Advanced MPI

John Zollweg

Introduction to Parallel Computing
May 28, 2009

based on material developed by Bill Barth, TACC

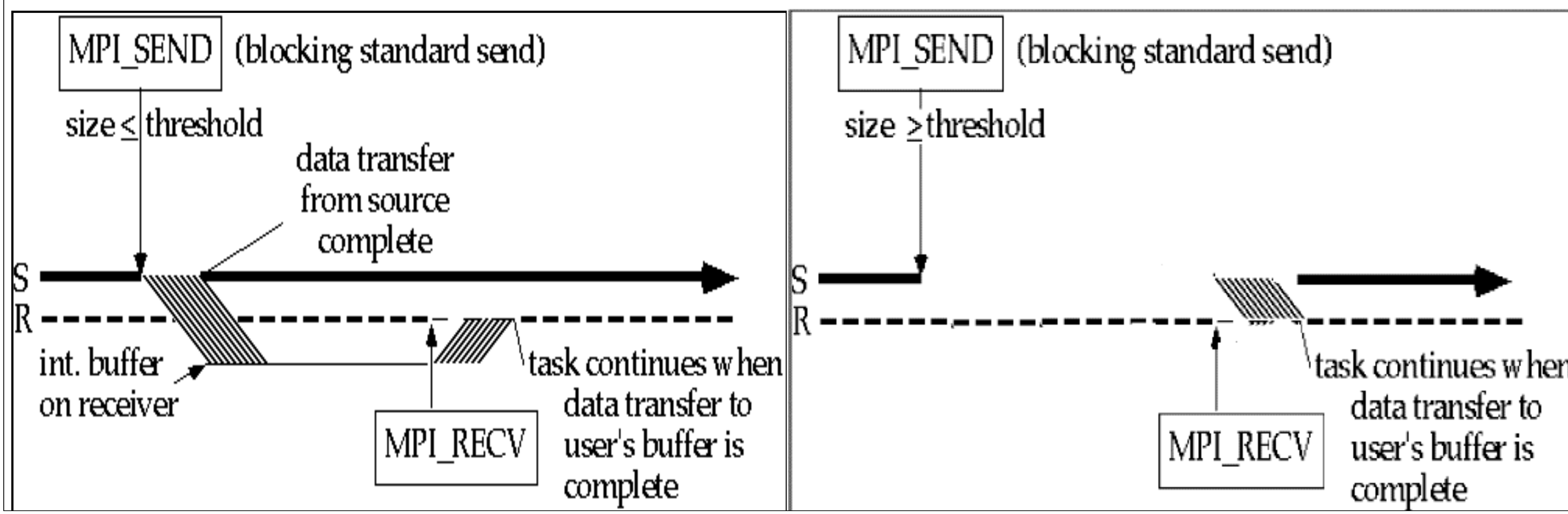
Introduction & Outline

- Point to Point blocking/non-blocking communication
- Collective communication with non-contiguous data
- Groups and communication management
- Derived Datatypes
- Persistent communication
- Parallel I/O
- Status of MPI-2

Advanced point-to-point communication

Point to Point Comm. I

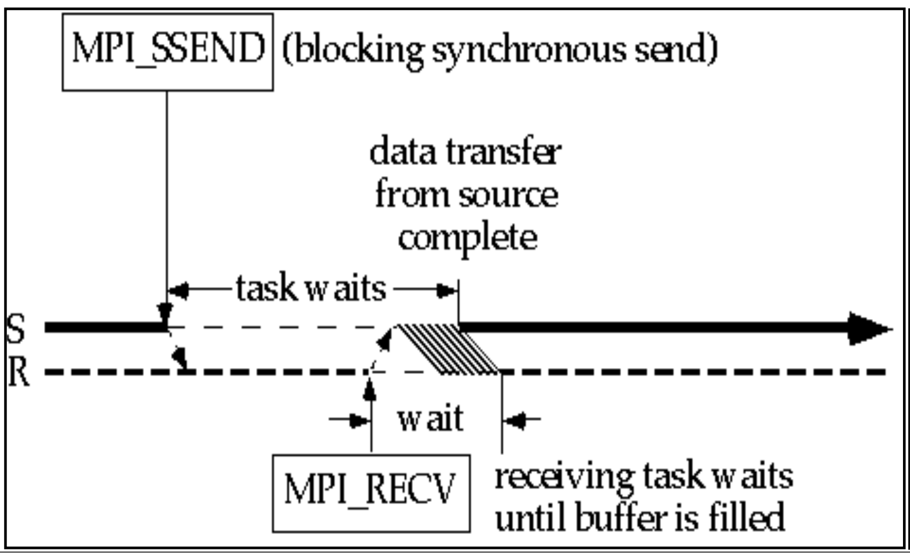
- Blocking send/receive
- MPI_Send, does not return until buffer is safe to reuse: either when buffered, or when actually received. (implementation / runtime dependent)
- Rule of thumb: send completes only if receive is posted/executed

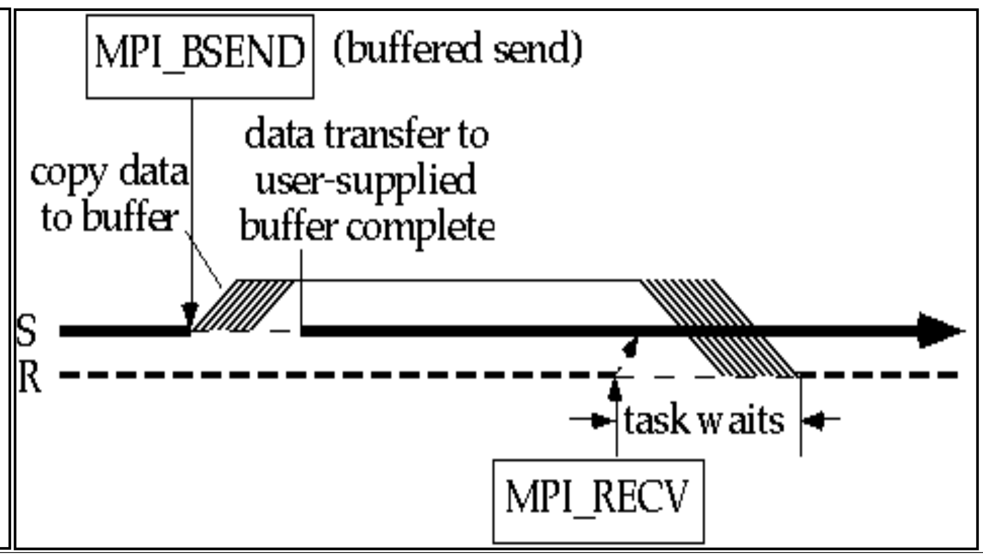


Cornell University
Center for Advanced Computing

Point to Point Comm. II

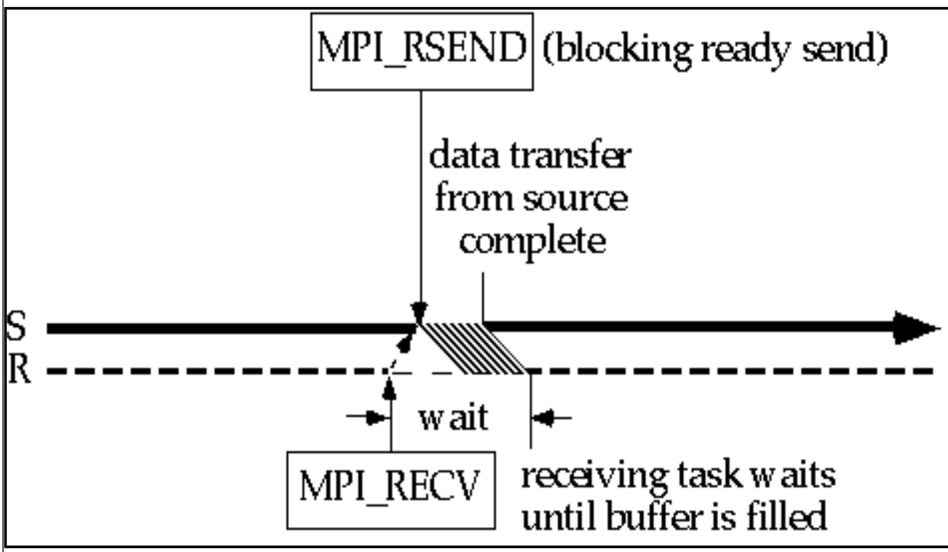
- Synchronous Mode
  - MPI_Ssend, which does not return until matching receive has been posted (non-local operation).
- Buffered Mode
  - MPI_Bsend, which completes as soon as the message buffer is copied into user-provided buffer (one buffer per process)

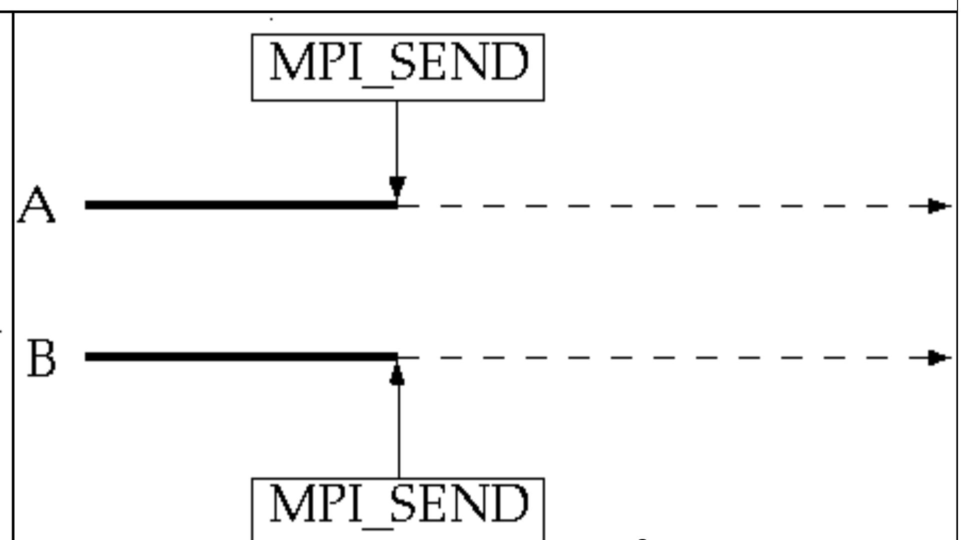




Point to Point Comm. III

- Ready Mode
  - MPI_Rsend, which returns immediately assuming that a matching receive has been posted, else erroneous.
- Deadlock occurs when all tasks are waiting for events that haven't been initiated yet. It is most common with blocking communication.



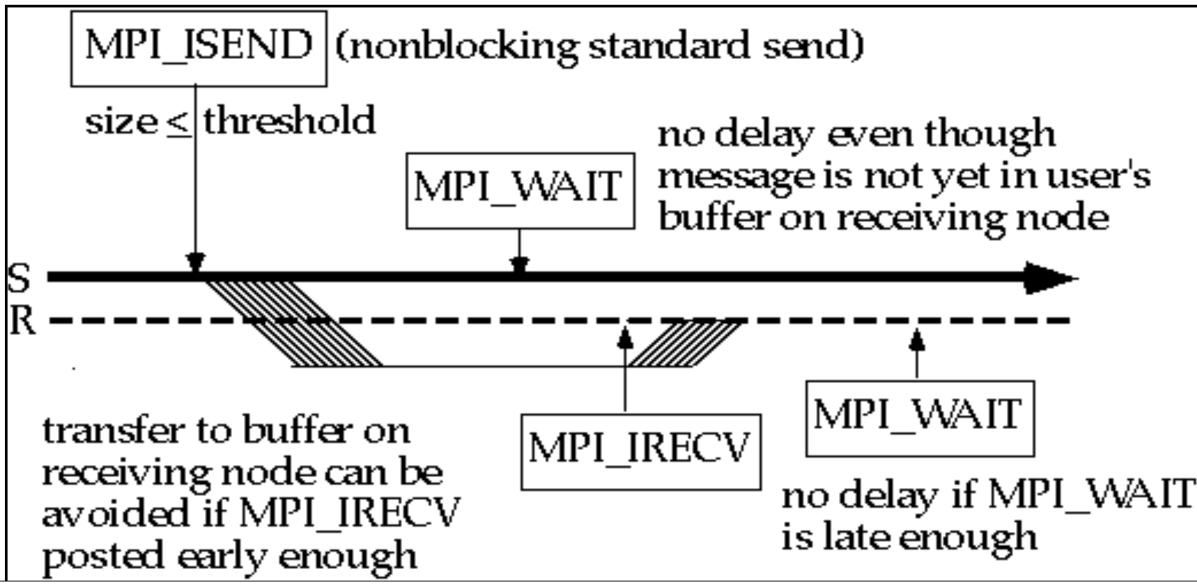


Point to Point Comm. III

- Ready Mode has least total overhead. However the assumption is that receive is already posted. Solution: post receive, synchronise (zero byte send), then post send.
- Synchronous mode is portable and “safe”. It does not depend on order (ready) or buffer space (buffered). However it incurs substantial overhead.
- Buffered mode decouples sender from receiver. No sync. overhead on sending task and order of execution does not matter (ready). User can control size of message buffers and total amount of space. However additional overhead may be incurred by copy to buffer.
- Standard Mode is implementation dependent. Small messages are generally *buffered* (avoiding sync. overhead) and large messages are usually sent synchronously (avoiding the required buffer space)

Point to Point Comm IV: non-blocking

- Nonblocking communication: calls return, system handles buffering
- MPI_Isend, completes immediately but user must check status before using the buffer for same (tag/receiver) send again; buffer can be reused for different tag/receiver.
- MPI_Irecv, gives a user buffer to the system; requires checking whether data has arrived.



Non-blocking Example

- Blocking operations can lead to deadlock
- Actual user code:
- Problem: all sends are waiting for corresponding receive: nothing happens
- Why did the user code work on one machine, but not in general?

```
!   SEND WELL DATA
    LM=6*NES+2
    DO I=1,NUMPRC
      NT=I-1
      IF (NT.NE.MYPRC) THEN
        print *,myprc,'send',msgtag,'to',nt
        CALL MPI_SEND(NWS,LM,MPI_INTEGER,NT,
 & MSGTAG,MPI_COMM_WORLD,IERR)
      ENDIF
    END DO

!   RECEIVE WELL DATA
    LM=6*100+2
    DO I=2,NUMPRC
      CALL MPI_RECV(NWS,LM,MPI_INTEGER,
 &    MPI_ANY_SOURCE,MSGTAG,MPI_COMM_WORLD,IERR)
! do something with data
    END DO
```

Solution using non-blocking send

```
      real*8 sendbuf(d,np-1), recvbuf(d)
      MPI_Request sendreq(np)
      do p=1,nproc-1
        pp = 0
        if (p.ge.mytid) pp = pp+1
          call mpi_isend(sendbuf(1,p),d,MPI_DOUBLE,pp,msgtag,
     &         comm,sendreq(p),ierr)
      end do
      do p=1,nproc-1
        call mpi_recv(recvbuf(1),d,MPI_DOUBLE,MPI_ANY_SOURCE,
     &         msgtag,comm,ierr)
c do something with incoming data
      end do
```

Note: This requires multiple send buffers, should "wait" later...

Solution using non-blocking send/recv

```
      real*8 sendbuf(d,np-1), recvbuf(d,np-1)
      MPI_Request sendreq(np-1), recvreq(np-1)
      integer sendstat(MPI_STATUS_SIZE),recvstat(MPI_STATUS_SIZE)
      do p=1,nproc-1
c       mpi_isend as before
      end do
      do p=1,nproc-1
        pp = p
        if (pp.ge.mytid) pp = pp+1
        call mpi_irecv(recvbuf(1,p),d,MPI_DOUBLE,pp,
     &           msgtag,comm,recvreq(p),ierr)
      end do
      call mpi_waitall(nproc-1,sendreq,sendstat,ierr)
      call mpi_waitall(nproc-1,recvreq,recvstat,ierr)
```

Note: multiple send and receive buffers; Explicit wait calls to make sure commun-ications are finished.

Non-blocking example

- Non-blocking operations allow overlap of computation and communication.
- Application: distributed matrix-vector product
- Also non-blocking R/B/Ssend

```
MPI_Irecv( <declare receive buffer> )
MPI_Isend( <send local data> )
…. Do local operations ….
MPI_Waitall( <make sure all receives finish> )
…. Operate on received data ….
```

Point to Point Comm. V

- “Wildcard communication”: source or details unknown
  can use **`MPI_ANY_SOURCE or MPI_ANY_TAG`**

`MPI_IPROBE(source, tag, comm, flag, status)`

(non-blocking; **`MPI_Probe`** is blocking)

`MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)`

(allows processing of data as it comes in)

`MPI_Testany / Testall :` non-blocking

Point to Point Comm. VI

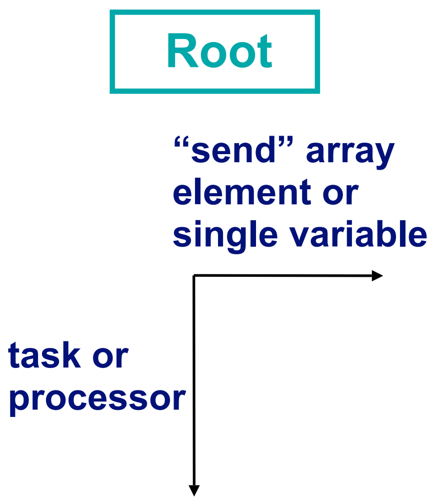
- MPI_Sendrecv : both in on call, source and destination can be the same
- Also MPI_Sendrecv_Replace; needs additional buffering
- Example 1: exchanging data with one other node; target and source the same
- Example 2: chain of processors
  - Operate on data
  - Send result to next processor, and receive next input from previous processor in line

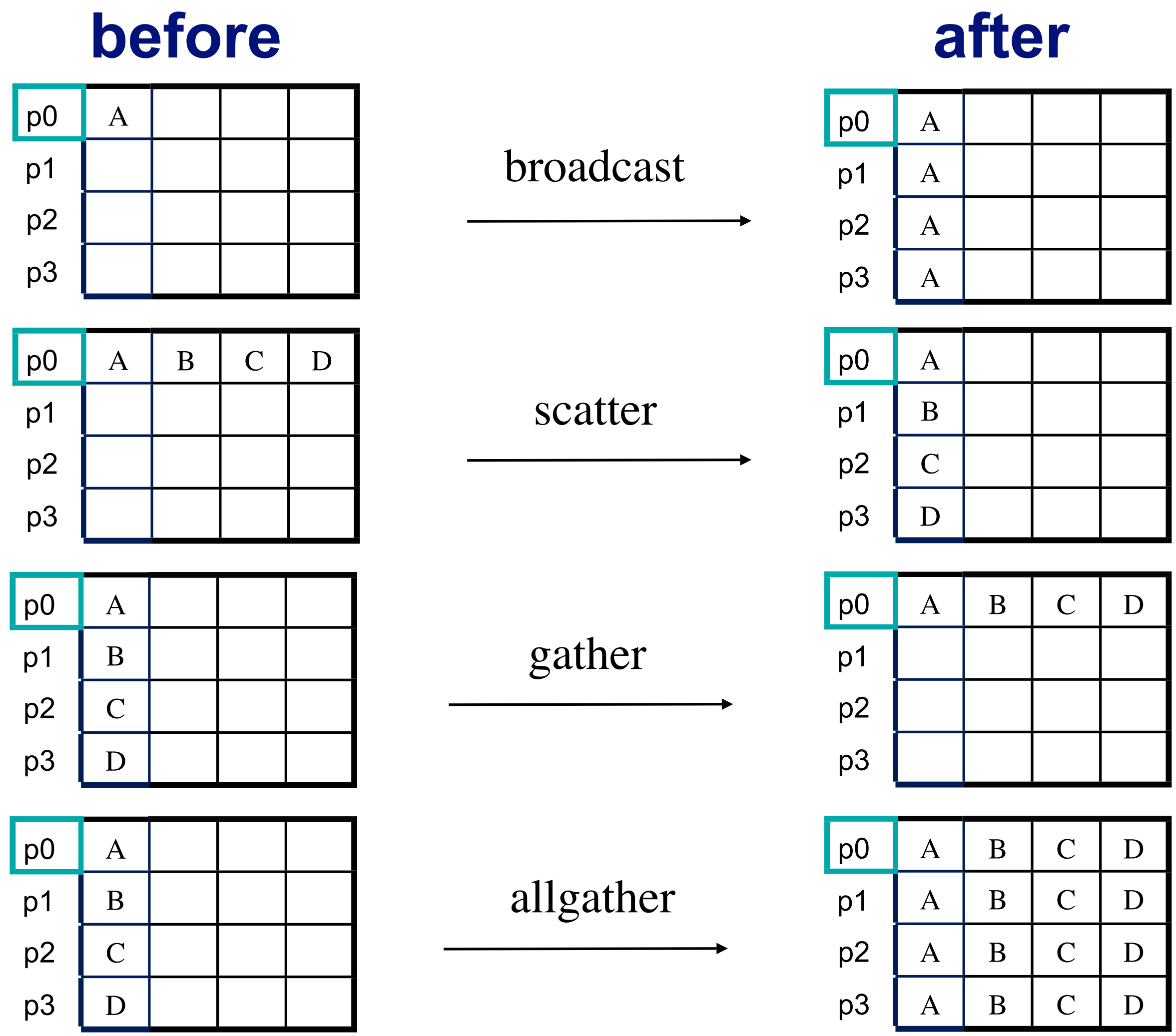
```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag,
             recvbuf, recvcount, recvtype, source, recvtag, comm, status)
```

Collective communications

Advanced Collective Comm. I

Scatter and Gather



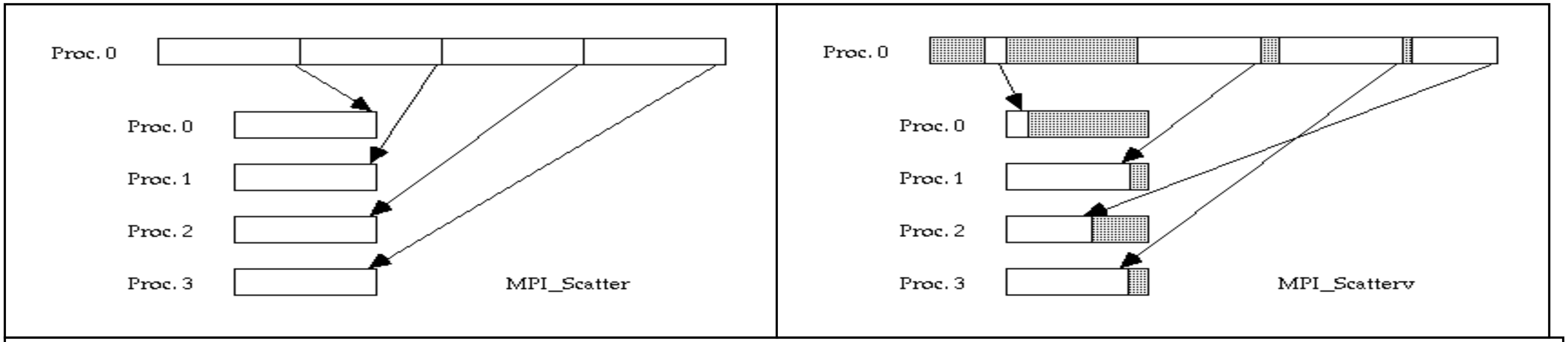


Advanced Collective Comm. II

- MPI_{Scatter,Gather,Allgather}*v*
- What does *v* stand for?
  - varying size, relative location of messages
- Advantages
  - more flexibility
  - less need to copy data into temp. buffers
  - more compact
- Disadvantage
  - Lot harder to program

Advanced Collective Comm. II+

- Scatter vs Scatterv




```
CALL mpi_scatterv  ( SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE,
RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERR )

SENDCOUNTS(I) is the number of items of type SENDTYPE to send
from process ROOT to process I.  Defined on ROOT.

DISPLS(I) is the displacement from SENDBUF to the
   beginning of the I-th message, in units of SENDTYPE.
   Defined on ROOT.
```

Allgatherv Example

```
MPI_Comm_size(comm,&ntids);
sizes = (int*)malloc(ntids*sizeof(int));
MPI_Allgather(&n,1,MPI_INT,sizes,1,MPI_INT,comm);
offsets = (int*)malloc(ntids*sizeof(int));
s=0;
for (i=0; i<ntids; i++)
  {offsets[i]=s; s+=sizes[i];}
N = s;
result_array = (int*)malloc(N*sizeof(int));
MPI_Allgatherv
   ((void*)local_array,n,MPI_INT,(void*)result_array,
    sizes,offsets,MPI_INT,comm);
free(sizes); free(offsets);
```

Derived Datatypes

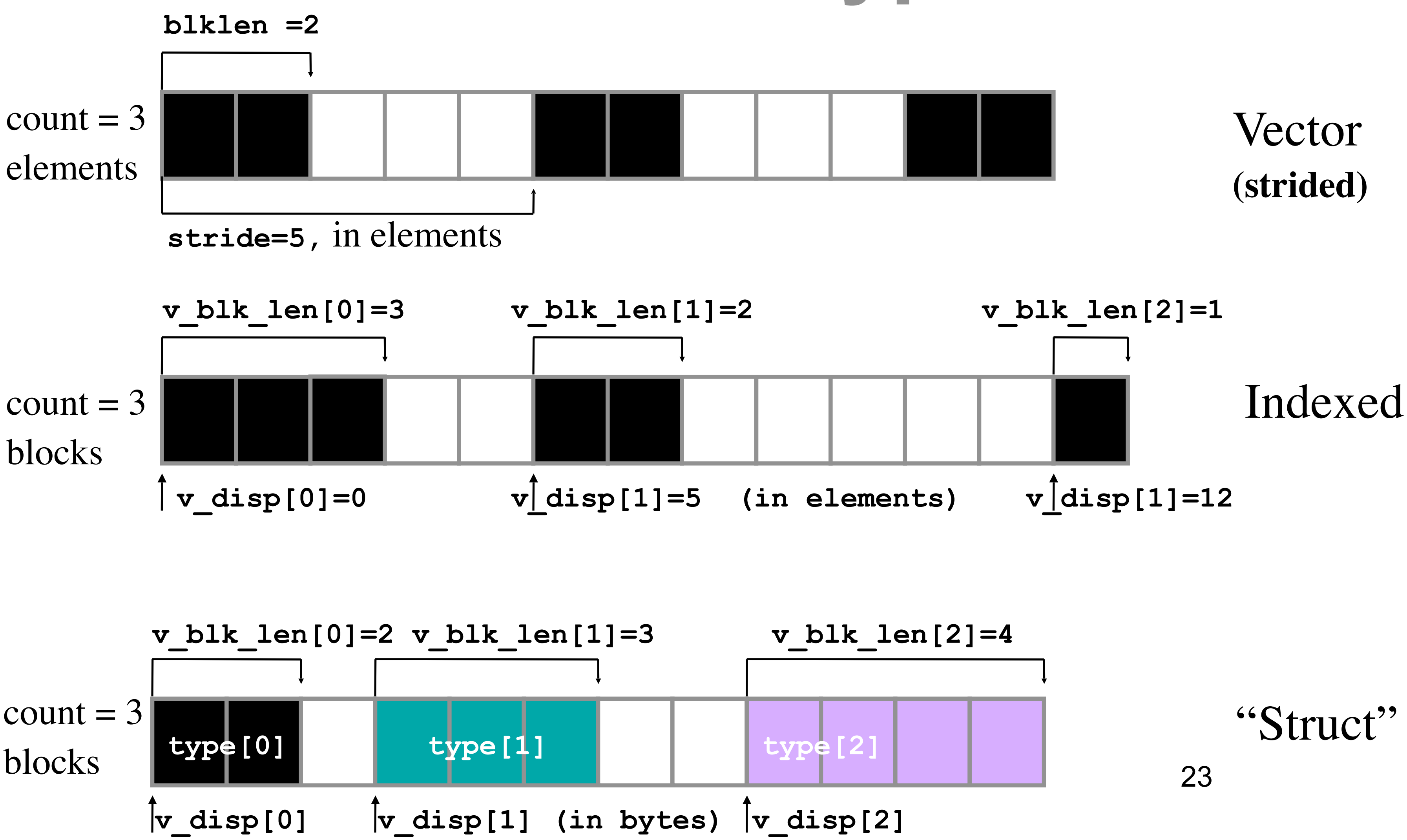
Derived Datatypes I

- MPI basic data-types are predefined for contiguous data of single type
- What if application has data of mixed types, or non-contiguous data?
  - existing solutions of multiple calls or copying into buffer and packing etc. are slow, clumsy and waste memory
  - better solution is creating/deriving datatypes for these special needs from existing datatypes
- Derived datatypes can be created recursively at runtime
- Automatic packing and unpacking

Derived Datatypes II

- Elementary: Language-defined types
- Contiguous: Vector with stride of one
- Vector: Separated by constant “stride”
- 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.)

Derived Datatypes III



Derived Datatypes IV

- MPI_TYPE_VECTOR function allows creating non-contiguous vectors with constant stride

mpi_type_vector(count, blocklen, stride, oldtype, vtype, ierr)
mpi_type_commit(vtype, ierr)

ncols = 4
nrows = 5

A

1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19
5	10	15	20

call MPI_Type_vector(ncols, 1, nrows, MPI_DOUBLE_PRECISION, vtype, ierr)

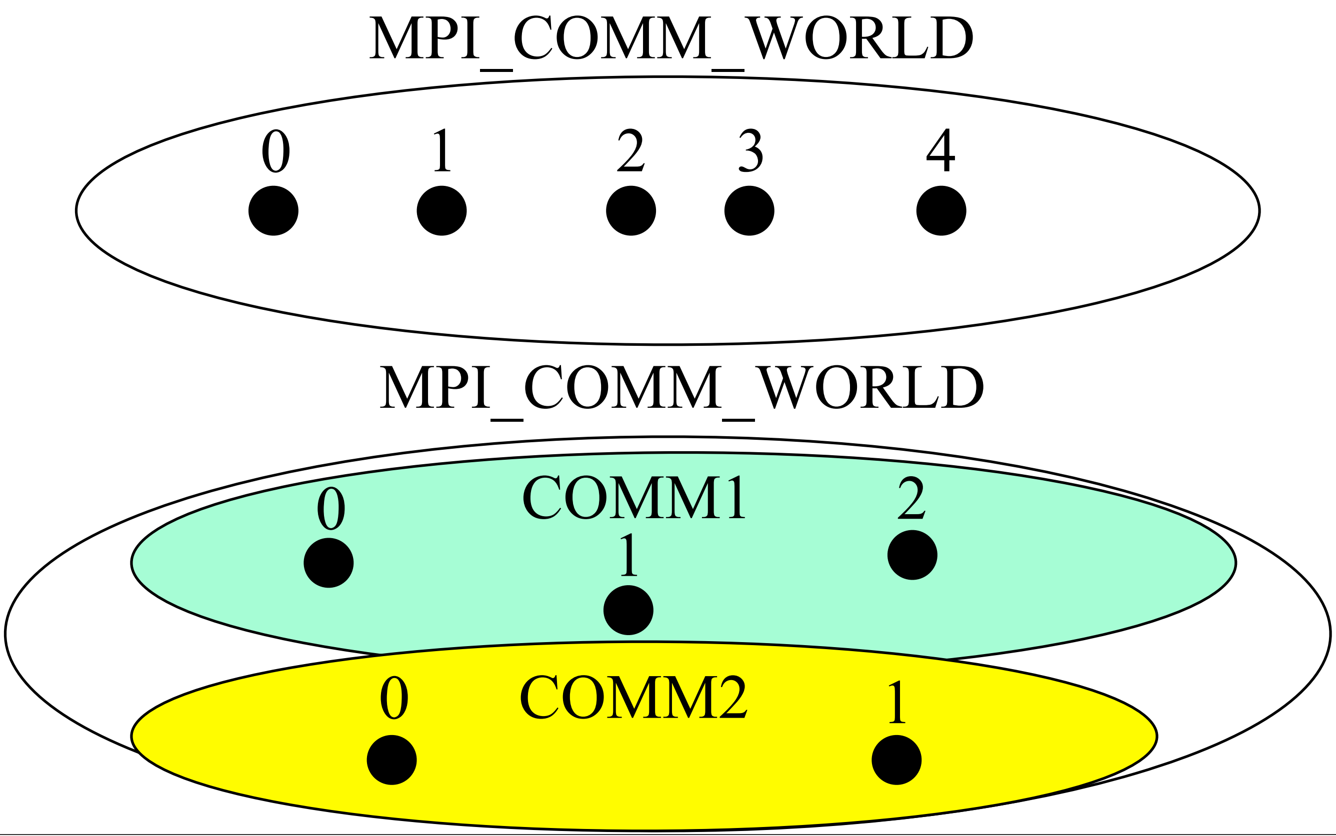
call MPI_Type_commit(vtype, ierr)

call MPI_Send(A(nrows,1) , 1 , vtype …)

Communicators and Groups

Communicators and Groups I

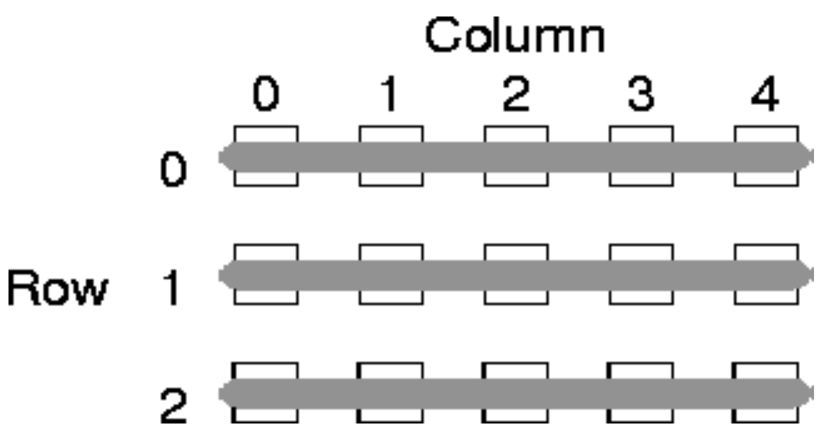
- All MPI communication is relative to a *communicator* which contains a *context* and a *group*. The group is just a set of processes.



Communicators and Groups II

- To subdivide communicators into multiple non-overlapping communicators – Approach I

```
                           :
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
myrow = (int)(rank/ncol);
                           :
```



MPI_Comm_split

- Argument #1: communicator to split
- Argument #2: key, all processes with the same key go in the same communicator
- Argument #3 (optional): value to determine ordering in the result communicator
- Argument #4: result communicator

```
                        :
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
myrow = (int)(rank/ncol);
MPI_Comm_split(MPI_COMM_WORLD, myrow,rank,row_comm);
                        :
```

Communicators and Groups III

- Same example – using groups
- MPI_Comm_group: extract group from communicator
- Create new groups
- MPI_Comm_create: communicator from group

Communicator groups example

```
                        :
MPI_Group base_grp, grp;  MPI_Comm row_comm, temp_comm;
int row_list[NCOL], irow, myrank_in_world;

MPI_Comm_group(MPI_COMM_WORLD,&base_grp);  // get base group


MPI_Comm_rank(MPI_COMM_WORLD,&myrank_in_world);
irow = (myrank_in_world/NCOL);
for (i=0; i <NCOL; i++)  row_list[i] = i;


for (i=0; i <NROW; i++){
   MPI_Group_incl(base_grp,NCOL,row_list,&grp);
   MPI_Comm_create(MPI_COMM_WORLD,grp,&temp_comm);
   if (irow == i) *row_comm=temp_comm;
   for (j=0;j<NCOL;j++) row_list[j] += NCOL;
}
                     :
```

Communicators and Groups IV

- When using *MPI_Comm_split*, one communicator is split into multiple non-overlapping communicators. Approach I is more compact and is most suitable for regular decompositions.
- Approach II is most generally applicable. Other group commands: union, difference, intersection, range in/exclude

Persistent communication

Persistent Communication I

- Saves arguments of a communication call and reduces the overhead for subsequent calls
- The INIT call takes the original argument list of a send or receive call and creates a corresponding communication request (e.g., MPI_SEND_INIT, MPI_RECV_INIT)
- The START call uses the communication request to start the corresponding operation (e.g. MPI_START, MPI_STARTALL)
- The REQUEST_FREE call frees the persistent communication request(MPI_REQUEST_FREE)

Persistent Communication II

- A typical situation where *persistence* might be used.

```
                    :
MPI_Recv_init(buf1, count,type,src,tag,comm,&req[0]);
MPI_Send_init(buf2, count,type,src,tag,comm,&req[1]);

for (i=1; i < BIGNUM; i++)
{
      MPI_Start(&req[0]);
      MPI_Start(&req[1]);
      MPI_Waitall(2,req,status);
      do_work(buf1, buf2);
}

MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
                    :
```

Persistent Communication III

- Performance benefits (IBM SP2) from using *Persistence*

Improvement in Wallclock Time
Persistent vs. Conventional Communication

msize (bytes)	mode	improvement	mode	improvement
8	async	19 %	sync	15 %
4096	async	11 %	sync	4.7 %
8192	async	5.9 %	sync	2.9 %
800,000	-	-	sync	0 %
8,000,000	-	-	sync	0 %

Parallel I/O

What is Parallel I/O?

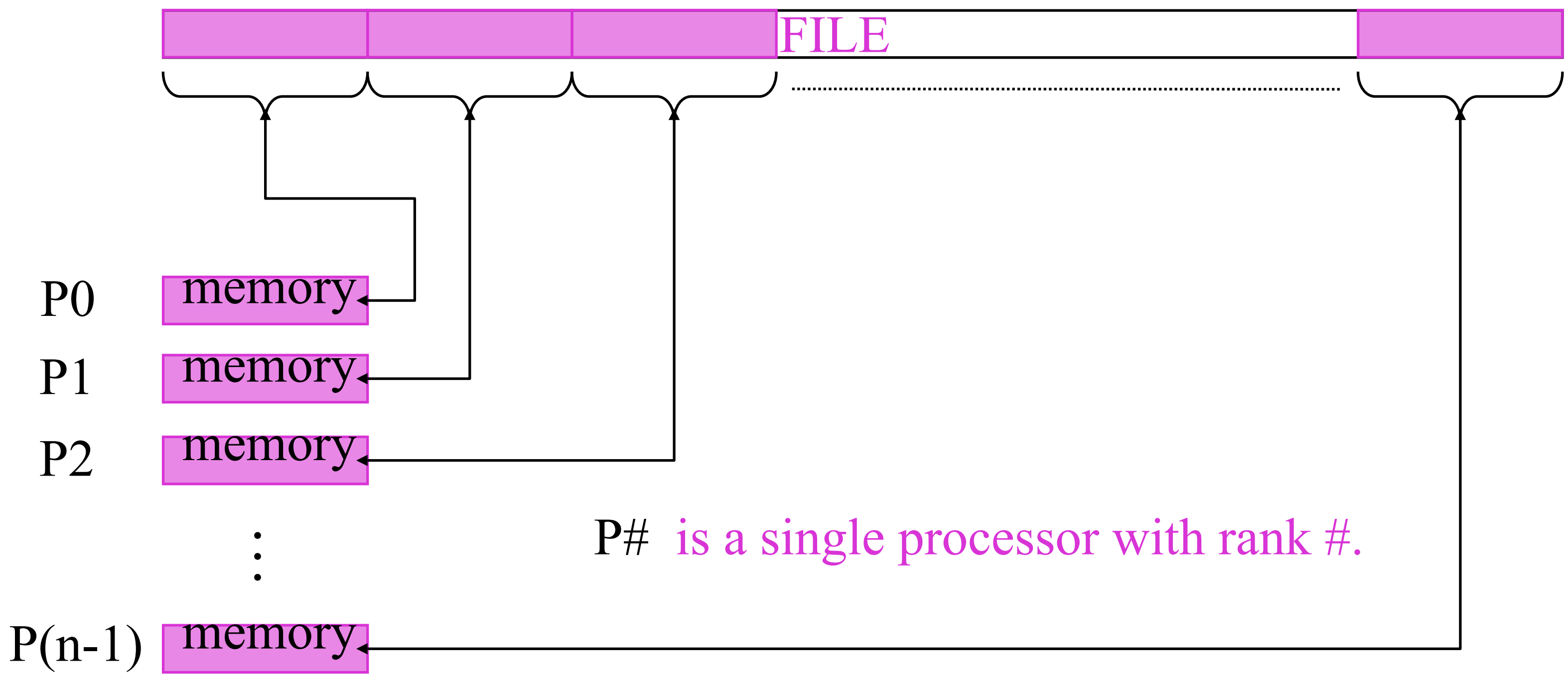
- In HPC parallel I/O, multiple MPI tasks can
  - simultaneously read or write to
  - a single file
  - in a parallel file system,
  - through the MPI-IO interface. A parallel file system appears as a normal Unix file system and (usually) employs multiple I/O servers for sustaining high I/O throughput.
- Alternatives to parallel MPI-IO:
  - Task 0 accesses file. Task 0 gathers/scatters data.
  - Each process opens a separate file and writes to it

Why Parallel I/O?

- I/O missing from MPI-1 standard, defined independently, then subsumed into MPI-2
- HPC Parallel I/O requires some work, but
  - Provides high throughput
  - Single (unified) file for vis. and pre/post processing
- Alternative I/O is simple to code, but has
  - Poor convenience (single task access to 1 file) or
  - Requires file management (each task uses local disk)
- MPI-IO has mechanisms to
  - perform synchronization and data movement syntax.
  - define noncontiguous data layout in file (*MPI datatypes*)

Simple MPI-IO

Each MPI task reads/writes a single block



Reading, Using Individual File Pointers

C Code

```
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);
```

Reading, Using Explicit Offsets

F90 Code

```
include 'mpif.h'
integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset

nints  = FILESIZE/(nprocs*INTSIZE)
offset = rank * nints * INTSIZE

call MPI_FILE_OPEN(   MPI_COMM_WORLD, '/pfs/datafile', &
                      MPI_MODE_RDONLY,                 &
                      MPI_INFO_NULL, fh, ierr)

call MPI_FILE_READ_AT(fh, offset, buf, nints,
                      MPI_INTEGER, status, ierr)

call MPI_FILE_CLOSE(fh, ierr)
```

Writing (with pointers or offsets)

- Use **`MPI_File_write`** or **`MPI_File_write_at`**
- **`MPI_File_open`** flags:
  - **`MPI_MODE_WRONLY`** (write only)
  - **`MPI_MODE_RDWR`** (read and write)
  - **`MPI_MODE_CREATE`** (create file if it doesn't exist)
  - Use bitwise-or '|' in C, or addition '+" in Fortran to combine multiple flags.

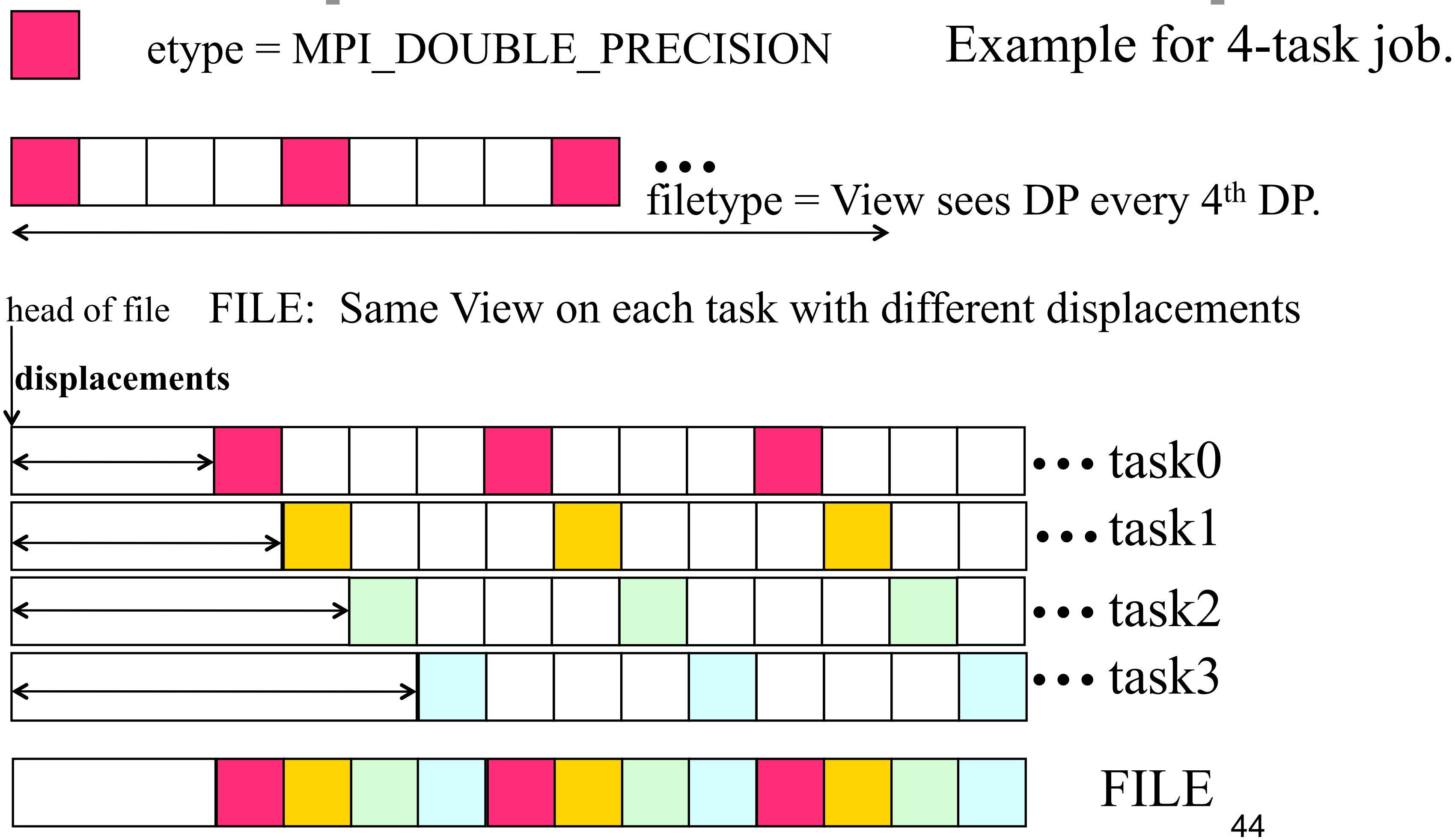
Shared Pointers

- One implicitly maintained pointer per collective file open
- **`MPI_File_read_shared`**
- **`MPI_File_write_shared`**
- **`MPI_File_seek_shared`**

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.

A Simple File View Example

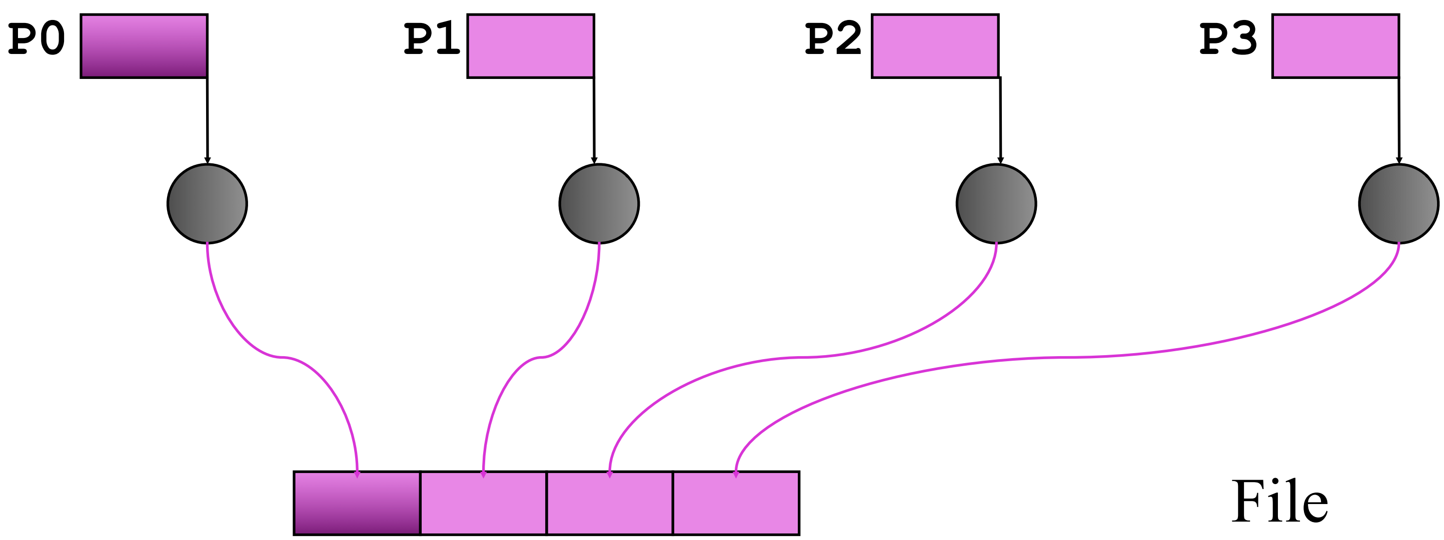


File Views

- 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 layout of etypes on disk.

Using File Views

- 1 block from each task, written in task order.



File

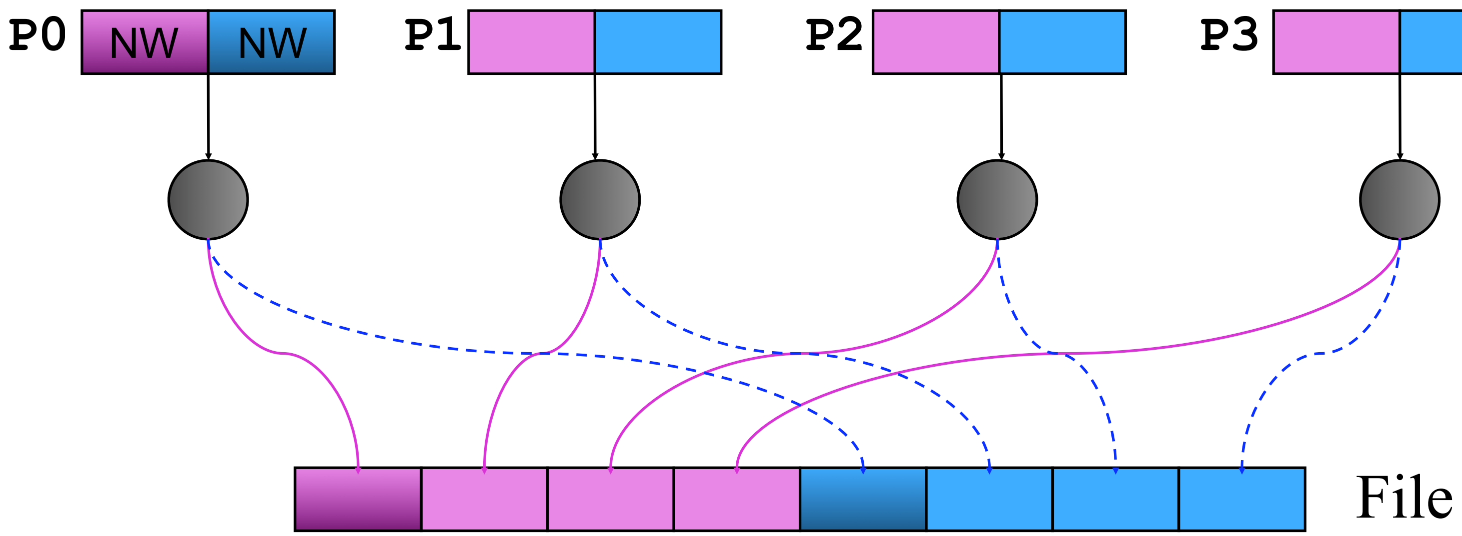
- **`MPI_File_set_view`** assigns regions of the file to separate processes

File View Code

```
#define N 100
MPI_Datatype arraytype;
MPI_Offset disp;

MPI_Type_contiguous(N, MPI_INT, &arraytype);
MPI_Type_commit(&arraytype);

disp = rank*sizeof(int)*N; 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, MPI_INT, arraytype, "native",
                  MPI_INFO_NULL);
MPI_File_write(   fh, buf, N, MPI_INT, MPI_STATUS_IGNORE);
```

Using File Views

- 2 blocks from each task, round-robin to file.



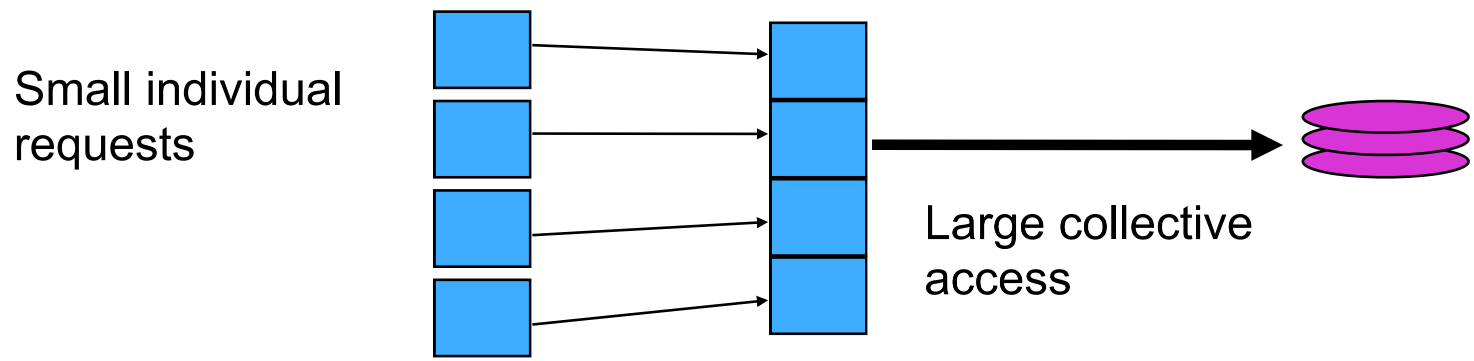
- **`MPI_File_set_view`** assigns regions of the file to separate processes

File View Code

```
int buf[NW*2];
MPI_File_open(MPI_COMM_WORLD, "/data2", MPI_MODE_CREATE |
                MPI_MODE_RDWR,MPI_INFO_NULL,&fh);

/* this processor can see only 2 blocks of NW ints,
   NW*npes apart */
MPI_Type_vector(2, NW, NW*npes, MPI_INT, &fileblk);
MPI_Type_commit(                         &fileblk);
disp = (MPI_Offset)rank*NW*sizeof(int);
MPI_File_set_view(fh, disp, MPI_INT, fileblk, "native",
                  MPI_INFO_NULL);

/* processor writes 2 'ablk', which are NW ints each */
MPI_Type_contiguous(NW,   MPI_INT, &ablk);
MPI_Type_commit(&ablk);
MPI_File_write(fh, (void *)buf, 2, ablk, &status);
```

Collective I/O in MPI

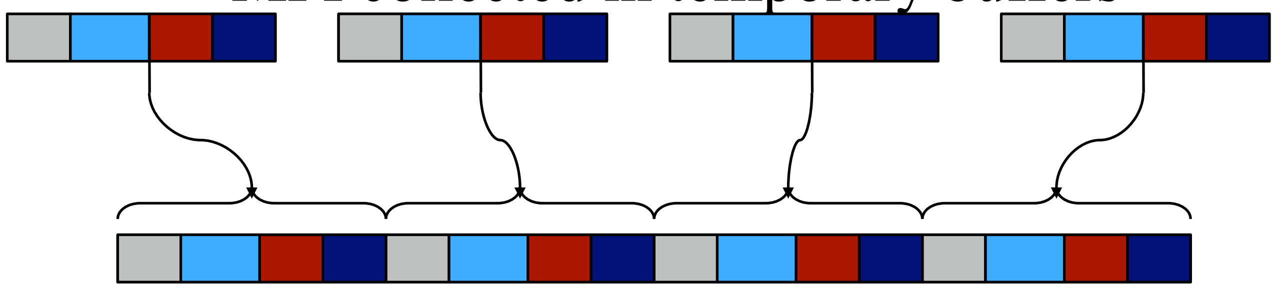
- A critical optimization in parallel I/O
- Allows communication of “big picture” to file system
- Framework for 2-phase I/O, in which communication precedes I/O (uses MPI machinery)
- Basic idea: build large blocks, so that reads/writes in I/O system will be large



COLLECTIVE I/O

Memory layout on 4 processors

MPI collected in temporary buffers



then written to File layout

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 requests 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

More advanced I/O

- Asynchronous I/O: **`iwrite/iread`**; terminate with **`MPI_Wait`**
- Split operations: **`read/write_all_begin/end`**; give the system more chance to optimize

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`**

MPI-2 predefined hints

- **`ind_rd_buffer_size`**
- **`ind_wr_buffer_size`**

New Algorithm Parameters

- **`start_iodevice`**
- **`pfs_svr_buf`**
- **`direct_read`**
- **`direct_write`**

Platform-specific hints

Summary of Parallel I/O Issues

- 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
- Use is encouraged, because I/O is expensive!
- In particular, when accesses are noncontiguous, users must create derived datatypes, define file views, and use the collective I/O functions

MPI-2 Status Assessment

- All vendors now have MPI-1. Free implementations (MPICH, LAM) support heterogeneous workstation networks.
- MPI-2 implementations are being undertaken now by all vendors.
  - Fujitsu, NEC have complete MPI-2 implementations
- MPI-2 implementations appearing piecemeal, with I/O first.
  - I/O available in most MPI implementations
  - One-sided available in some (e.g., NEC and Fujitsu, parts from SGI and HP, parts coming soon from IBM)
  - OpenMPI (*aka* LAM) and MPICH2 now becoming complete

References

- Using MPI by Gropp, Lusk and Skjellum
- Using MPI-2 by Gropp, Lusk and Thakur
- http://www.nersc.gov/vendor_docs/ibm/pe
- https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/ior/
- MPI 1.1 standard (http://www.mpi-forum.org/docs/mpi-11-html/node182.html)
- MPI 2 standard (http://www.mpi-forum.org/docs/mpi-20-html/node306.htm)