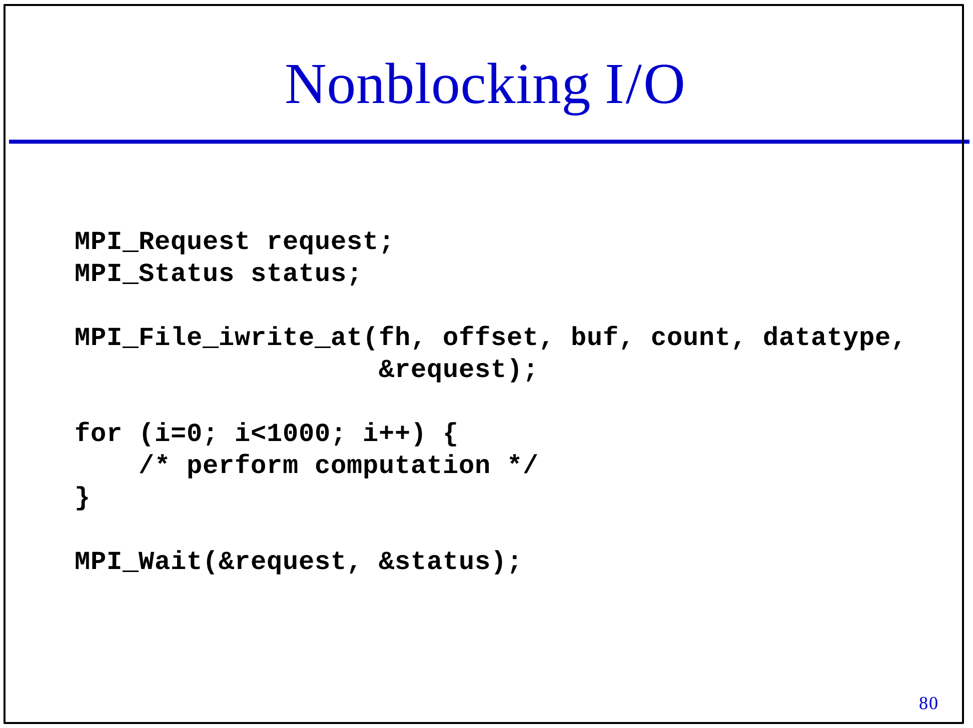


The map array describes the location of each element of the data array in the common file



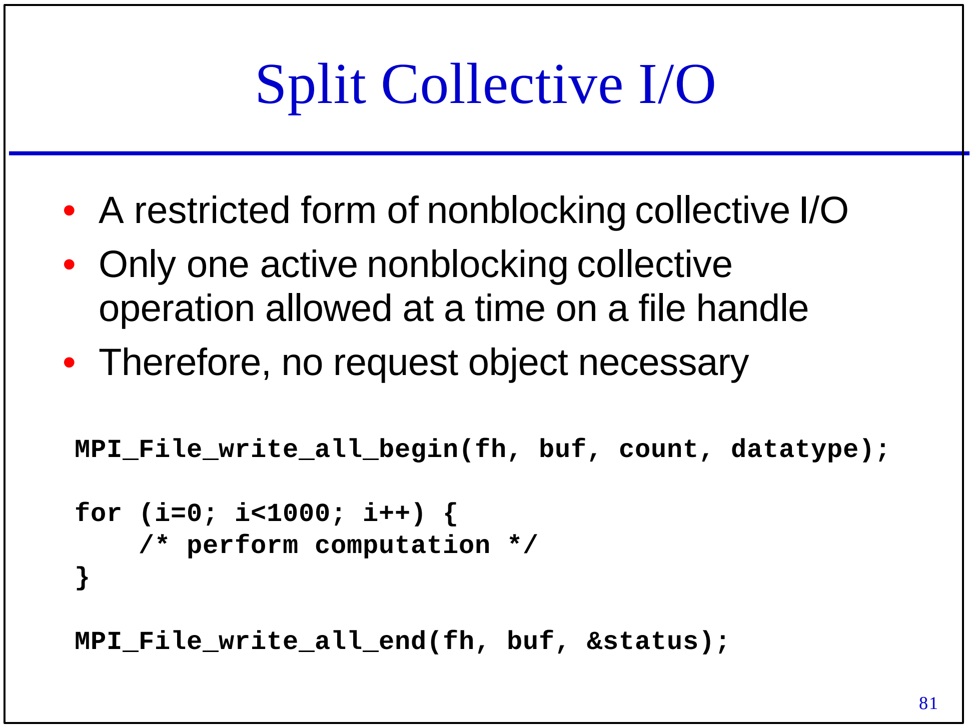
Accessing Irregularly Distributed Arrays

```
integer (kind=MPI_OFFSET_KIND) disp

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
                   MPI_MODE_CREATE + MPI_MODE_RDWR, &
                   MPI_INFO_NULL, fh, ierr)

call MPI_TYPE_CREATE_INDEXED_BLOCK(bufsize, 1, map, &
                    MPI_DOUBLE_PRECISION, filetype, ierr)
call MPI_TYPE_COMMIT(filetype, ierr)
disp = 0
call MPI_FILE_SET_VIEW(fh, disp, MPI_DOUBLE_PRECISION, &
                 filetype, 'native', MPI_INFO_NULL, ierr)

call MPI_FILE_WRITE_ALL(fh, buf, bufsize, &
                      MPI_DOUBLE_PRECISION, status, ierr)

call MPI_FILE_CLOSE(fh, ierr)
```



Nonblocking I/O

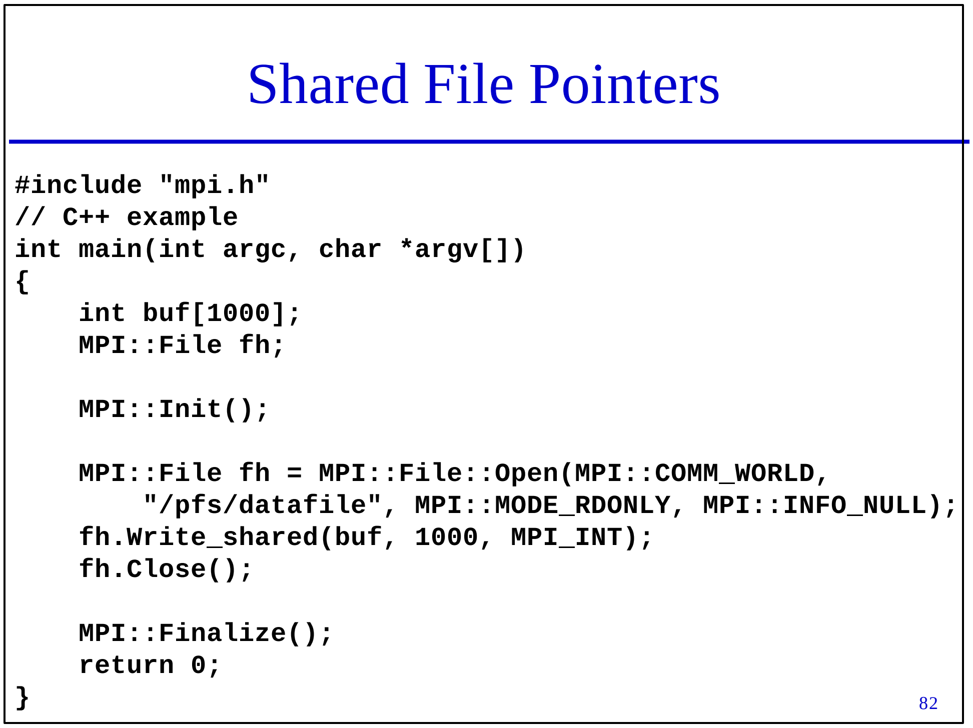
```
MPI_Request request;
MPI_Status status;

MPI_File_iwrite_at(fh, offset, buf, count, datatype,
                   &request);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_Wait(&request, &status);
```



Split Collective I/O

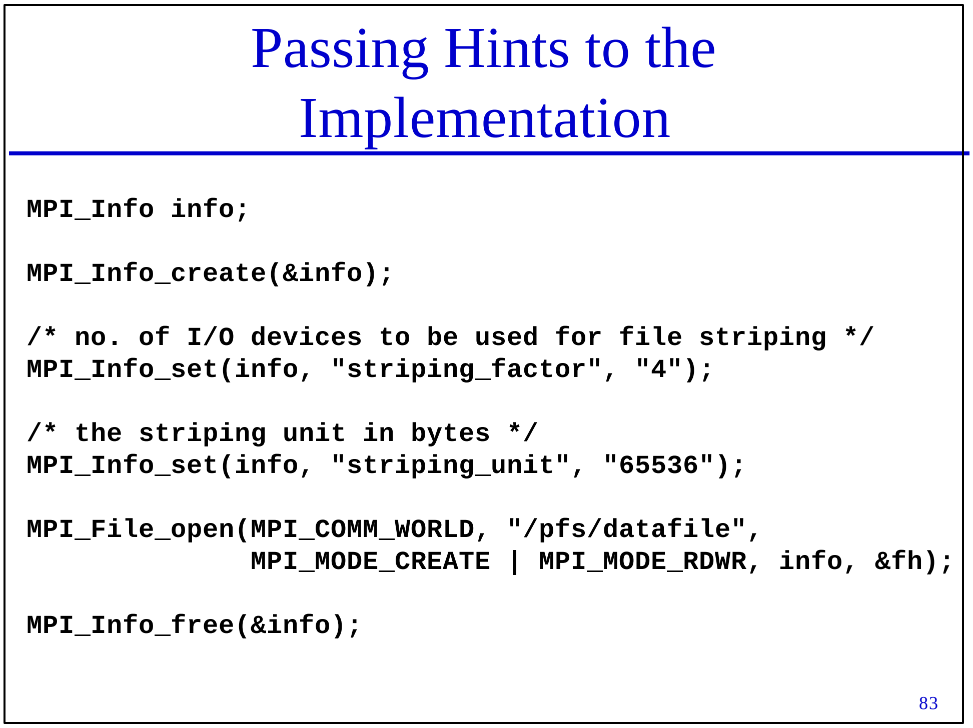
- A restricted form of nonblocking collective I/O
- Only one active nonblocking collective operation allowed at a time on a file handle
- Therefore, no request object necessary

```
MPI_File_write_all_begin(fh, buf, count, datatype);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_File_write_all_end(fh, buf, &status);
```



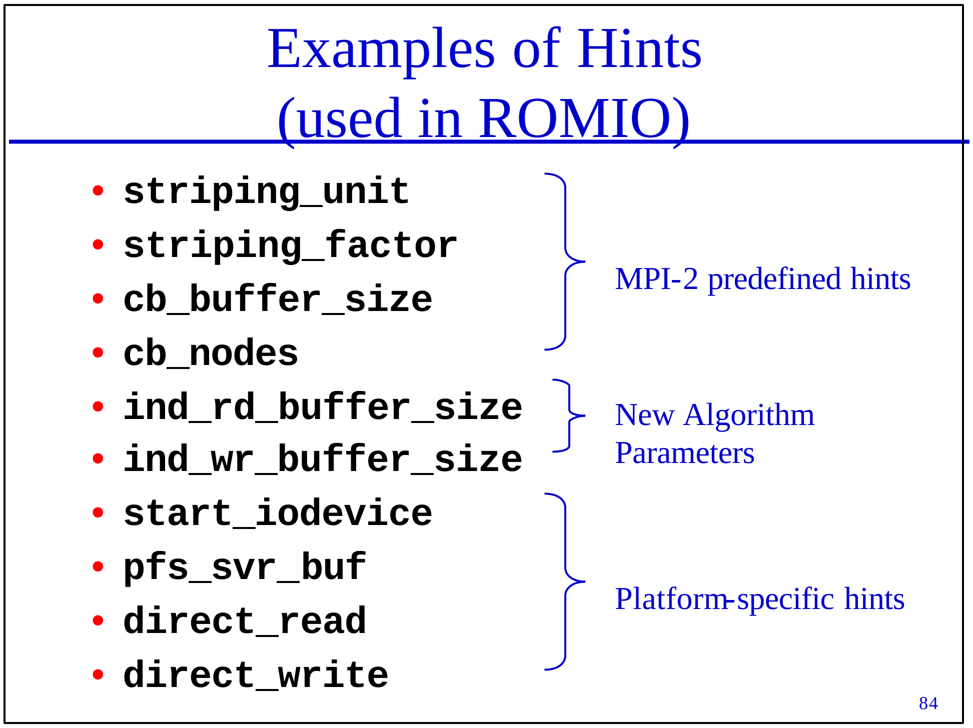
Shared File Pointers

```
#include "mpi.h"
// C++ example
int main(int argc, char *argv[])
{
    int buf[1000];
    MPI::File fh;

    MPI::Init();

    MPI::File fh = MPI::File::Open(MPI::COMM_WORLD,
        "/pfs/datafile", MPI::MODE_RDONLY, MPI::INFO_NULL);
    fh.Write_shared(buf, 1000, MPI_INT);
    fh.Close();

    MPI::Finalize();
    return 0;
}
```



Passing Hints to the Implementation

```
MPI_Info info;

MPI_Info_create(&info);

/* no. of I/O devices to be used for file striping */
MPI_Info_set(info, "striping_factor", "4");

/* the striping unit in bytes */
MPI_Info_set(info, "striping_unit", "65536");

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);

MPI_Info_free(&info);
```



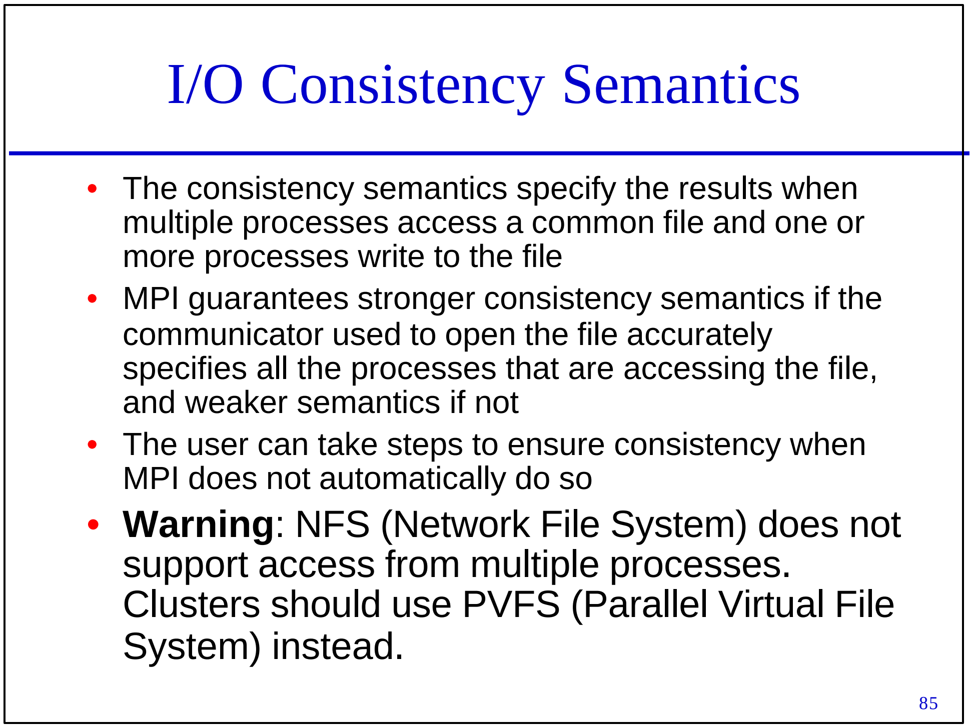
Examples of Hints (used in ROMIO)

- **`striping_unit`**
- **`striping_factor`**
- **`cb_buffer_size`**
- **`cb_nodes`**
- **`ind_rd_buffer_size`**
- **`ind_wr_buffer_size`**
- **`start_iodevice`**
- **`pfs_svr_buf`**
- **`direct_read`**
- **`direct_write`**

MPI-2 predefined hints

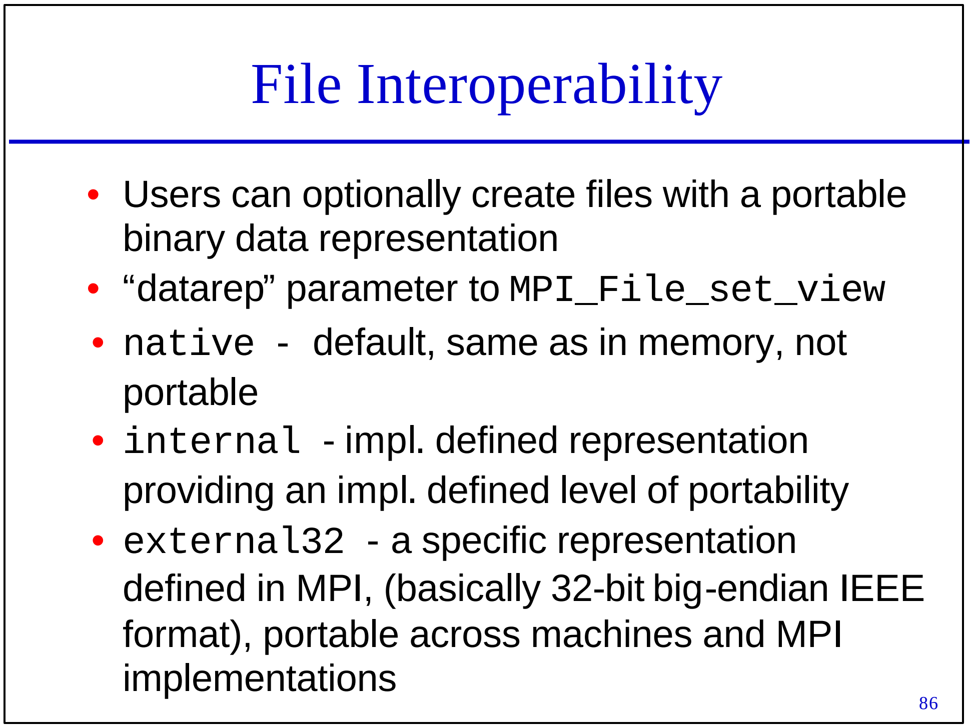
New Algorithm Parameters

Platform-specific hints



I/O Consistency Semantics

- The consistency semantics specify the results when multiple processes access a common file and one or more processes write to the file
- MPI guarantees stronger consistency semantics if the communicator used to open the file accurately specifies all the processes that are accessing the file, and weaker semantics if not
- The user can take steps to ensure consistency when MPI does not automatically do so
- **Warning**: NFS (Network File System) does not support access from multiple processes. Clusters should use PVFS (Parallel Virtual File System) instead.



File Interoperability

- Users can optionally create files with a portable binary data representation
- “datarep” parameter to `MPI_File_set_view`
- `native -` default, same as in memory, not portable
- `internal` - impl. defined representation providing an impl. defined level of portability
- `external32` - a specific representation defined in MPI, (basically 32-bit big-endian IEEE format), portable across machines and MPI implementations

General Guidelines for Achieving High I/O Performance

- Buy sufficient I/O hardware for the machine
- Use fast file systems (such as PVFS), not NFS-mounted home directories
- Do not perform I/O from one process only
- Make large requests wherever possible
- For noncontiguous requests, use derived datatypes and a single collective I/O call

Achieving High I/O Performance with MPI

- Any application as a particular “I/O access pattern” based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- See the SC98 paper in which we classify the different ways of expressing I/O access patterns in MPI-IO into four *levels*: level 0 — level 3 (http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Thakur447)
- We demonstrate how the user’s choice of *level* affects performance

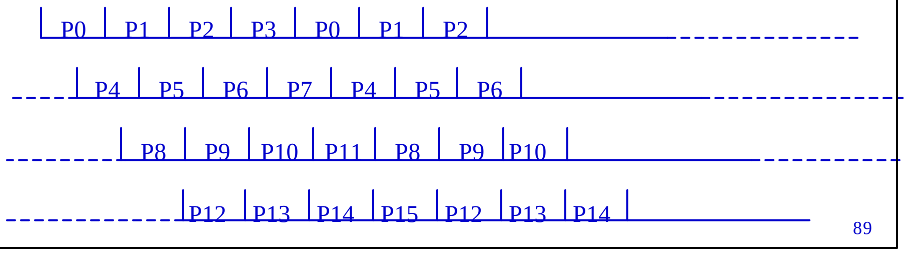
Example: Distributed Array Access

Large array distributed among 16 processes

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Each square represents a subarray in the memory of a single process

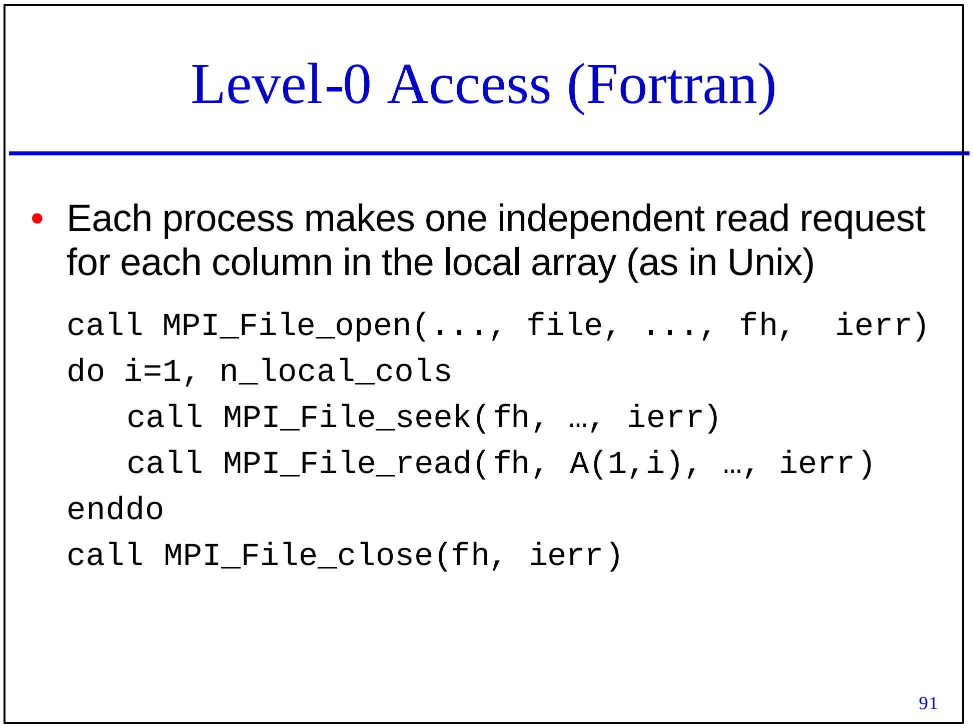
Access Pattern in the file



Level-0 Access (C)

- Each process makes one independent read request for each row in the local array (as in Unix)

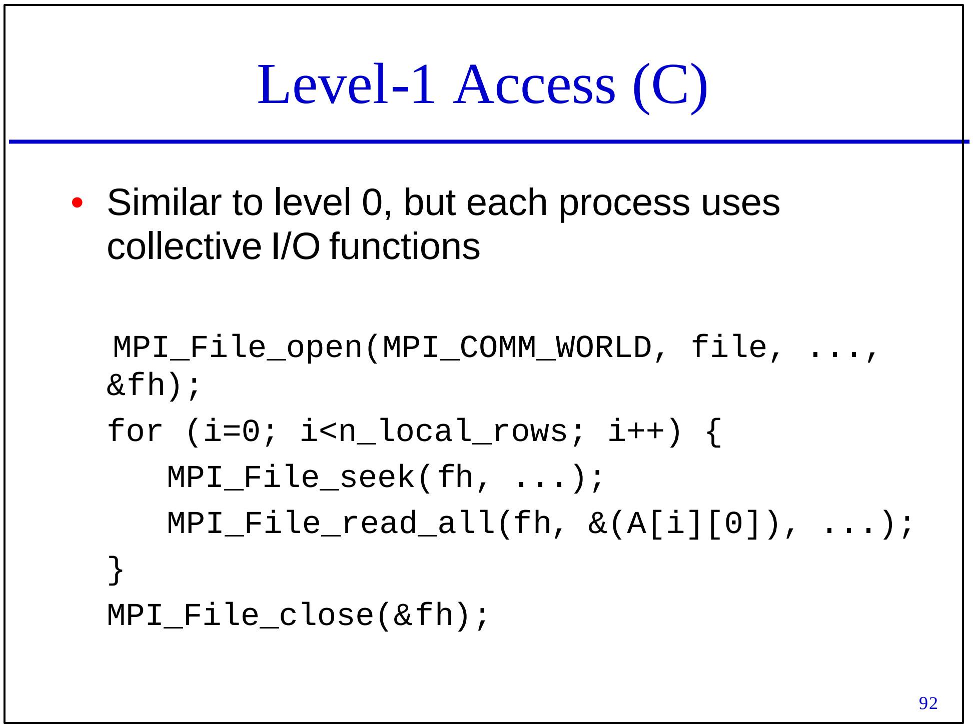
```
MPI_File_open(..., file, ..., &fh)
for (i=0; i<n_local_rows; i++) {
   MPI_File_seek(fh, ...);
   MPI_File_read(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```



Level-0 Access (Fortran)

- Each process makes one independent read request for each column in the local array (as in Unix)

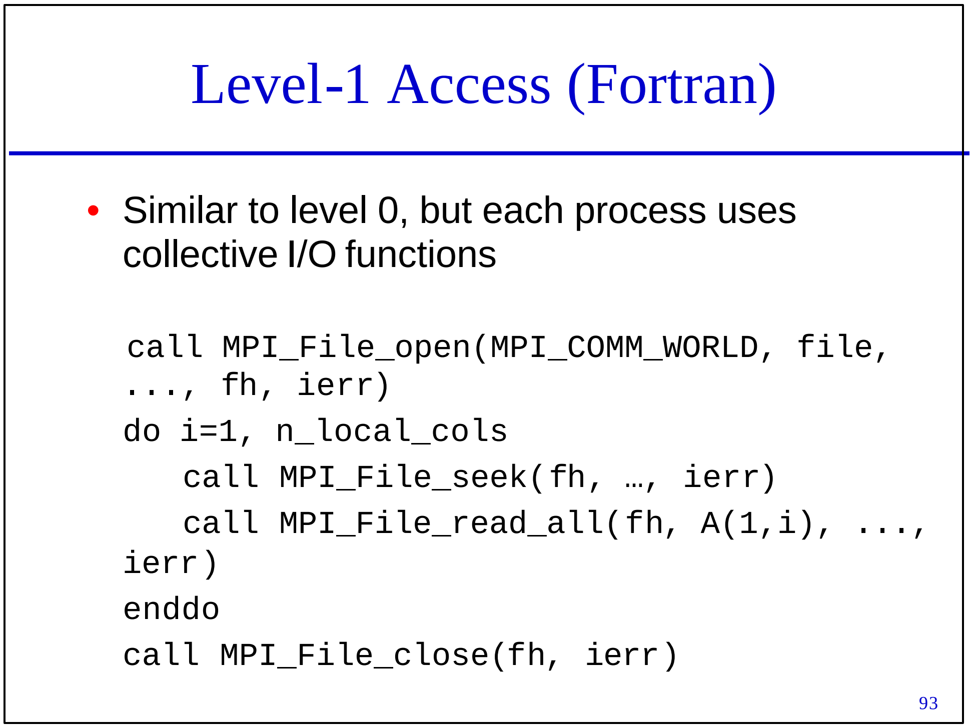
```
call MPI_File_open(..., file, ..., fh,  ierr)
do i=1, n_local_cols
   call MPI_File_seek(fh, …, ierr)
   call MPI_File_read(fh, A(1,i), …, ierr)
enddo
call MPI_File_close(fh, ierr)
```



Level-1 Access (C)

- Similar to level 0, but each process uses collective I/O functions

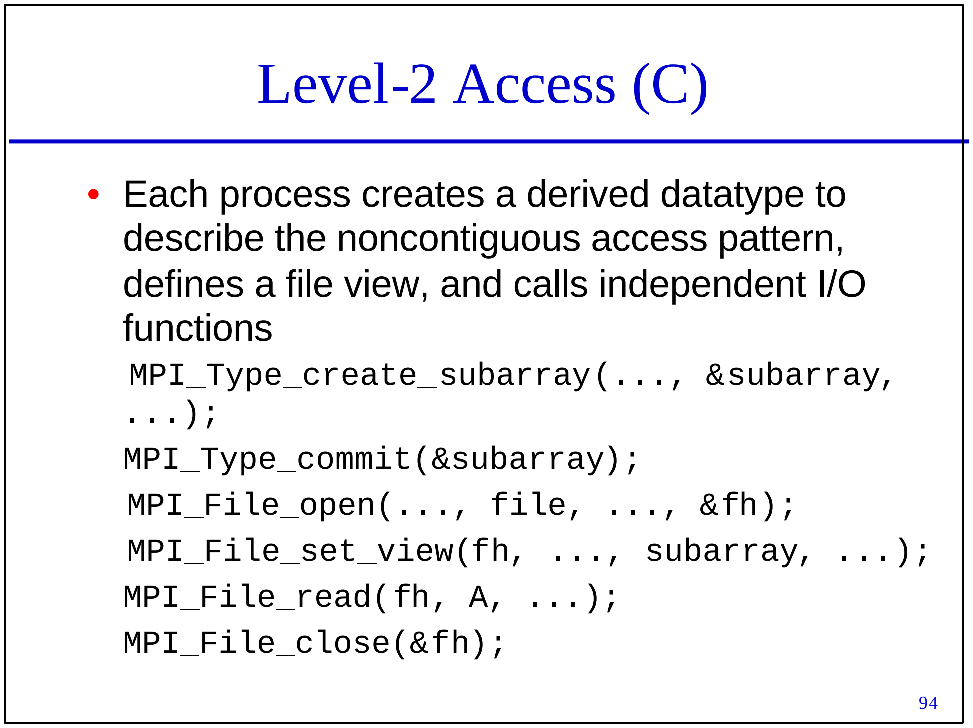
```
 MPI_File_open(MPI_COMM_WORLD, file, ...,
&fh);
for (i=0; i<n_local_rows; i++) {
   MPI_File_seek(fh, ...);
   MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```



Level-1 Access (Fortran)

- Similar to level 0, but each process uses collective I/O functions

```
call MPI_File_open(MPI_COMM_WORLD, file,
..., fh, ierr)
do i=1, n_local_cols
   call MPI_File_seek(fh, …, ierr)
   call MPI_File_read_all(fh, A(1,i), ...,
ierr)
enddo
call MPI_File_close(fh, ierr)
```



Level-2 Access (C)

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
MPI_Type_create_subarray(..., &subarray,
...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

Level-2 Access (Fortran)

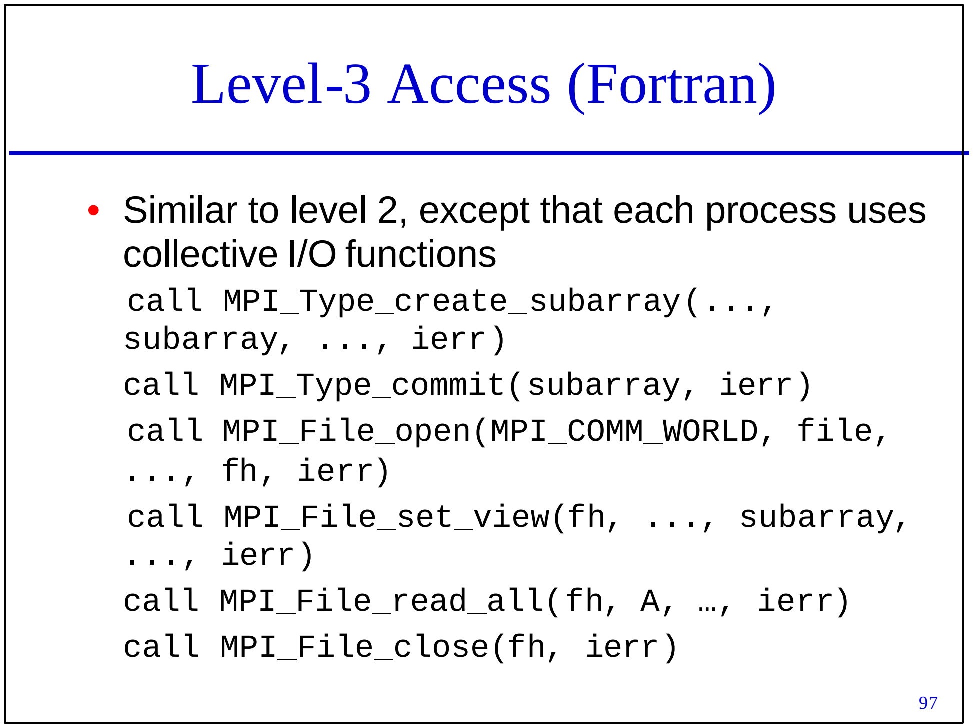
- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
call MPI_Type_create_subarray(..., subarray,
..., ierr)
call MPI_Type_commit(subarray, ierr)
call MPI_File_open(..., file, ..., fh, ierr)
call MPI_File_set_view(fh, ..., subarray,
..., ierr)
call MPI_File_read(fh, A, ..., ierr)
call MPI_File_close(fh, ierr)
```

Level-3 Access (C)

- Similar to level 2, except that each process uses collective I/O functions

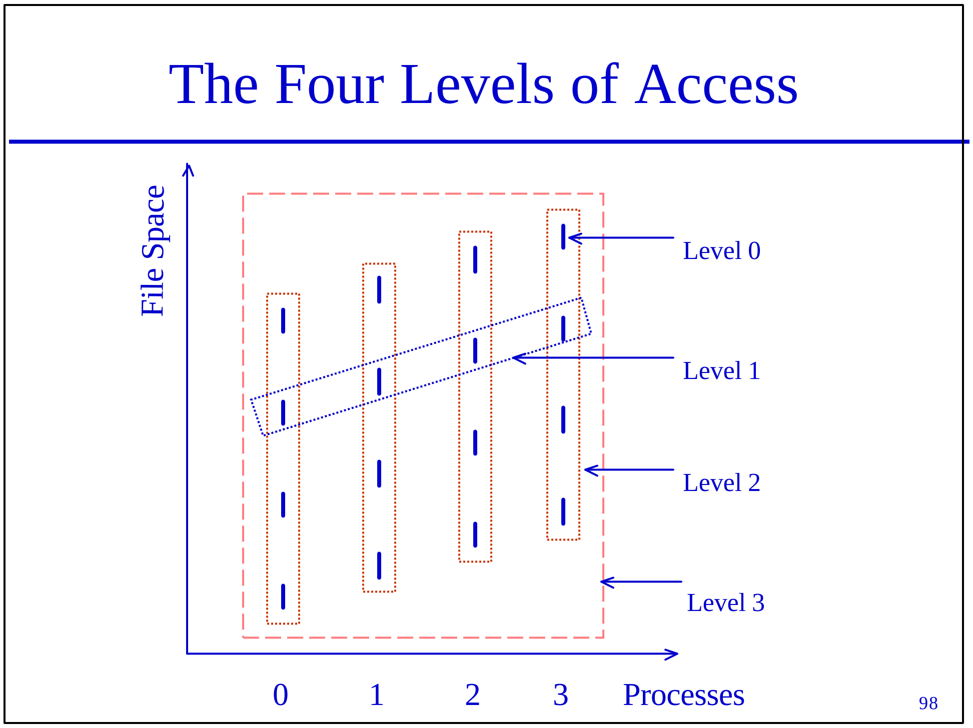
```
MPI_Type_create_subarray(...,
&subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file,
..., &fh);
MPI_File_set_view(fh, ..., subarray,
...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```



Level-3 Access (Fortran)

- Similar to level 2, except that each process uses collective I/O functions

```
call MPI_Type_create_subarray(...,
subarray, ..., ierr)
call MPI_Type_commit(subarray, ierr)
call MPI_File_open(MPI_COMM_WORLD, file,
..., fh, ierr)
call MPI_File_set_view(fh, ..., subarray,
..., ierr)
call MPI_File_read_all(fh, A, …, ierr)
call MPI_File_close(fh, ierr)
```



The Four Levels of Access

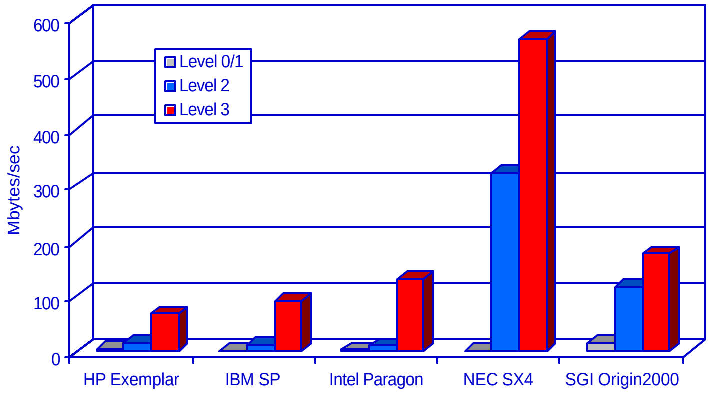
Optimizations

- Given complete access information, an implementation can perform optimizations such as:
  - Data Sieving: Read large chunks and extract what is really needed
  - Collective I/O: Merge requests of different processes into larger requests
  - Improved prefetching and caching

Performance Results

- Distributed array access
- Unstructured code from Sandia
- On five different parallel machines:
  - HP Exemplar
  - IBM SP
  - Intel Paragon
  - NEC SX-4
  - SGI Origin2000

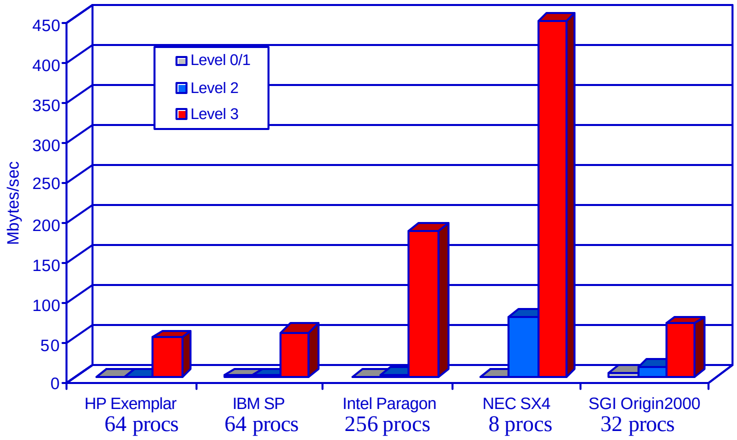
Distributed Array Access: Read Bandwidth



64 procs 64 procs 256 procs 8 procs 32 procs

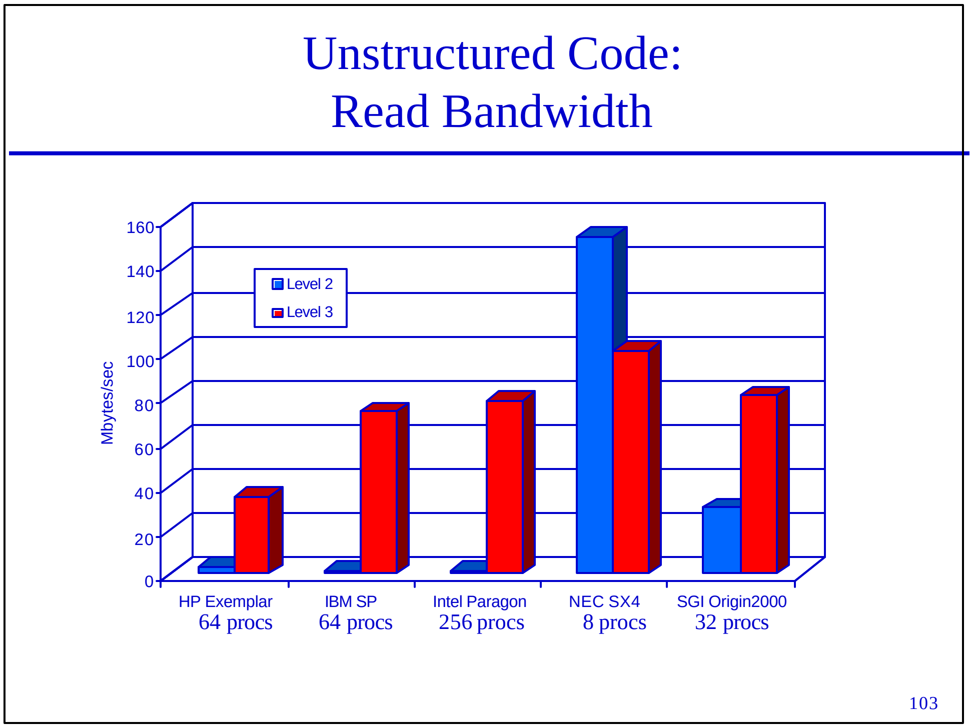
Array size: 512 x 512 x 512

Distributed Array Access: Write Bandwidth



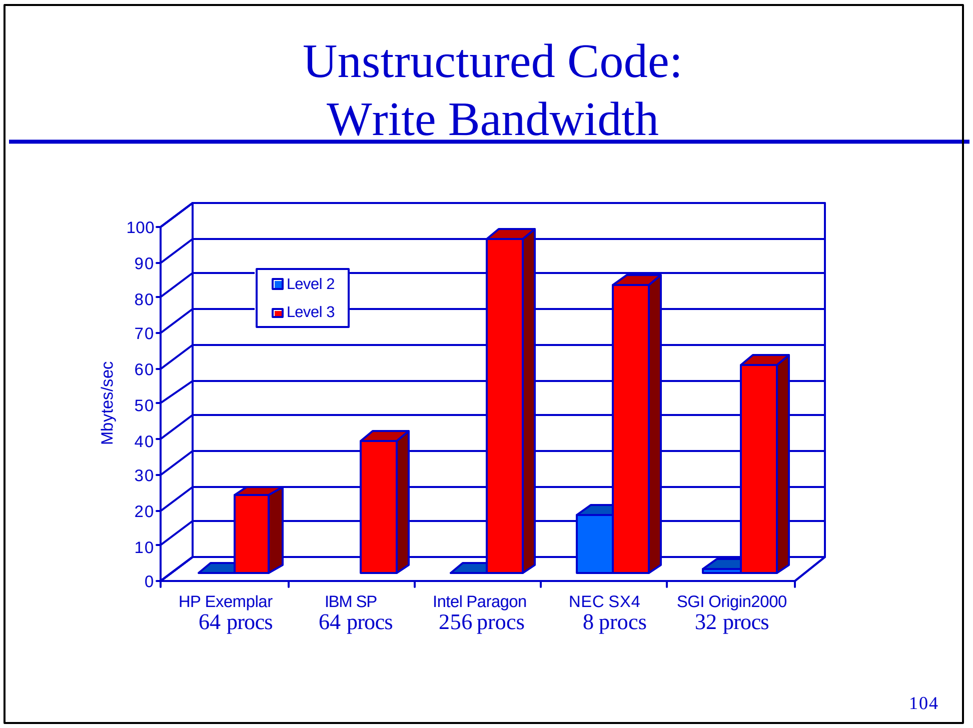
64 procs 64 procs 256 procs 8 procs 32 procs

Array size: 512 x 512 x 512



Unstructured Code: Read Bandwidth

103

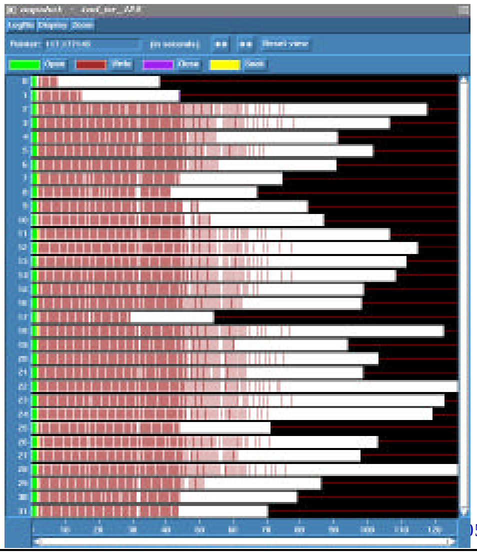


Unstructured Code: Write Bandwidth

104

Independent Writes

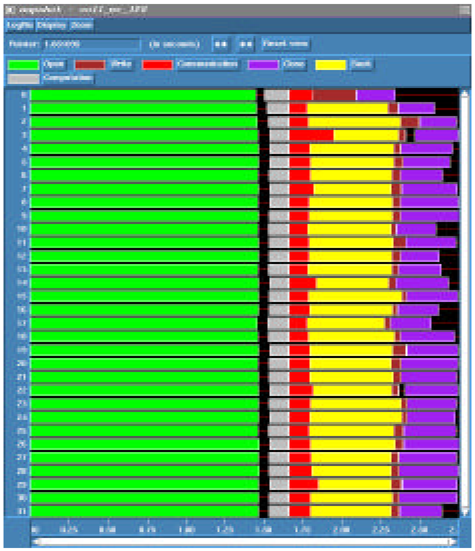
- On Paragon
- Lots of seeks and small writes
- Time shown = 130 seconds



05

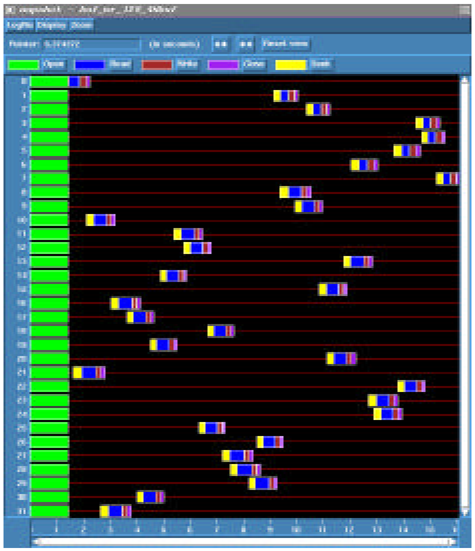
Collective Write

- On Paragon
- Computation and communication precede seek and write
- Time shown = 2.75 seconds



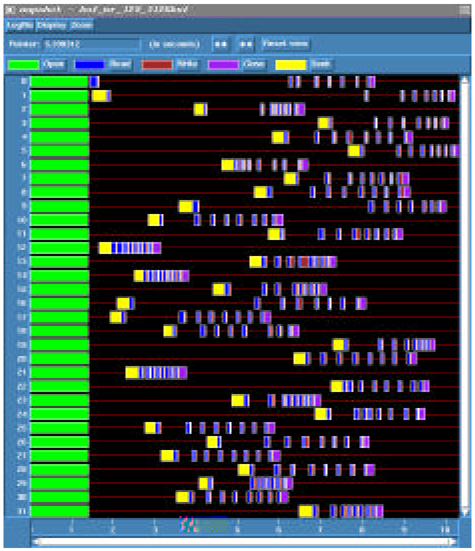
Independent Writes with Data Sieving

- On Paragon
- Access data in large “blocks” and extract needed data
- Requires lock, read, modify, write, unlock for writes
- 4 MB blocks
- Time = 16 sec.



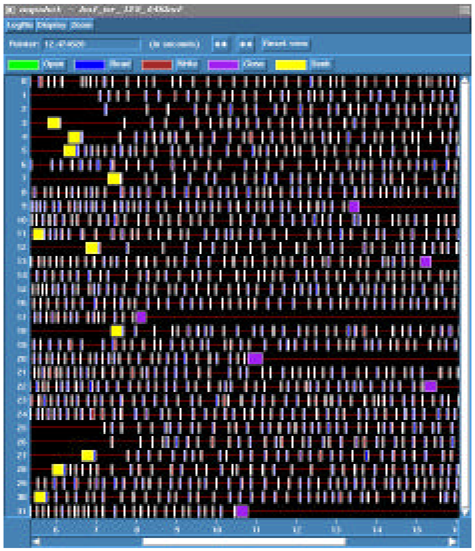
Changing the Block Size

- Smaller blocks mean less contention, therefore more parallelism
- 512 KB blocks
- Time = 10.2 seconds



Data Sieving with Small Blocks

- If the block size is too small, however, the increased parallelism doesn't make up for the many small writes
- 64 KB blocks
- Time = 21.5 seconds



Common Errors

- Not defining file offsets as `MPI_Offset` in C and `integer (kind=MPI_OFFSET_KIND)` in Fortran (or perhaps `integer*8` in Fortran 77)
- In Fortran, passing the offset or displacement directly as a constant (e.g., 0) in the absence of function prototypes (F90 mpi module)
- Using darray datatype for a block distribution other than the one defined in darray (e.g., floor division)
- filetype defined using offsets that are not monotonically nondecreasing, e.g., 0, 3, 8, 4, 6. (happens in irregular applications)

Summary

- MPI I/O has many features that can help users achieve high performance
- The most important of these features are the ability to specify noncontiguous accesses, the collective I/O functions, and the ability to pass hints to the implementation
- Users must use the above features!
- In particular, when accesses are noncontiguous, users must create derived datatypes, define file views, and use the collective I/O functions

Dynamic Process Management in MPI

- Standard way of starting processes in PVM
- Not so necessary in MPI
- Useful in assembling complex distributed applications
- Issues
  - maintaining simplicity, flexibility, and correctness
  - interaction with operating system, resource manager, and process manager
  - connecting independently started processes

Starting New MPI Processes

- MPI_Comm_spawn
  - Starts n new processes
  - Collective over communicator
    - Necessary for scalability
  - Returns an intercommunicator
    - Does *not* change MPI_COMM_WORLD
  - "SPMD" (Single Program Multiple Data)
- MPI_Comm_spawn_multiple
  - Link MPI_Comm_spawn
  - "MPMD" (Multiple Program Multiple Data)

The MPI Environment

- MPI does not specify a particular process management environment
  - You can use whatever you have (Condor, LoadLeveler, PBS, etc.)
  - No particular system is defined
    - MPI Forum attempted to define an abstract interface to third-party process managers, but could no consensus reached
- Still need to pass information to process management system
  - Led to MPI_Info

Communicating Data to (and through) MPI

- MPI-2 defines a new object, *MPI_Info*
- Provides an extensible list of key=value pairs
- Used in I/O, One-sided, and Dynamic to package variable, optional types of arguments that may not be standard

Using MPI_Info in Fortran

- Example use for MPI_Comm_spawn:

```
numslaves = 10
call MPI_INFO_CREATE( spawninfo, ierr )
call MPI_INFO_SET( spawninfo, 'host', 'nome.mcs.anl.gov', ierr )
call MPI_INFO_SET( spawninfo, 'path', '/home/kosh/progs', ierr )
call MPI_INFO_SET( spawninfo, 'wdir', '/home/kosh/tmp', ierr )
call MPI_COMM_SPAWN( 'slave', MPI_ARGV_NULL, numslaves, &
                     spawninfo, 0, MPI_COMM_WORLD,      &
                     slavecomm, MPI_ERRCODES_IGNORE, ierr )
call MPI_INFO_FREE( spawninfo, ierr )
```

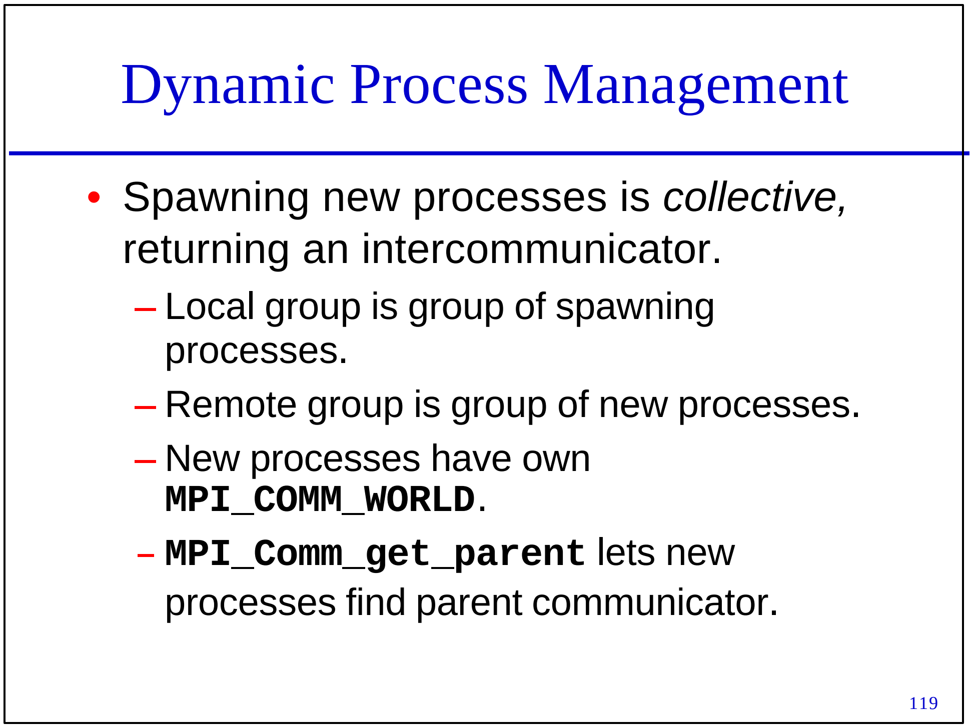
Using MPI_Info in C

- Example use for MPI_Comm_spawn:

```
numslaves = 10;
MPI_Info_create( &spawninfo );
MPI_Info_set( spawninfo, “host”, “nome.mcs.anl.gov” );
MPI_Info_set( spawninfo, “path”, “/home/kosh/progs” );
MPI_Info_set( spawninfo, “wdir”, “/home/kosh/tmp” );
MPI_Comm_spawn( “slave”, MPI_ARGV_NULL, numslaves,
                      spawninfo, 0, MPI_COMM_WORLD,
                      &slavecomm, MPI_ERRCODES_IGNORE );
MPI_Info_free( &spawninfo );
```

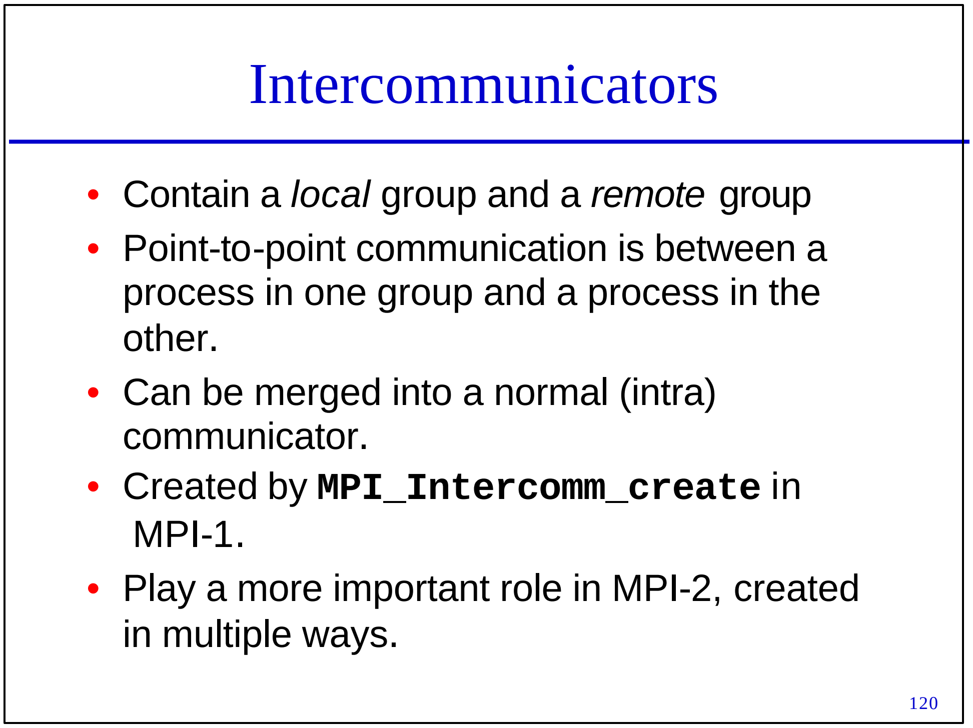
Other Process Management Features

- MPI_Comm_connect and MPI_Comm_accept allow two running MPI programs to connect and interoperate
  - Not intended for client/server applications
  - Useful in assembling complex distributed applications
- MPI_Join allows the use of a TCP socket to connect two applications



Dynamic Process Management

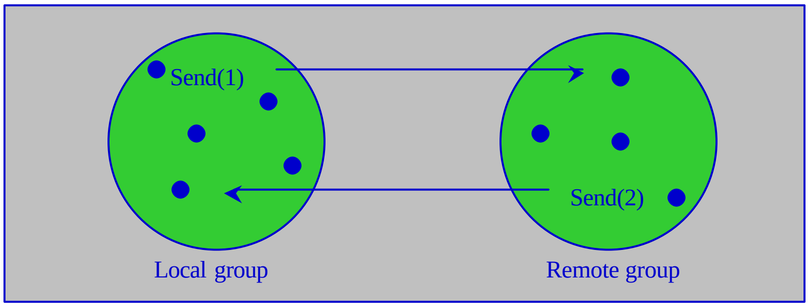
- Spawning new processes is *collective,* returning an intercommunicator.
  - Local group is group of spawning processes.
  - Remote group is group of new processes.
  - New processes have own **`MPI_COMM_WORLD`**.
  - **`MPI_Comm_get_parent`** lets new processes find parent communicator.



Intercommunicators

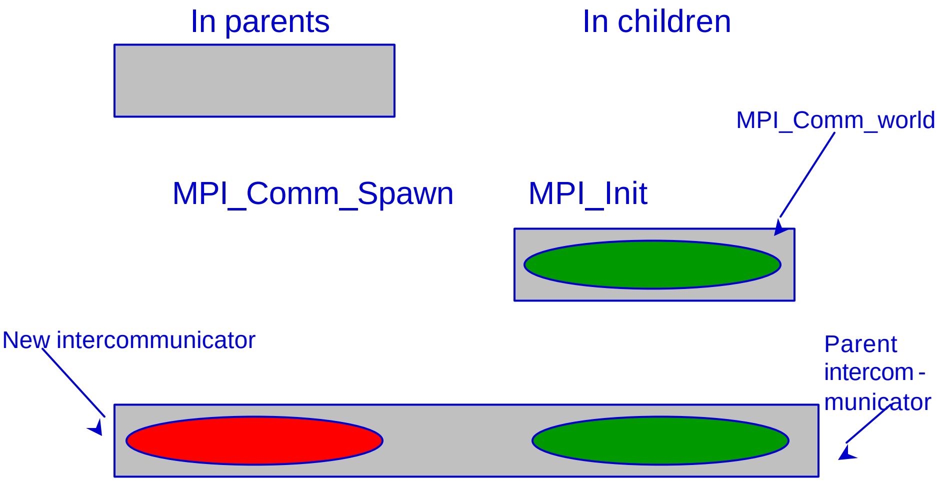
- Contain a *local* group and a *remote* group
- Point-to-point communication is between a process in one group and a process in the other.
- Can be merged into a normal (intra) communicator.
- Created by **`MPI_Intercomm_create`** in MPI-1.
- Play a more important role in MPI-2, created in multiple ways.

Intercommunicators

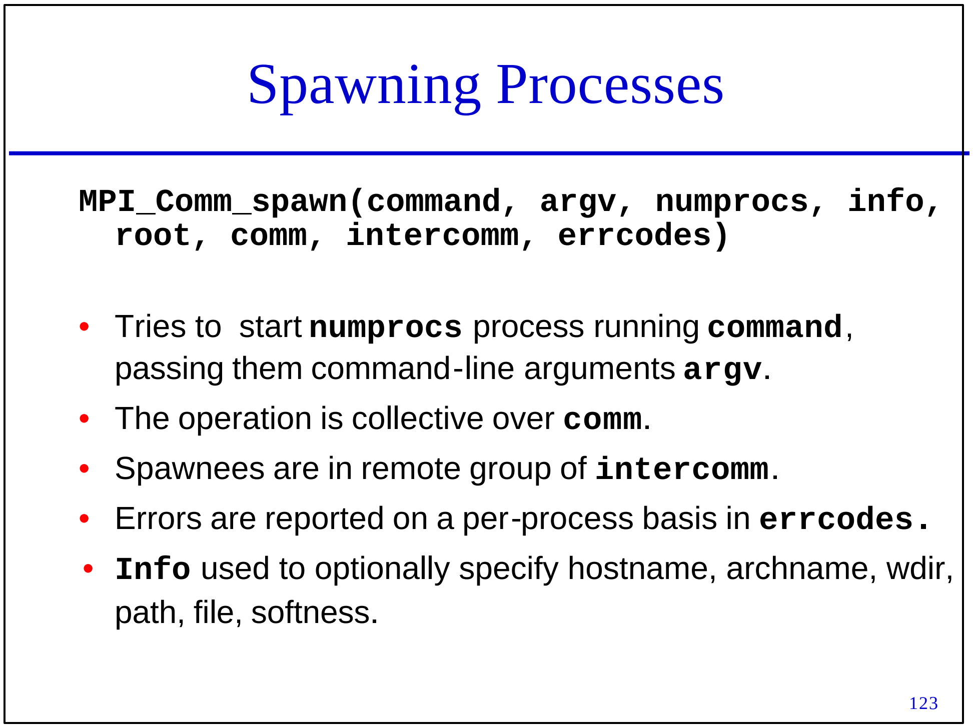


121

Spawning New Processes



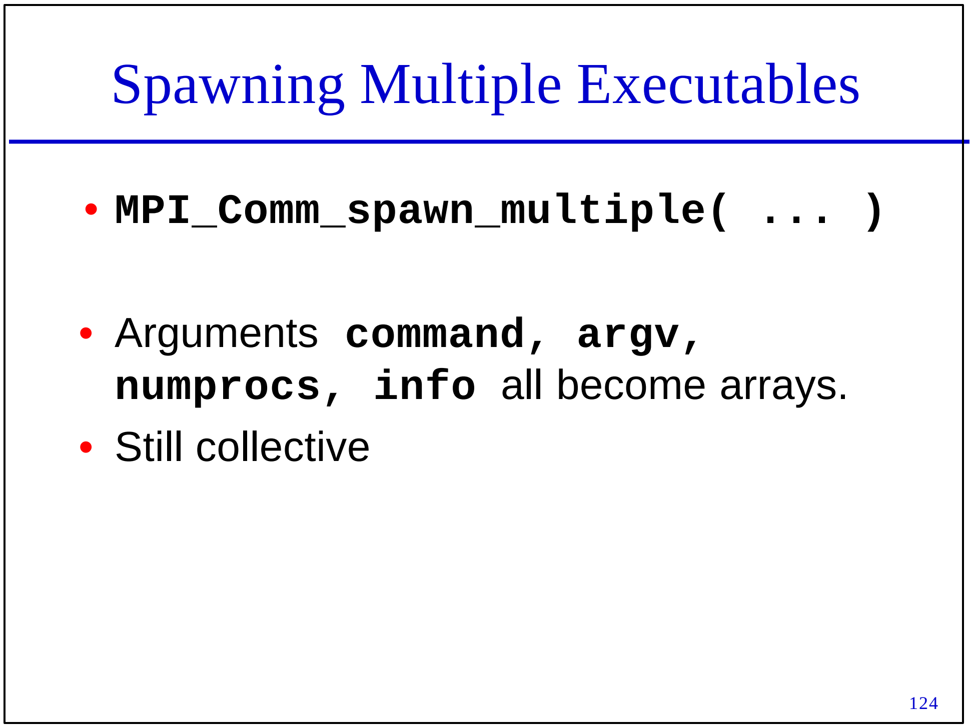
122



Spawning Processes

```
MPI_Comm_spawn(command, argv, numprocs, info,
  root, comm, intercomm, errcodes)
```

- Tries to start **`numprocs`** process running **`command`**, passing them command-line arguments **`argv`**.
- The operation is collective over **`comm`**.
- Spawnees are in remote group of **`intercomm`**.
- Errors are reported on a per-process basis in **`errcodes.`**
- **`Info`** used to optionally specify hostname, archname, wdir, path, file, softness.



Spawning Multiple Executables

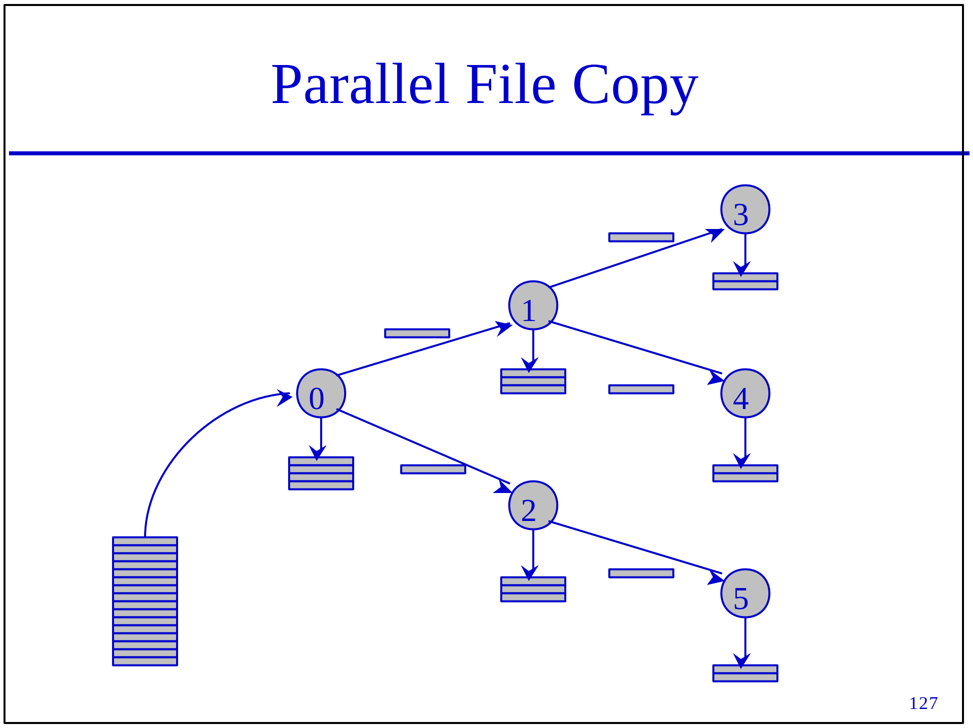
- **`MPI_Comm_spawn_multiple( ... )`**
- Arguments **`command, argv, numprocs, info`** all become arrays.
- Still collective

In the Children

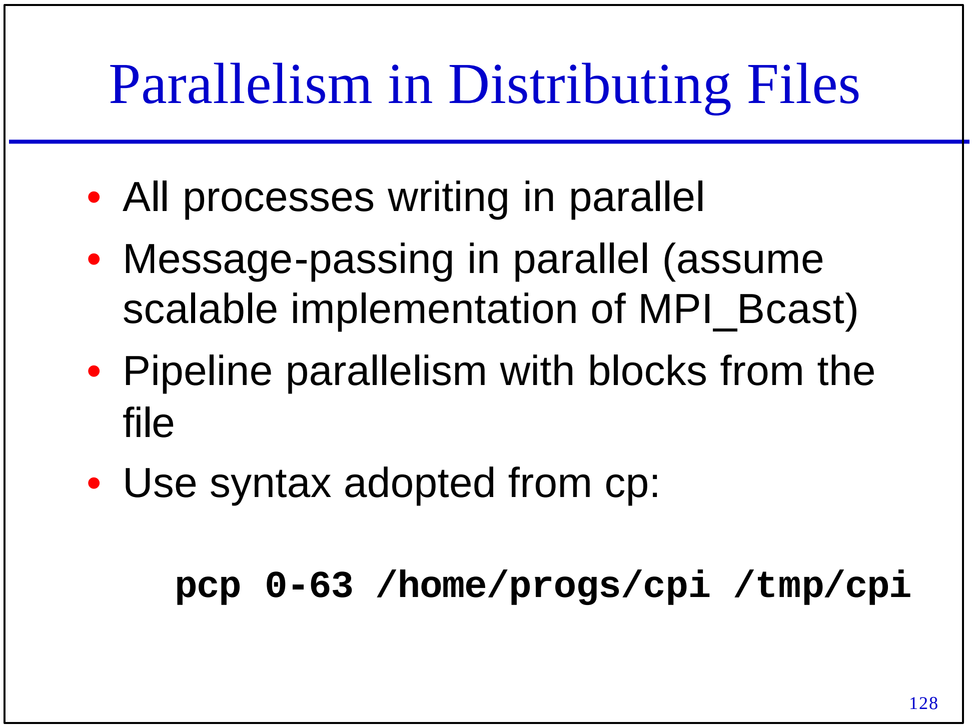
- **`MPI_Init`** (only MPI programs can be spawned)
- **`MPI_COMM_WORLD`** is processes spawned with one call to **`MPI_Comm_spawn.`**
- **`MPI_Comm_get_parent`** obtains parent intercommunicator.
  - Same as intracommunicator returned by **`MPI_Comm_spawn`** in parents.
  - Remote group is spawners.
  - Local group is those spawned.

Manager-Worker Example

- Single manager process decides how many workers to create and which executable they should run.
- Manager spawns n workers, and addresses them as 0, 1, 2, ..., n-1 in new intercomm.
- Workers address each other as 0, 1, ... n-1 in **`MPI_COMM_WORLD`**, address manager as 0 in parent intercomm.
- One can find out how many processes can usefully be spawned (MPI_UNIVERSE_SIZE attribute of MPI_COMM_WORLD).



Parallel File Copy



Parallelism in Distributing Files

- All processes writing in parallel
- Message-passing in parallel (assume scalable implementation of MPI_Bcast)
- Pipeline parallelism with blocks from the file
- Use syntax adopted from cp:

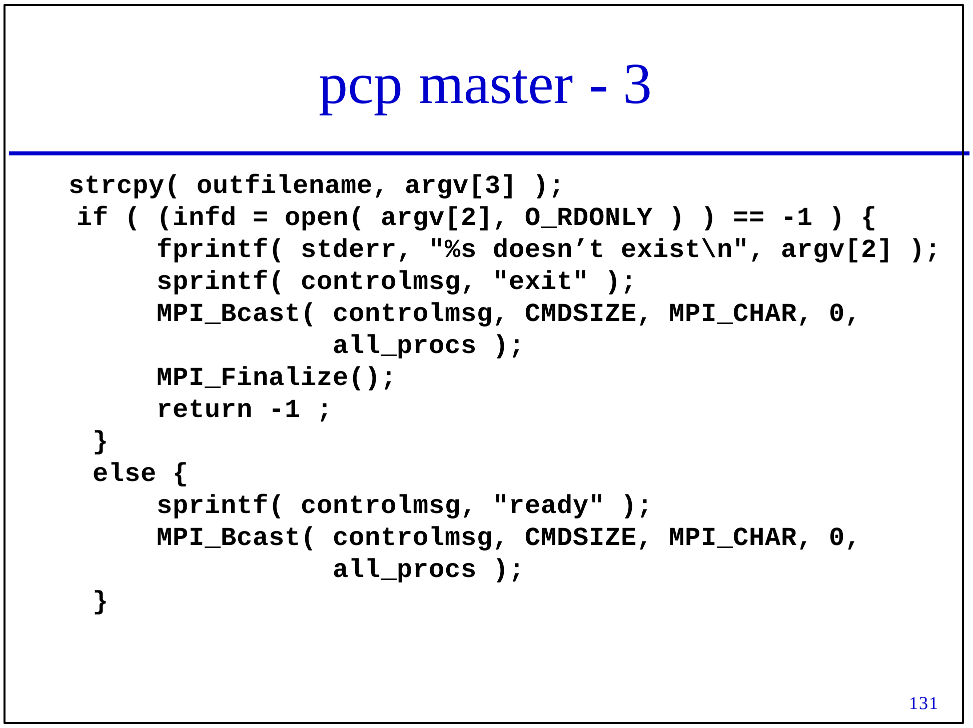
```
pcp 0-63 /home/progs/cpi /tmp/cpi
```

Code for pcp Master -1

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BUFSIZE    256*1024
#define CMDSIZE    80
int main( int argc, char *argv[] )
{
    int      num_hosts, mystatus, allstatus, done, numread;
    int      infd, outfd;
    char     outfilename[MAXPATHLEN], controlmsg[CMDSIZE];
    char     buf[BUFSIZE];
    char     soft_limit[20];
    MPI_Info hostinfo;
    MPI_Comm pcpslaves, all_procs;

    MPI_Init( &argc, &argv );
```

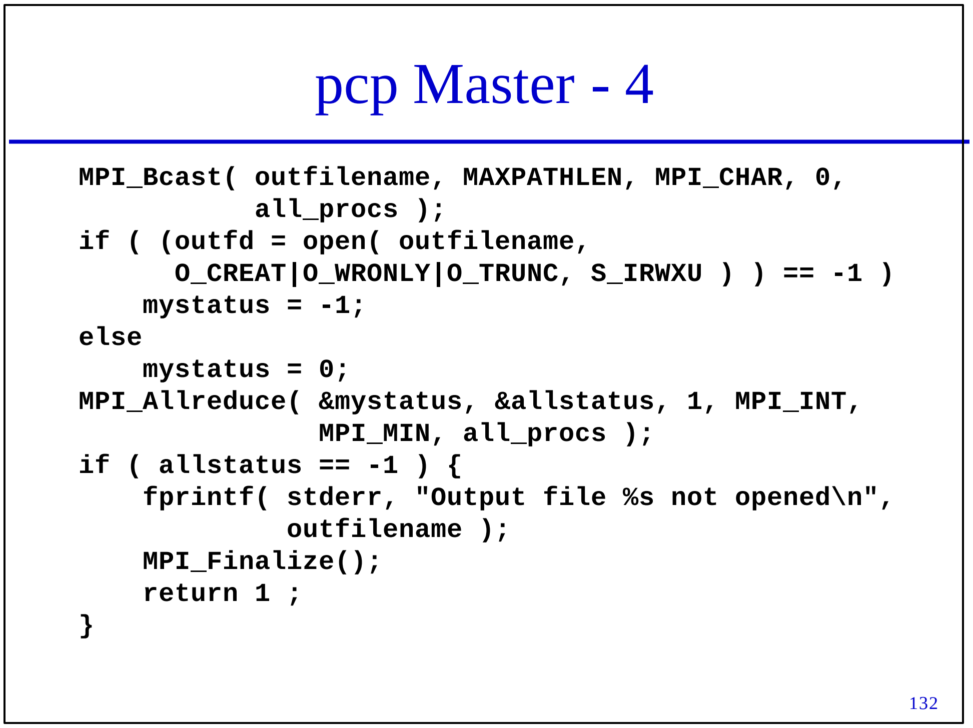
pcp Master - 2

```
    makehostlist( argv[1], "targets", &num_hosts );
    MPI_Info_create( &hostinfo );
    MPI_Info_set( hostinfo, "file", "targets" );
    sprintf( soft_limit, "0:%d", num_hosts );
    MPI_Info_set( hostinfo, "soft", soft_limit );
    MPI_Comm_spawn( "pcp_slave", MPI_ARGV_NULL, num_hosts,
                    hostinfo, 0, MPI_COMM_SELF, &pcpslaves,
                    MPI_ERRCODES_IGNORE );
    MPI_Info_free( &hostinfo );
    MPI_Intercomm_merge( pcpslaves, 0, &all_procs );
```



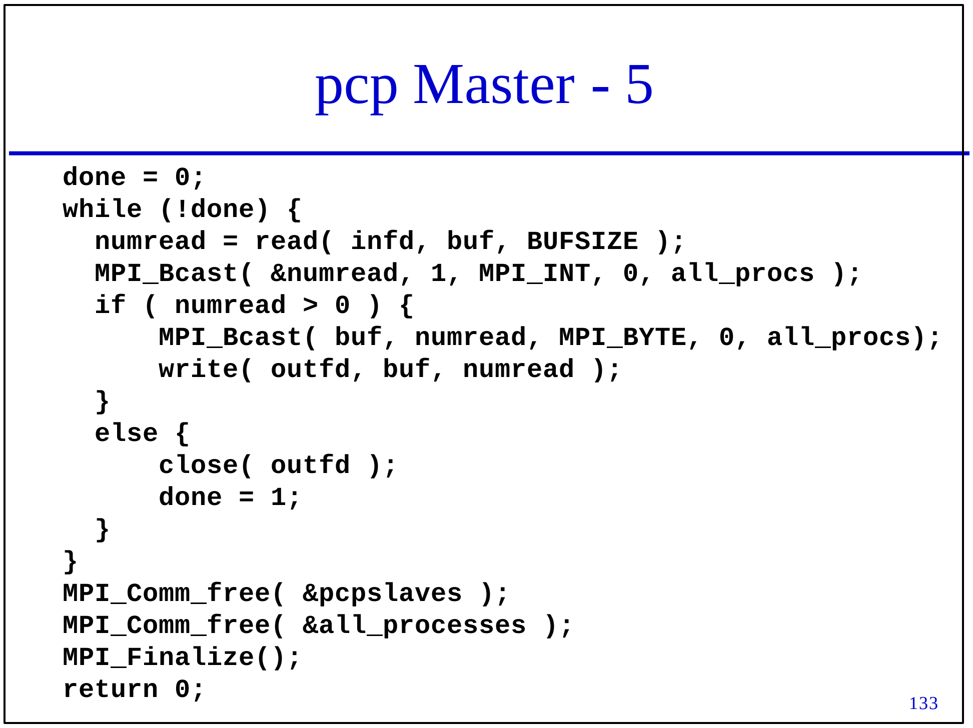
pcp master - 3

```
strcpy( outfilename, argv[3] );
if ( (infd = open( argv[2], O_RDONLY ) ) == -1 ) {
     fprintf( stderr, "%s doesn’t exist\n", argv[2] );
     sprintf( controlmsg, "exit" );
     MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0,
                all_procs );
     MPI_Finalize();
     return -1 ;
 }
 else {
     sprintf( controlmsg, "ready" );
     MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0,
                all_procs );
 }
```



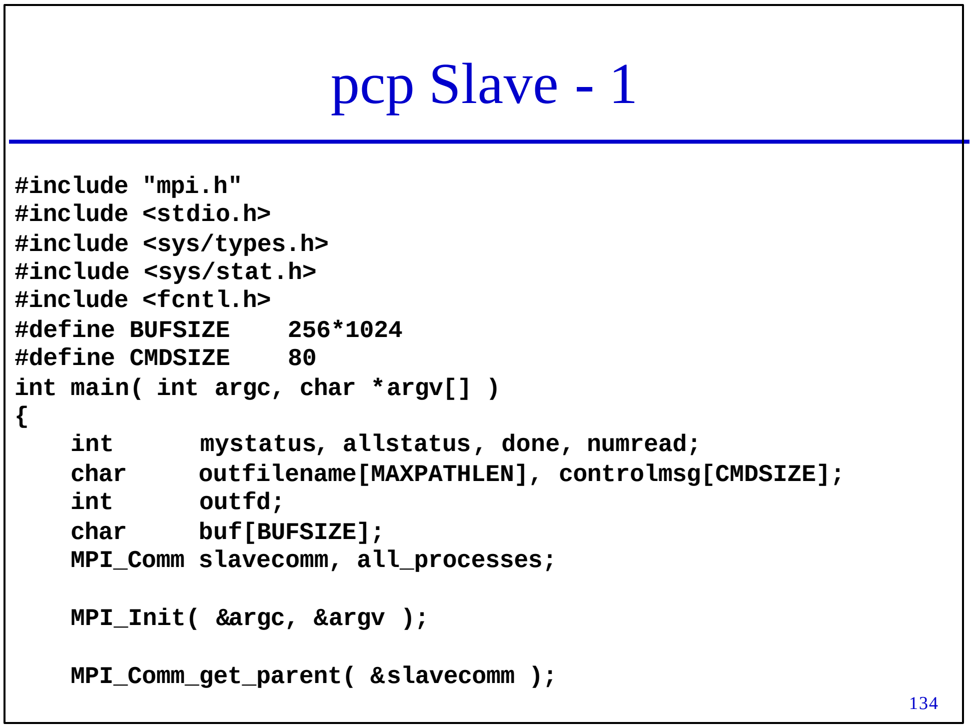
pcp Master - 4

```
MPI_Bcast( outfilename, MAXPATHLEN, MPI_CHAR, 0,
           all_procs );
if ( (outfd = open( outfilename,
      O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU ) ) == -1 )
    mystatus = -1;
else
    mystatus = 0;
MPI_Allreduce( &mystatus, &allstatus, 1, MPI_INT,
               MPI_MIN, all_procs );
if ( allstatus == -1 ) {
    fprintf( stderr, "Output file %s not opened\n",
             outfilename );
    MPI_Finalize();
    return 1 ;
}
```



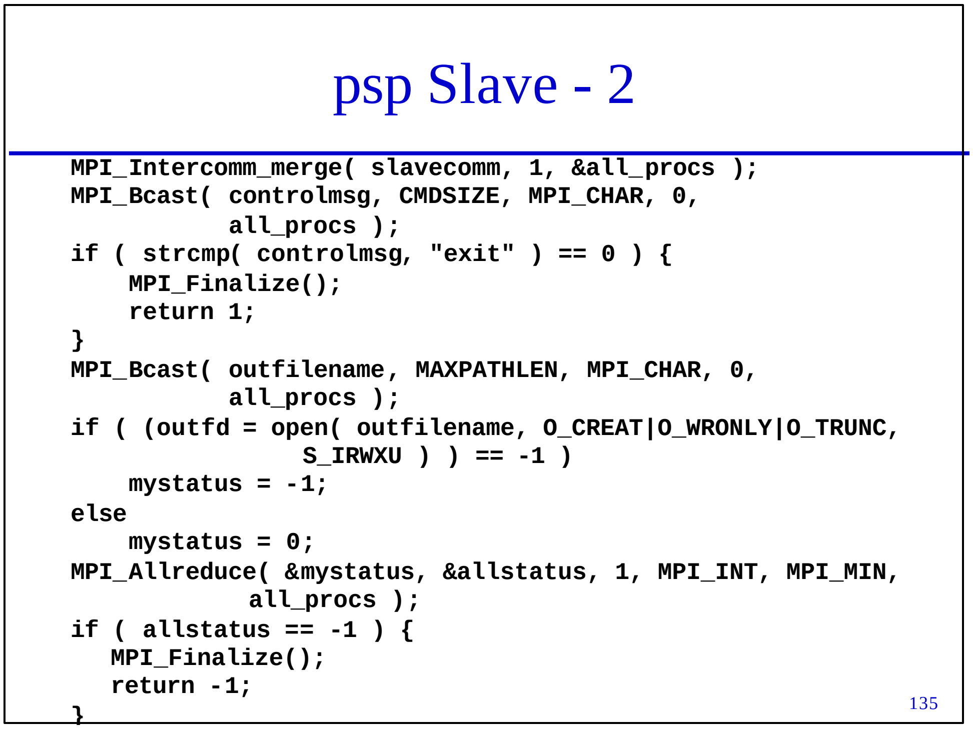
pcp Master - 5

```
    done = 0;
    while (!done) {
      numread = read( infd, buf, BUFSIZE );
      MPI_Bcast( &numread, 1, MPI_INT, 0, all_procs );
      if ( numread > 0 ) {
          MPI_Bcast( buf, numread, MPI_BYTE, 0, all_procs);
          write( outfd, buf, numread );
      }
      else {
          close( outfd );
          done = 1;
      }
    }
    MPI_Comm_free( &pcpslaves );
    MPI_Comm_free( &all_processes );
    MPI_Finalize();
    return 0;
```



pcp Slave - 1

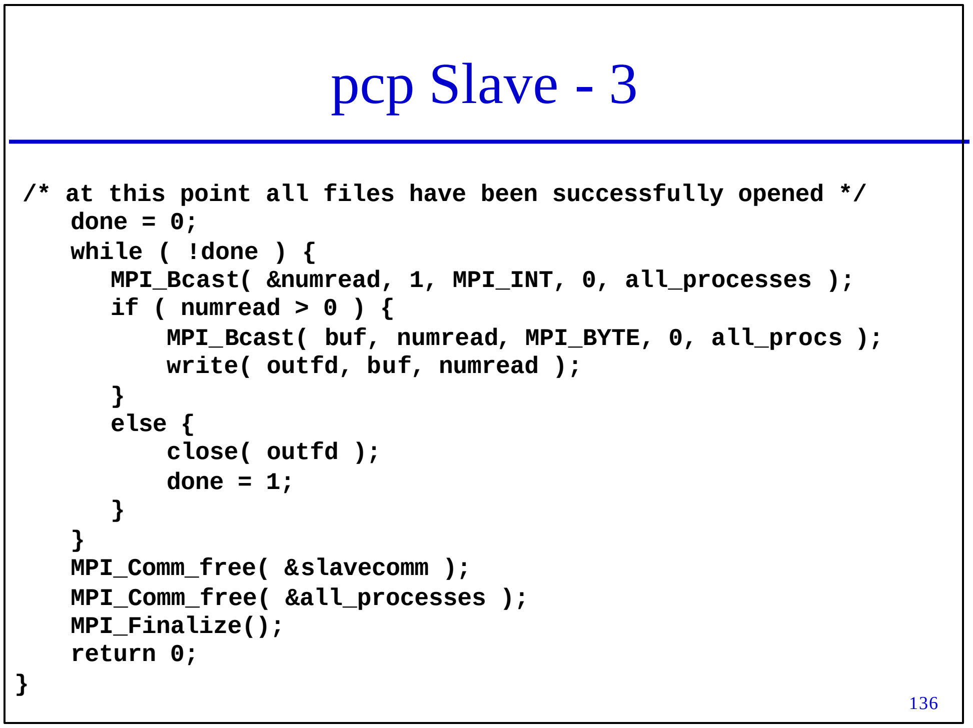
```
#include "mpi.h"
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BUFSIZE    256*1024
#define CMDSIZE    80
int main( int argc, char *argv[] )
{
    int      mystatus, allstatus, done, numread;
    char     outfilename[MAXPATHLEN], controlmsg[CMDSIZE];
    int      outfd;
    char     buf[BUFSIZE];
    MPI_Comm slavecomm, all_processes;

    MPI_Init( &argc, &argv );

    MPI_Comm_get_parent( &slavecomm );
```



psp Slave - 2

```
    MPI_Intercomm_merge( slavecomm, 1, &all_procs );
    MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0,
               all_procs );
    if ( strcmp( controlmsg, "exit" ) == 0 ) {
        MPI_Finalize();
        return 1;
    }
    MPI_Bcast( outfilename, MAXPATHLEN, MPI_CHAR, 0,
               all_procs );
    if ( (outfd = open( outfilename, O_CREAT|O_WRONLY|O_TRUNC,
                    S_IRWXU ) ) == -1 )
        mystatus = -1;
    else
        mystatus = 0;
    MPI_Allreduce( &mystatus, &allstatus, 1, MPI_INT, MPI_MIN,
                 all_procs );
    if ( allstatus == -1 ) {
       MPI_Finalize();
       return -1;
    }
```

135



pcp Slave - 3

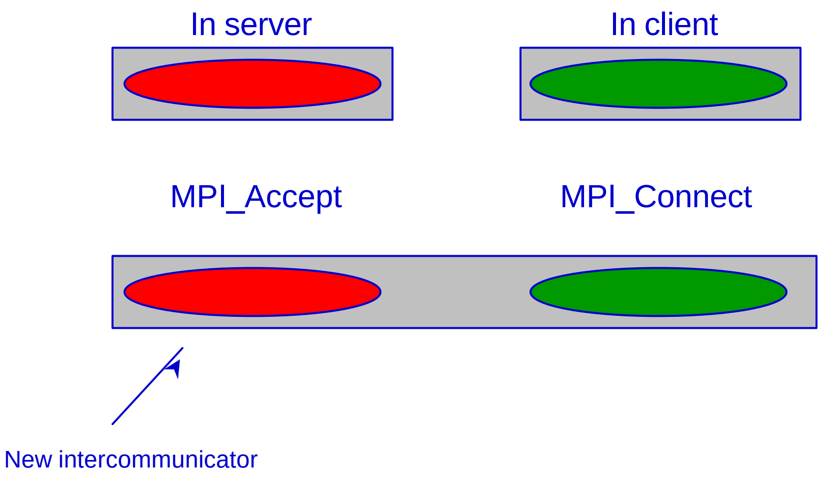
```
/* at this point all files have been successfully opened */
    done = 0;
    while ( !done ) {
       MPI_Bcast( &numread, 1, MPI_INT, 0, all_processes );
       if ( numread > 0 ) {
           MPI_Bcast( buf, numread, MPI_BYTE, 0, all_procs );
           write( outfd, buf, numread );
       }
       else {
           close( outfd );
           done = 1;
       }
    }
    MPI_Comm_free( &slavecomm );
    MPI_Comm_free( &all_processes );
    MPI_Finalize();
    return 0;
}
```

136

Establishing Connections

- Two sets of MPI processes may wish to establish connections, e.g.,
  - Two parts of an application started separately.
  - A visualization tool wishes to attach to an application.
  - A server wishes to accept connections from multiple clients. Both server and client may be parallel programs.
- Establishing connections is collective but asymmetric (“Client”/“Server”).
- Connection results in an intercommunicator.

Establishing Connections Between Parallel Programs



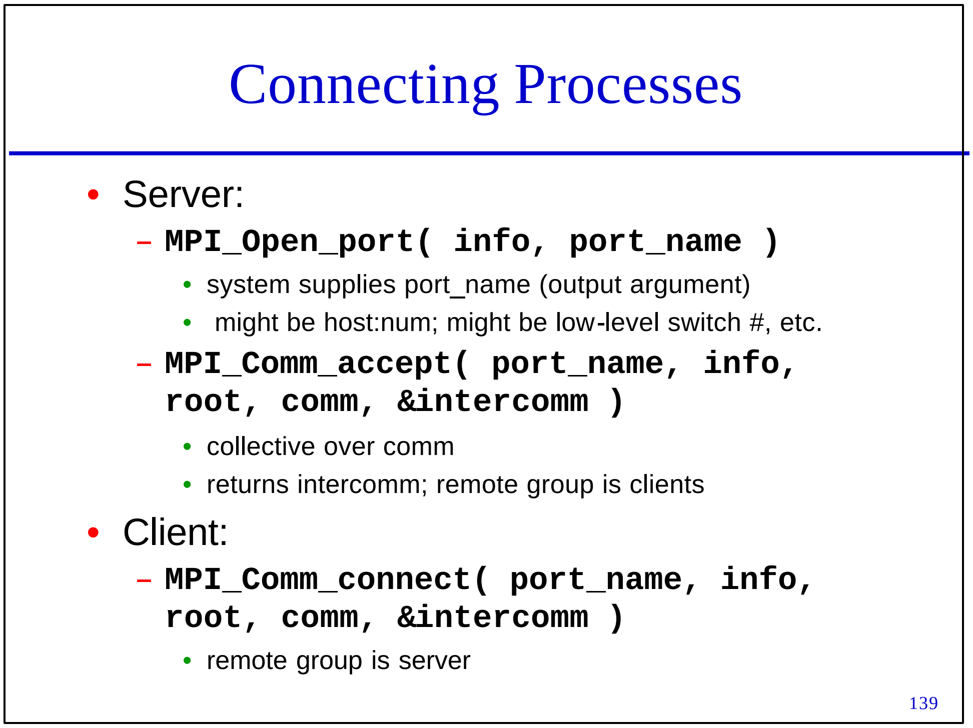
In server

In client

MPI_Accept

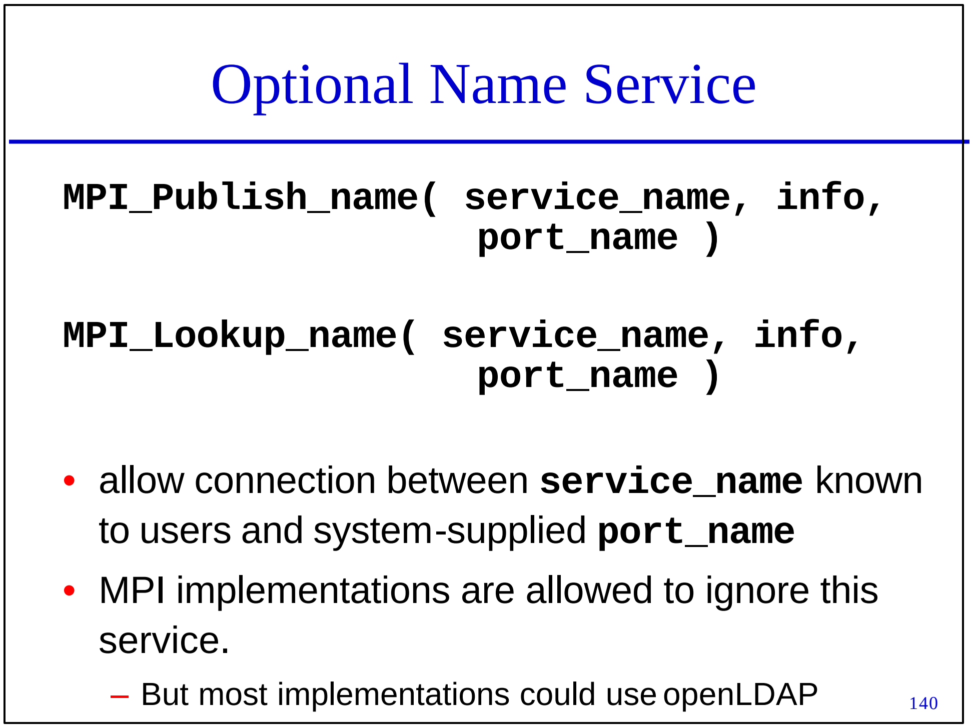
MPI_Connect

New intercommunicator



Connecting Processes

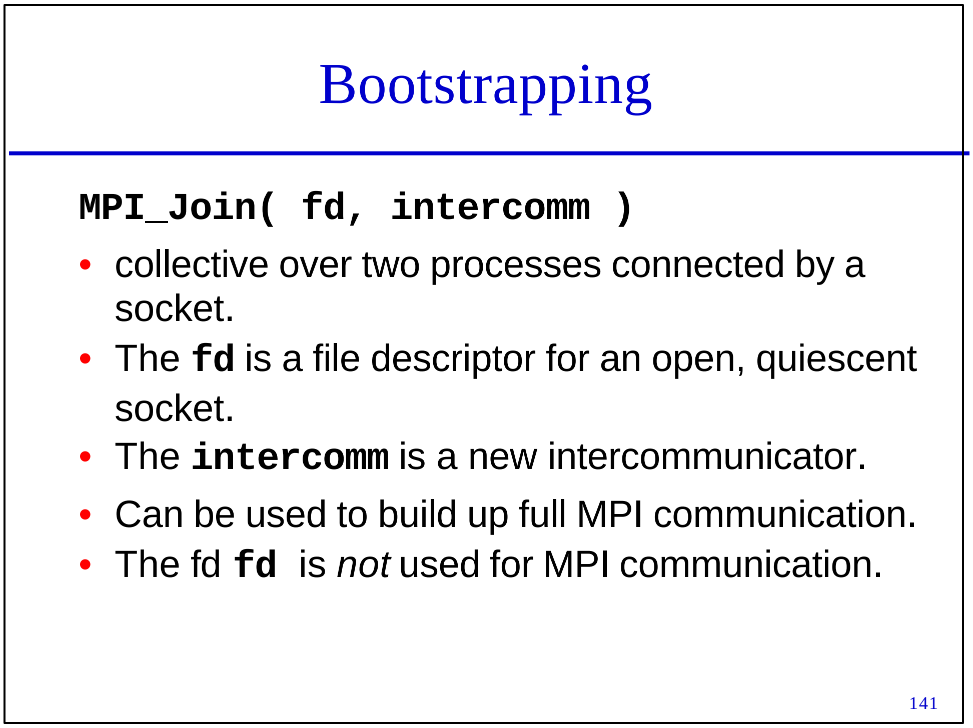
- Server:
  - `MPI_Open_port( info, port_name )`
    - system supplies port_name (output argument)
    - might be host:num; might be low-level switch #, etc.
  - `MPI_Comm_accept( port_name, info, root, comm, &intercomm )`
    - collective over comm
    - returns intercomm; remote group is clients
- Client:
  - `MPI_Comm_connect( port_name, info, root, comm, &intercomm )`
    - remote group is server



Optional Name Service

```
MPI_Publish_name( service_name, info,
                  port_name )

MPI_Lookup_name( service_name, info,
                 port_name )
```

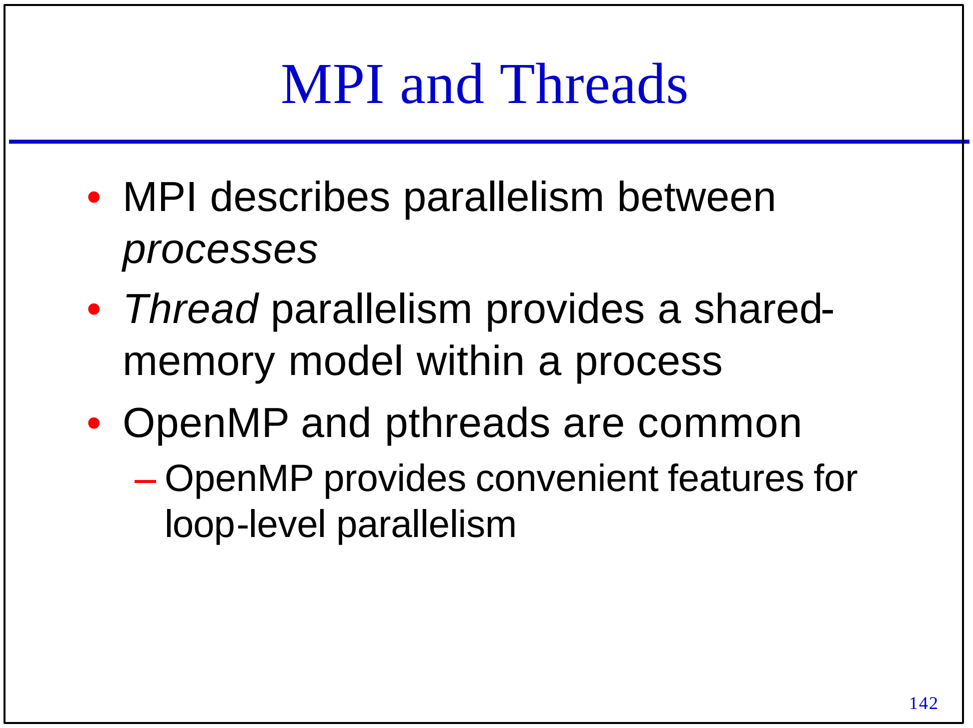
- allow connection between **`service_name`** known to users and system-supplied **`port_name`**
- MPI implementations are allowed to ignore this service.
  - But most implementations could use openLDAP



Bootstrapping

`MPI_Join( fd, intercomm )`

- collective over two processes connected by a socket.
- The **`fd`** is a file descriptor for an open, quiescent socket.
- The **`intercomm`** is a new intercommunicator.
- Can be used to build up full MPI communication.
- The fd **`fd`** is *not* used for MPI communication.



MPI and Threads

- MPI describes parallelism between *processes*
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and pthreads are common
  - OpenMP provides convenient features for loop-level parallelism

Threads and MPI in MPI-2

- MPI-2 specifies four levels of thread safety
  - MPI_THREAD_SINGLE : only one thread
  - MPI_THREAD_FUNNELED : only one thread that makes MPI calls
  - MPI_THREAD_SERIALIZED : only one thread at a time makes MPI calls
  - MPI_THREAD_MULTIPLE : any thread can make MPI calls at any time
- MPI_Init_thread(..., required, &provided) can be used instead of MPI_Init

MPI and OpenMP

- Loop-level parallelism
  - Requires only MPI_THREAD_FUNNELLED
- Task-parallelism
  - Requires MPI_THREAD_MULTIPLE

Using OpenMP with MPI

- The famous cpi program, using OpenMP for loop-level
- Worker/manager

cpi with Bcast/Reduce and OpenMP

```
<get n to use>
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h   = 1.0 / (double) n;
sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += f(x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
if (myid == 0) {
    printf("pi is %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
}
```

Task Parallelism with MPI and OpenMP

```
#pragma omp sections
{
#   pragma omp section
    {
    while (not_done) {
        MPI_Recv( … );
        if (status.MPI_TAG == ALL_DONE) break;
        <handle message>
        }
    }
#   pragma omp section
    {
    <compute thread>
    }
}
```

MPI Thread Levels and OpenMP

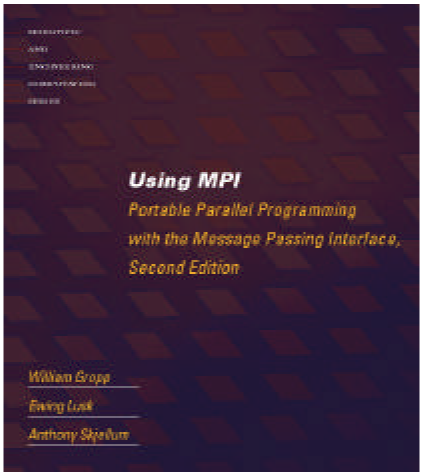
- MPI_THREAD_SINGLE
  - OpenMP not permitted
- MPI_THREAD_FUNNELED
  - All MPI calls in the “main” thread
  - MPI calls outside of any OpenMP sections
  - Most if not all MPI implementations support this level
- MPI_THREAD_SERIALIZED

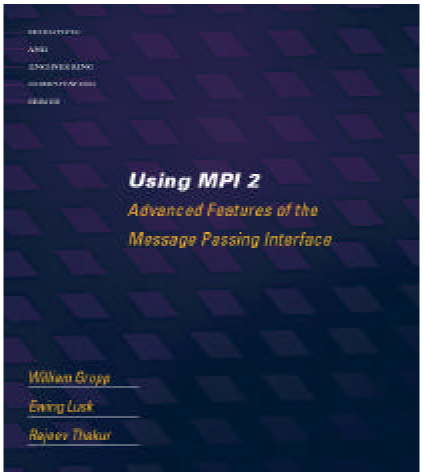
  -
    ```
    #pragma omp parallel
    …
    #pragma omp single
    {
      MPI calls allowed here as well
    }
    ```
- MPI_THREAD_MULTIPLE
  - Any MPI call anywhere
  - But be careful of races
  - Some MPI implementations support this level

Conclusions

- MPI is a proven, effective, portable parallel programming model
- MPI has succeeded because
  - features are orthogonal (complexity is the product of the number of *features*, not routines)
  - programmer can control memory motion (critical in high-performance computing)
  - complex programs are no harder than easy ones
  - open process for defining MPI led to a solid design
- MPI I/O is widely available and efficient
- Dynamic process management becoming available

Tutorial Material on MPI, MPI-2


Using MPI
Portable Parallel Programming
with the Message Passing Interface,
Second Edition
William Gropp
Ewing Lusk
Anthony Skjellum



http://www.mcs.anl.gov/mpi/{usingmpi,usingmpi2}