

---

# A Second Course in MPI

---

Science & Technology Support  
High Performance Computing

Ohio Supercomputer Center  
1224 Kinnear Road  
Columbus, OH 43212-1163



Ohio Supercomputer Center

# Table of Contents

---

- Data Transfer MPI Capabilities
  - Data decomposition and transferring boundary data
  - Single-sided Data Transfer
  - Probing vs. Message Receiving
- MPI support for Fortran 90 ?
- C++ syntax for MPI
- Introduction to MPI Parallel I/O

# Data Decomposition

---

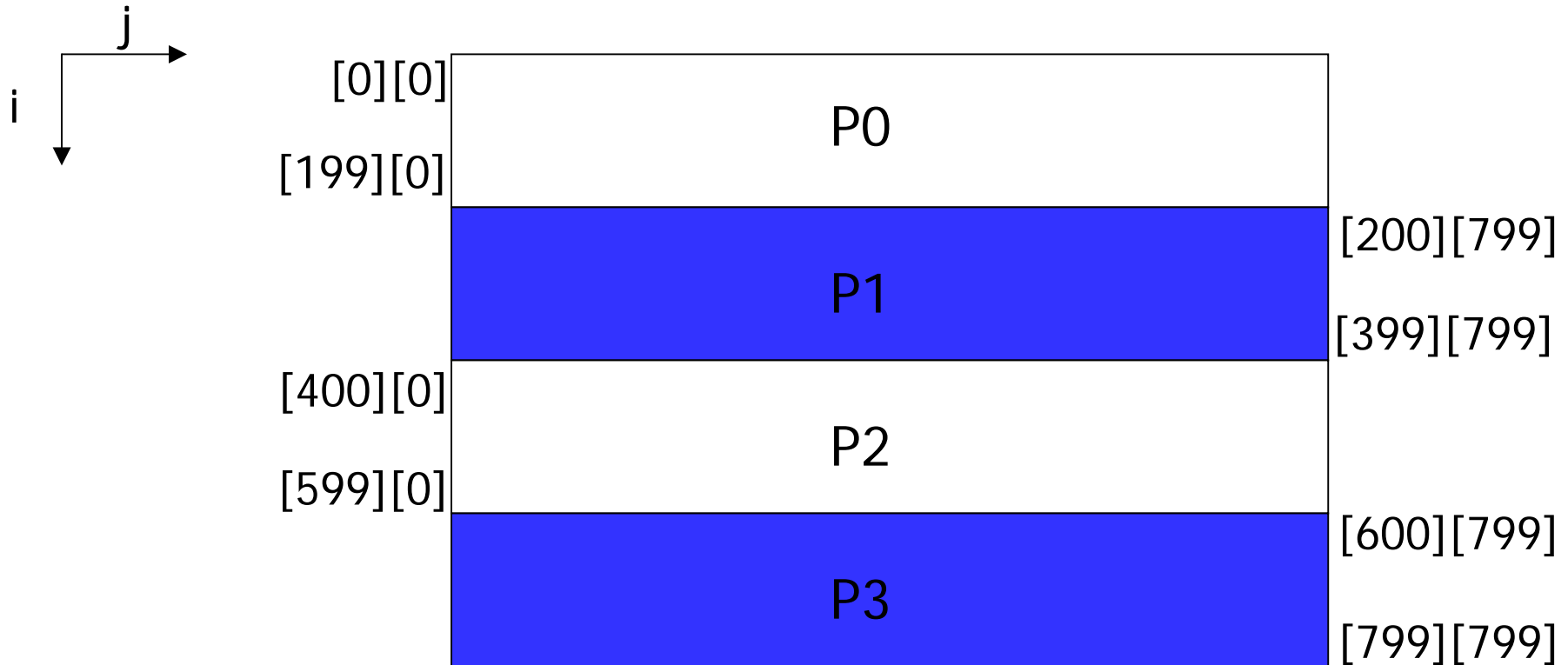
- Definition
- Memory storage decomposition
- Local arrays vs. Global array
- Alternate decomposition patterns

# Definition

---

- Simple idea: have each processor work on some subset of the data
  - If I have a wall to paint, I paint it all (serial). If I can get 7 friends to help, we each paint 1/8 th the area of the wall simultaneously (parallel)
  - Many terms for this parallel approach: Domain Decomposition, Domain Division, Data Decomposition, Different Data streams, etc.
  - How the programmer divides up the global data into local pieces for each process depends upon:
    - Number of parallel processors available
    - Algorithm to be used
    - Calculation/Communication time ratio
    - Data Cache Size
-

# Example



# Description

---

- This sample decomposition takes a 800 by 800 C array and assigns 200 rows to each of 4 processes
  - Dividing up the data as shown in the previous slide is based on how the array data is stored in memory. The “chunks” of data assigned to each process are contiguous in memory. (C 2-D array stored row-first)
  - Advantages of this “natural” decomposition
    - The default operation of all MPI communication routines – including the workhorse `MPI_Send()` and `MPI_Recv()` routines – is to pass as the message memory contiguous (stride 1) data
    - Alternative is to create your own MPI datatype that has a different memory template. For example, you could create a new MPI type `Column` in which C columns were transferred. (Several MPI routines exist for creating derived datatypes)
    - Some indexing registers are “load and increment” (in hardware!)
-

# Decomposition Code

---

- How does the programmer “assign” some set of data to a certain process?
- Have the evaluation of indexing array values dependent on rank:

```
const int N=200;  
istart = rank * N; iend=istart + (N-1);
```

- For P2 istart->400, iend->599, etc.
- Parallel loop nest:

```
for (i=istart; i < iend; ++i)  
  for (j=0; j < 1000; ++j) {  
    // Work done on the array  
    // MUST be independent of order of i,j  
  }
```

# Local vs. Global Arrays

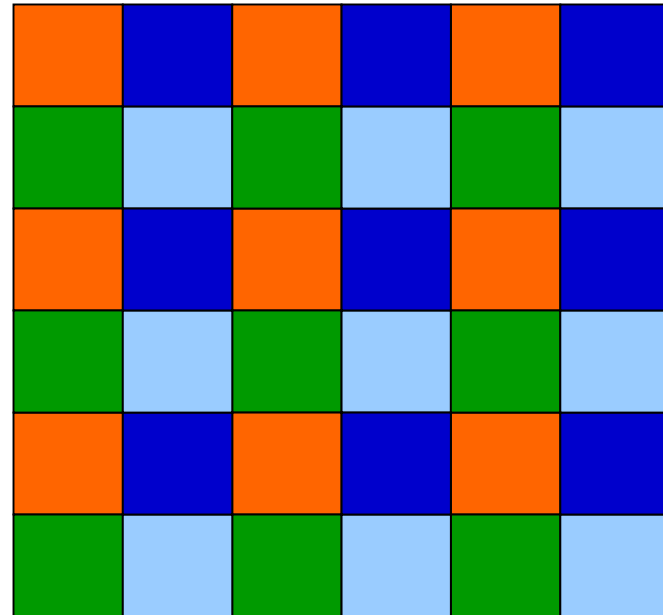
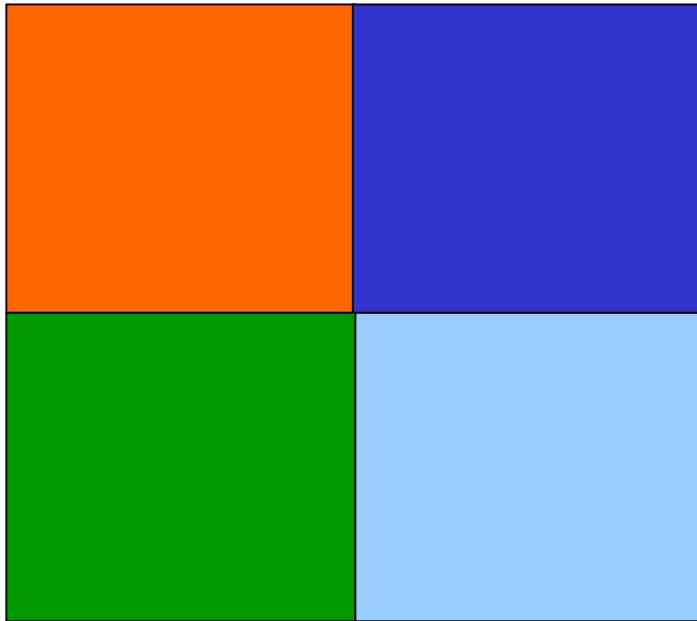
---

- In a serial version of code working on a 2-D array, the memory of the computer would contain all 800 x 800 of the elements of the data array
  - In this parallel version, each process' local memory has memory allocated for a smaller "local" array that is 200 x 800. For each process, it only knows and works on its local array.
  - Only the programmer knows that these separate local arrays are specific sections of a "global" array all the processes together are working on simultaneously
  - The programmer must initialize each local array with the appropriate data from the envisioned global array
-

# Alternate Decomposition Patterns

---

- Anything you want: quadrants, by columns, checkerboard, non-symmetric, decomposition hierarchy, etc.



# Parallel Stencil Program

---

- Stencil Definition
- Process' Data Boundaries
- Required message passing
- “Ghost” cells
- New local arrays

# Stencil

---

- In stencil code, an array is scanned element by element. The new value for a specific array element is calculated from the old value and some linear combination of its neighbors.
  - 5 element stencil: center, top, bottom, left, right
  - 9 element stencil: above + 4 diagonal neighbors
  - Plethora of uses for stencil algorithms:
    - Iterative solutions for partial differential equations in fluid mechanics, mechanical engineering, astrophysics, wave analysis, computational chemistry, high energy physics, diffusion phenomena, etc.
    - Computational sciences: cellular automata, grid mathematics, neural networks, etc.
    - Image processing
    - Scientific visualization: scene rendering, simulated lighting and surface characteristics, frame development, etc.
-

# Examples

---

- Finite-difference approximation to second-order partial derivatives

$$U_{i,j}^{n+1} = 1/4 (U_{i+1,j}^n + U_{i-1,j}^n + U_{i,j+1}^n + U_{i,j-1}^n)$$

	.25	
.25	0	.25
	.25	

- Image Sharpening

-1	-1	-1
-1	9	-1
-1	-1	-1

# Parallel Data Boundaries

---

- Assume we have the “natural” data distribution shown earlier and our problem uses a 5-stencil. Each of the processes are working on its 200  $i$  values simultaneously.
- Consider what happens when  $p2$  tries to update its boundary element  $U[599][10]$ . In its own memory it has this element’s left, right, and top neighbors:  $U[599][9]$ ,  $U[599][11]$ , and  $U[598][10]$  respectively.
- **BUT**, the bottom neighbor-  $U[600][10]$  -is in the local memory of  $p3$ .
- Thus, before  $p2$  can update its boundary elements MPI routines must be used by  $p3$  to send  $U[600][10]$  and by  $p2$  to receive it.

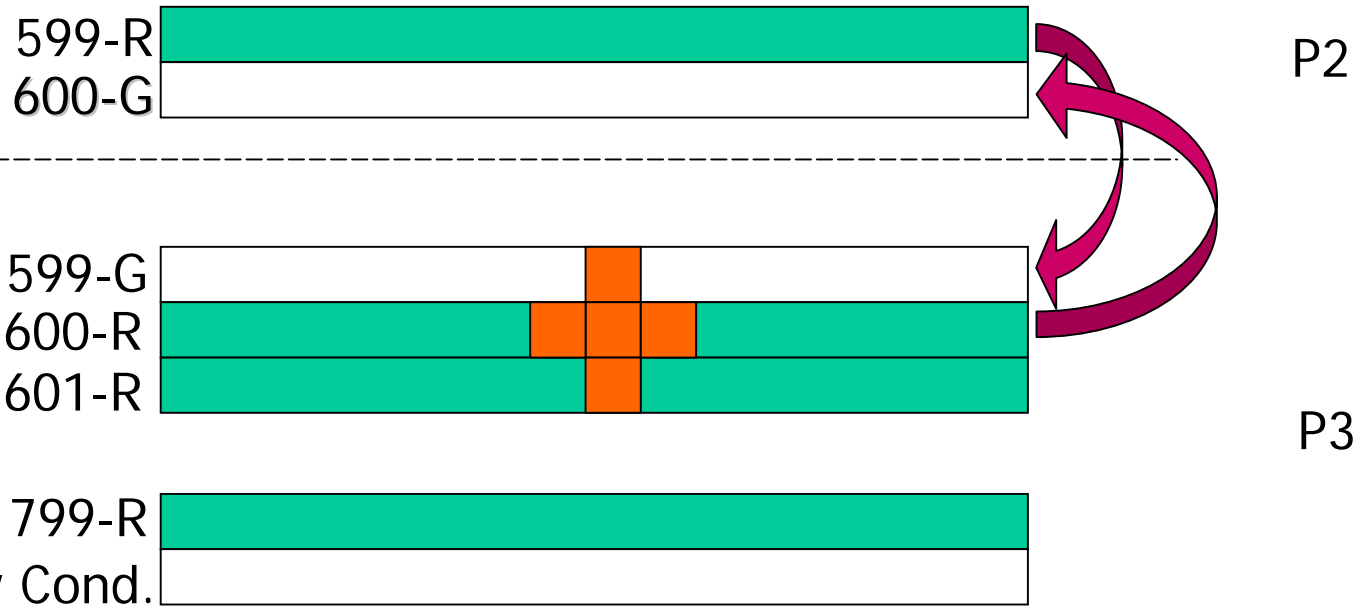
# Ghost Cells

---

- A popular method for handling the data transfer needed at the local array boundaries for each process is to make its local array have two extra rows, at the top and bottom.
- Before updating  $U$  for the next iteration, all the processes transfer boundary data to other processes and the values are placed in the extra rows.
- The extra “fake” elements of each process’ local array are referred to as “ghost cells”
- By using ghost cells each process can use its same values for  $istart$  and  $iend$ , and can use the code for the complete 5-stencil at all the “real” elements it is working on.

# New Local Array (p3)

```
double lu[202][800];
```



# Transfer Problem

---

- With the ghost cell approach, each processor has to both send and receive data from other processors. If written poorly the transfer can result in a **deadlock**.
- For example, if all processors first initiate MPI\_Sends and then MPI\_Recvs, The processors may be held up sending and can't receive.
- A common solution to this potential problem, is to have odd numbered MPI processes send first, while even numbered MPI processes recv first.
- MPI has developed a combined send and receive routine that guarantees that message sequencing will take place.

# MPI\_Sendrecv()

---

MPI\_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

- [ IN sendbuf] initial address of send buffer (choice)
  - [ IN sendcount] number of elements in send buffer (integer)
  - [ IN sendtype] type of elements in send buffer (handle)
  - [ IN dest] rank of destination (integer)
  - [ IN sendtag] send tag (integer)
  - [ IN recvbuf] initial address of receive buffer (choice)
  - [ IN recvcount] number of elements in receive buffer (integer)
  - [ IN recvtype] type of elements in receive buffer (handle)
  - [ IN source] rank of source (integer)
  - [ IN recvtag] receive tag (integer)
  - [ IN comm] communicator (handle)
  - [ OUT status] status object (Status)
-

# MPI\_Sendrecv() Program

---

```
if (myid == 0) then
  do i=1,4
    a(i)=i*101.
    b(i)=i+500.
  end do
else
  do i=1,4
    a(i)=i
    b(i)=i+1.
  end do
end if

if (myid == 0) then
  call mpi_sendrecv(a,4,mpi_real,1,33,b,4,mpi_real,1,66,MPI_COMM_WORLD,
&                                                              status,ierr)
elseif (myid == 1) then
  call mpi_sendrecv(b,4,mpi_real,0,66,a,4,mpi_real,0,33,MPI_COMM_WORLD,
&                                                              status,ierr)
end if
```

# Program Output

---

```
P:1 a=101.0000 202.0000 303.0000 404.0000  
    b=2.000000 3.000000 4.000000 5.000000
```

```
P:0 a=101.0000 202.0000 303.0000 404.0000  
    b=2.000000 3.000000 4.000000 5.000000
```

# Single-sided Data Transfer

---

- As is well known, the most common MPI data transfer is to have one processor post an `MPI_Send()` and a *different* processor post an `MPI_Recv()`.
- There are disadvantages to this “Two-sided” approach:
  - Two processors are involved and their chip circuitry must be used for data cache storage
  - With two MPI calls, the chances for syntax errors increases
  - With two MPI calls, the chances for logical errors increases
- An alternative to send → recv are the MPI functions `MPI_Get()` and `MPI_Put()`. To cause a data transfer between local memories only **one** processor has to call only **one** of these routines.

# Single-sided Terminology

---

- **Origin** = buffer in the calling processor's memory
- **Target** = buffer in the remote processor's memory
- A **Window** must be create before an MPI\_Get() or MPI\_Put() can be used. A window is a part of the target memory made accessible to remote processes.
- A **Fence** provides synchronization for the RMA (Remote Memory Access). When the Fence call is first made, the Window is open for data transfer. The RMA is then performed and a second (bracketing) Fence call closes the Window.

# Making a Window

---

`MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)`

**IN** `base` -- initial address of window (choice)

Typically the address of the array to be transferred

**IN** `size` -- size of window in bytes (nonnegative integer)

`MPI_Type_extent()` handy for the conversion

Often the total size of the array to be transferred

**IN** `disp_unit` -- local unit size for displacements, in bytes (positive integer)

**IN** `info` -- info argument (handle)

**IN** `comm` -- communicator (handle)

**OUT** `win` -- window object returned by the call (handle)

---

# Building a Fence

---

`MPI_WIN_FENCE(assert, win)`

**IN** `assert` -- program assertion (integer)

This argument allows the user to customize and optimize the RMA by setting desired assertions

0 is the general case: no special guarantees are made

**IN** `win` -- window object (handle)

The previous created window object that the fence will control

# Transferring Data (finally)

---

`MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)`

OUT `origin_addr` -- initial address of origin buffer (choice)

IN `origin_count` -- number of entries in origin buffer (integer)

IN `origin_datatype` -- datatype of each entry in origin buffer (handle)

IN `target_rank` -- rank of target (integer)

IN `target_disp` -- displacement from window start to the beginning of the target buffer (integer)

IN `target_count` -- number of entries in target buffer (integer)

IN `target_datatype` -- datatype of each entry in target buffer (handle)

IN `win` -- window object used for communication (handle)

# Putting it all together (prep work)

---

```
REAL A(m),B(m)
INTEGER index(m)
DATA index /3,10,8,4,6,9,1,5,2,7/

IF (rank .eq. 1) THEN
  DO i=1,m
    B(i)=i**2 ! Array to be "gotten" by rank 0 process
  END DO
END IF

CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
  ! Determines how many bytes there are in a REAL
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal,
  MPI_INFO_NULL,MPI_COMM_WORLD, win, ierr)
```

---

# Getting the Data

---

```
CALL MPI_WIN_FENCE(0, win, ierr)
  IF (rank .eq. 0) THEN
    DO i=1,m
      k=index(i)
CALL MPI_GET(A(i),1,MPI_REAL,1,k-1,1,MPI_REAL,win,ierr)
    END DO
  END IF
CALL MPI_WIN_FENCE(0, win, ierr)

PRINT *, "P:", rank
DO i=1,m
  print *, "A(", i, ")=", A(i)
END DO
CALL MPI_WIN_FREE(win, ierr)
```

---

# What we got ...

---

```
P: 0
A( 1) = 9.00000
A( 2) = 100.0000
A( 3) = 64.0000
A( 4) = 16.0000
A( 5) = 36.0000
A( 6) = 81.0000
A( 7) = 1.00000
A( 8) = 25.0000
A( 9) = 4.00000
A(10) = 49.0000
```

# Message Probing

---

Message Passing Review

Message Unknowns: Tag, Source, and Count

Message Probing

Dynamic Memory Reception

Message Tag Signaling

Receiving Different Data Types

# Canonical Program

---

```
double eleven[500];
double data[500];

if (rank==1) {
    for(i=0;i<50;++i) { eleven[i]=11.0; }
    MPI_Send(eleven,50,MPI_DOUBLE,0,1001,MPI_COMM_WORLD);
}else{
    MPI_Recv(data,500,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG
,MPI_COMM_WORLD, &status);
    printf("Message from proc %d\n",status.MPI_SOURCE);
    printf("Message has tag %d\n",rank,status.MPI_TAG);
    MPI_Get_count(&status,MPI_DOUBLE,&count);
    printf("Received %d doubles\n",rank,count);
}
```

---

# Program Output

---

Message from proc 1

Message has tag 1001

Received 50 doubles

# Message Unknowns

---

- What is wrong with the previous program?
  - The receiver doesn't know how big to make the receiving array
  - Result: MEMORY WASTE (500 compared to 50)
  - Because of wildcarding, receiver doesn't know tag until after reception
    - Usually unimportant, but tag value may be a signal
  - Because of wildcarding, receiver doesn't know source until after reception
    - May be important: receiver performs different tasks depending on who the message is from
- Solution: Probe the message **BEFORE** reception to find out count, tag, and source
  - Use `MPI_Probe()` routine

# Message Probing

---

`MPI_Probe( int source, int tag, MPI_Comm comm, MPI_Status info)`

-How to message probe

-Call `MPI_Probe` function with source and tag arguments filled with integer values or their Wildcards. (Communicator `comm` must always be specified by the programmer)

-When a message is found that matches the Probe, the status variable *info* is filled with the **exact same data** it would be if an actual reception had occurred.

-Based on the information extracted from *info*, perform whatever steps are necessary before reception.'

-Finally, post the `MPI_Recv()` and actually receive the message.

# Dynamic Memory Reception

---

- The most common use of Message Probing
- Probe the message to find exactly how many elements are being sent
- At that point in your program dynamically allocate memory to an array whose size exactly matches the element count.
- Receive the message in the normal fashion with the dynamically created array
- N.B. , both C and Fortran 90 have dynamic memory allocations for arrays.

# Sample Program

---

```
double value[500];
double *data;
if (rank==1) {
    for(i=0;i<50;+i) { value[i]=11.0+i; }
    MPI_Send(value,50,MPI_DOUBLE,0,1001,MPI_COMM_WORLD);
} else {
    MPI_Probe(1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    MPI_Get_count(&status,MPI_DOUBLE,&count);
    MPI_Type_size(MPI_DOUBLE,&n);
    data= (double*) calloc(count,n);
    MPI_Recv(data,count,MPI_DOUBLE,1,MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
    for(i=0;i<count;i+=10) {
        printf("data element %d is %f\n",i,data[i]);}
}
```

---

# Program Results

---

data element 0 is 11.000000

data element 10 is 21.000000

data element 20 is 31.000000

data element 30 is 41.000000

data element 40 is 51.000000

# Message Tag = Signal

---

- Although the tag is, in general, not useful to the programmer, it can be.
- Personal Experience: had an `MPI_Recv()` in a while loop and didn't know how many messages it was to receive. Had the source send an empty message with a special tag value that indicated no more messages coming.
- Procedure used: PROBE each message and check the tag value. If the message had the semaphore tag, don't even bother to receive it. Just exit the while loop. If it had a "normal" tag, receive it and stay in the loop waiting for more messages.
- This procedure is shown in our sample code

# Sample Program

---

```
int rank,i,done=0;
MPI_Status status;
double set[100],array[100],nil=0;
if (rank==1) {
    for(i=0;i<100;++i) { set[i]=i; }
    MPI_Send(set,100,MPI_DOUBLE,0,1001,MPI_COMM_WORLD);
    MPI_Send(&nil,0,MPI_DOUBLE,0,33,MPI_COMM_WORLD);
} else {
    while (!done) {
        MPI_Probe(1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        if (status.MPI_TAG == 1001 ) {
            MPI_Recv(array,100,MPI_DOUBLE,1,MPI_ANY_TAG,
                    MPI_COMM_WORLD,&status);
            printf("50 element is %f\n",array[50]); }
        if (status.MPI_TAG == 33 ) { done=1; } } }
```

**OUTPUT = 50 element is 50.000000**

---

# Different Data Types

---

- Which message parameter has the receiver not been able to “wildcard” in some way?
- Answer: The MPI Data type. The receiving process has to know *before* reception what type is being sent.
- We can use MPI\_Probe() to allow the receiver to work with different data types.
- Procedure: Have several processes send messages to the receiver. Each source sends a message with data of different types. PROBE each message, and get the source. Finally, post an MPI\_Recv() which specifies a data type that corresponds to that particular source.

# Sample Program

---

```
char lett[13]="Robert Plant"; int num[100];
if (rank==1) {
    for(i=0;i<100;++i) {num[i]= (int) 300 * drand48();}
    MPI_Send(num,100,MPI_INT,0,11,MPI_COMM_WORLD); }
if (rank==3) {
    MPI_Send(lett,13,MPI_CHAR,0,11,MPI_COMM_WORLD); }
if (rank==0) {
    for(i=0;i<2;++i) {
        MPI_Probe(MPI_ANY_SOURCE,11,MPI_COMM_WORLD,&state);
        if (state.MPI_SOURCE == 1) {
            MPI_Recv(num,100,MPI_INT,1,11,MPI_COMM_WORLD,&state);
            divisible(num); }
        if (state.MPI_SOURCE == 3) {
            MPI_Recv(lett,13,MPI_CHAR,3,11,MPI_COMM_WORLD,&state);
            up(lett); } } }
```

---

# Program Output

---

-The external function `divisible(num)` outputs the index and value of those elements in `num` that are evenly divisible by five.

-The external function `up(lett)` outputs all the characters in the string `lett` as upper-case.

-Thus, the program output is:

P:0 The name is ROBERT PLANT

P:0 index=2 value=0 is divisible by 5

P:0 index=16 value=270 is divisible by 5

P:0 index=19 value=95 is divisible by 5

P:0 index=23 value=55 is divisible by 5

P:0 index=46 value=165 is divisible by 5

P:0 index=47 value=220 is divisible by 5

P:0 index=48 value=195 is divisible by 5 ...

---

# Fortran 90 and MPI-1

---

Introduction

Fortran 90 Syntax

Fortran 90 Records

KIND Variables

Allocatable Arrays

Arrays Sections

Array Indexing

Pointer and Targets



# Introduction

---

- Socratic chapter approach
- In each section a question is asked, then answer provided, and finally sample code
- “I am writing a code using this Fortran 90 feature, how can I run it in parallel with MPI-1?”

# New Fortran 90 Syntax

---

- Q: “Can I even use MPI routines in a Fortran 90 program?”
  - The most basic question of all
- A: Yes, because Fortran 90 is backward-compatible with Fortran 77
- In the sample program, two MPI processes exchange arrays, but all the code syntax is Fortran 90

# Sample Program

---

```
integer :: count
real    :: data(10),value(20)
integer :: err,rank,count,status(MPI_STATUS_SIZE)
call MPI_INIT(err)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
if (rank==1) then
  data=3.0
  call MPI_SEND(data,10,MPI_REAL,0,55,MPI_COMM_WORLD,err)
else
  call MPI_RECV(value,20,MPI_REAL,MPI_ANY_SOURCE,55, &
               MPI_COMM_WORLD, status,err)
  print *,"P:",rank," Message from proc ",status(MPI_SOURCE)
  call MPI_GET_COUNT(status,MPI_REAL,count,err)
  print *,"P:",rank," got ",count," elements"
  print *,"P:",rank," value array is"
  print *,value
end if
```

# Program Output

---

P:0 Message from proc 1

P:0 got 10 elements

P:0 value array is

3.000000	3.000000	3.000000	3.000000
3.000000			
3.000000	3.000000	3.000000	3.000000
3.000000			
0.000000E+00	0.000000E+00	0.000000E+00	
0.000000E+00	0.000000E+00		
0.000000E+00	0.000000E+00	0.000000E+00	
0.000000E+00	0.000000E+00		

# Fortran 90 Records

---

- Q: “Can I transfer Fortran 90 records variables with MPI routines?”
- A: Yes, because a user can define their own MPI\_datatype that matches the memory stencil of the record
- Sample program will work with a record type *card* which has two members representing the number and suit of a playing card

# Sample Program (Creating MPI\_Card)

---

```
type card
  integer :: number
  character (LEN=8) :: suit
end type card
type(card), dimension(13) :: suit1,suit0

integer, dimension(2) :: blength=(/1,8/)
integer, dimension(2)::types=(/MPI_INTEGER,MPI_CHARACTER/)
integer, dimension(2) :: delta=(/0,0/)
integer :: intex,MPI_CARD

call MPI_TYPE_EXTENT(MPI_INTEGER,intex,err)
delta(2)=intex
call MPI_TYPE_STRUCT(2,blength,delta,types,MPI_CARD,err)
call MPI_TYPE_COMMIT(MPI_CARD,err)
```

---

# Sample Program (Copying Records)

---

```
if (rank==1) then
  suit1=card(2,"Clubs") ! All suit1 cards are clubs
  do i=1,13
    suit1(i)%number=i+1
  end do
  call MPI_SEND(suit1,13,MPI_CARD,0,5,MPI_COMM_WORLD,err)

else

  call MPI_RECV(suit0,13,MPI_CARD,1,MPI_ANY_TAG, &
               MPI_COMM_WORLD,status,err)
  print *,"P:",rank," suit0 card array is"
  print *,suit0
end if
```

---

# Program Output

---

```
P:0  suit0 card array is
```

```
2 Clubs 3 Clubs 4 Clubs 5 Clubs 6 Clubs 7 Clubs
```

```
8 Clubs 9 Clubs 10 Clubs
```

```
11 Clubs 12 Clubs 13 Clubs 14 Clubs
```

# KIND Variables

---

- Q: How can I transfer data between Fortran 90 variables declared to be a specific KIND?
- Programmer must do preliminary work to find out the size (in bytes) of the KIND variables they are using
- This size can then be use to set MPI\_Datatype for communication

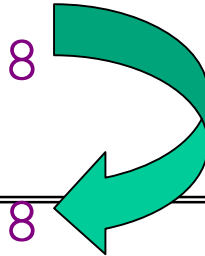
# Preliminary Program

```
integer, parameter :: r10=selected_real_kind(10)
real(kind=r10)::a
real(kind=KIND(1.0D0))::b
integer :: bytes
if (rank==0) then
  print *,range(a),precision(a),kind(a)
  call MPI_TYPE_EXTENT(MPI_REAL,bytes,err)
  PRINT *, "kind=",kind(REAL), "r_extent=", bytes
  call MPI_TYPE_EXTENT(MPI_DOUBLE_PRECISION,bytes,err)
  PRINT *, "kind=",kind(b), "dp_extent=", bytes
end if
```

307 15

kind=4 r\_extent=4

kind=8 dp\_extent=8



# Sample Program

---

```
integer, parameter :: r10=selected_real_kind(10)
real(kind=r10)::x(1),y(1)
if (rank == 0) then
    x(1)=exp(1.0)
    write(6,30)x(1)
30  format("x=",f15.13)
    call MPI_SEND(x,1,MPI_REAL8,1,810,MPI_COMM_WORLD,err)
else
call MPI_RECV(y,1,MPI_REAL8,0,810,MPI_COMM_WORLD,info,err)
    write(6,20) y(1)
20  format("y=",f15.13)
end if
-----
x=2.7182818284590
y=2.7182818284590
```

---

# MPI\_TYPE\_MATCH\_SIZE()

---

MPI\_TYPE\_MATCH\_SIZE(typeclass, size, type)

IN typeclass      generic type specifier (integer)  
IN size            size, in bytes, of representation (integer)  
OUT type          datatype with correct type, size (handle)

- Giving this subroutine a generic type specifier (see program) and a size of a certain KIND of variable, it will return the MPI\_DATATYPE that matches the KIND of variables you wish to work with

# Type Match F90 Program

---

```
integer, parameter :: r10=selected_real_kind(10)
real(kind=r10)::a
integer :: a_size
integer :: mpitype

if (rank==0) then
  print *,range(a),precision(a),kind(a)
  call MPI_SIZEOF(a,a_size)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL,a_size,mpitype,err)
  print *,"mpitype=",mpitype
end if
```

# Program Output and Analysis

---

```
307 15 8
```

```
mpitype= 29
```

- The above is the output of our program. It tells us that the `MPI_DATATYPE` that will match the “r10” `KIND` variable has integer code 29.
- Looking at the file `mpif.h` we find the `MPI_DATATYPE` that matches code 29:  

```
parameter (MPI_REAL8=29)
```

# Message Passing with KIND data

---

```
integer, parameter :: r10=selected_real_kind(10)
real(kind=r10)::a,b=0.0,c=0.0
integer :: a_size
integer :: mpitype
integer :: status(MPI_STATUS_SIZE)
call MPI_SIZEOF(a,a_size)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL,a_size,mpitype,err)

if(rank==0) then
  b=2.718281828459045
  call MPI_SEND(b,1,mpitype,1,55,MPI_COMM_WORLD,err)
end if
if (rank==1) then
  call MPI_RECV(c,1,mpitype,0,55,MPI_COMM_WORLD,status,err)
  print *, "P:",rank, "c=",c
end if
```

P:1 c=2.7182817459106445

# Allocatable Arrays

---

- Q: “Can an MPI process create an allocatable array and then duplicate it exactly in another process’ memory?”
- Yes, but the sending process has to provide size information to the receiver before the actual data transfer is made.

# Sample Program

---

```
real, dimension(:, :), allocatable :: data, value
integer, dimension(2) :: dim
if (rank==1) then
    read (10,*)row,col ! Reads in 5 rows 8 columns
    allocate(data(row,col))
    dim=shape(data)
    call MPI_SEND(dim,2,MPI_INTEGER,0,77,MPI_COMM_WORLD,err)
    data=reshape( (/ ((real(i*j),i=1,row),j=1,col) /),SHAPE=(/row,col/))
    call MPI_SEND(data,row*col,MPI_REAL,0,55,MPI_COMM_WORLD,err)
else
    call MPI_RECV(dim,2,MPI_INTEGER,1,77,MPI_COMM_WORLD, status,err)
    allocate(value(dim(1),dim(2)))
    call MPI_RECV(value,dim(1)*dim(2),MPI_REAL,1,55,MPI_COMM_WORLD,&
                 status,err)
    print *, "value array is:"
    print *, value
end if
```

---

# Program Output

---

value array is:

1.000000	2.000000	3.000000	4.000000
5.000000	6.000000	7.000000	8.000000
2.000000	4.000000	6.000000	8.000000
10.000000	12.000000	14.000000	16.000000
3.000000	6.000000	9.000000	12.000000
15.000000	18.000000	21.000000	24.000000
4.000000	8.000000	12.000000	16.000000
20.000000	24.000000	28.000000	32.000000
5.000000	10.000000	15.000000	20.000000
25.000000	30.000000	35.000000	40.000000

---

# Array Sections

---

- Q: "Can a section of an existing array be used explicitly in MPI-1 communication routines?"
- Yes, the array section notation is recognized (and converted)
- In the sample program, we will transfer 3 elements of a 10 element array *data*. The elements are *data(3)*, *data(5)*, and *data(7)*

# Sample Program

---

```
real :: data(10),part(7)
...
if (rank==1) then
    data=(/ (REAL(i)**2,i=1,10) /)
call MPI_SEND(data(3:7:2),3,MPI_REAL,0,11,MPI_COMM_WORLD,&
              err)
else
    call MPI_RECV(part,7,MPI_REAL,1,11,MPI_COMM_WORLD, &
                 status,err)
    call MPI_GET_COUNT(status,MPI_REAL,count,err)
    print *, "P:",rank, " got ",count, " elements"
    do i=1,7
        print *, "P:",rank, " part index value=",i,part(i)
    end do
end if
```

---

# Program Output

---

```
P:0 got 3 elements
P:0 part index value= 1    9.000000
P:0 part index value= 2   25.000000
P:0 part index value= 3   49.000000
P:0 part index value= 4   0.000000E+00
P:0 part index value= 5   0.000000E+00
P:0 part index value= 6   0.000000E+00
P:0 part index value= 7   0.000000E+00
```

# Array Indexing

---

- Q: "Is non-default array indexing allowed in MPI-1 Message Passing?"
- A: Yes, whatever index value range is chosen by the user will be recognized both in `MPI_SEND()` and `MPI_RECV()`
- In our program an array with "axes-dimensioning" is sent to a Fortran array with C indexing

# Sample Program

---

```
real :: data(-5:5),part(0:9)
...
if (rank==1) then
  data=(/ (REAL(i)**2,i=-5,5) /)
  call MPI_SEND(data,10,MPI_REAL,0,1111,MPI_COMM_WORLD,err)
else
  call MPI_RECV(part,10,MPI_REAL,1,1111,MPI_COMM_WORLD, &
    status,err)
  call MPI_GET_COUNT(status,MPI_REAL,count,err)
  print *, "P:",rank," got ",count," elements"
  do i=0,9
    print *, "P:",rank," part index value=",i,part(i)
  end do
end if
```

---

# Program Output

---

```
P:0 got 10 elements
P:0 part index value= 0    25.00000
P:0 part index value= 1    16.00000
P:0 part index value= 2     9.000000
P:0 part index value= 3     4.000000
P:0 part index value= 4     1.000000
P:0 part index value= 5    0.000000E+00
P:0 part index value= 6     1.000000
P:0 part index value= 7     4.000000
P:0 part index value= 8     9.000000
P:0 part index value= 9    16.00000
```

# Pointers and Targets

---

- Q: "Can Fortran 90 pointer data be transferred in MPI-1 Message Passing?"
- Yes, relatively simply
- It is interesting to see what happens to the targets the transferred pointers are associated with.
- The changes in the targets is both straightforward and disturbing at the same time ...

# Sample Program

---

```
real,pointer :: p0,p1
real,target :: t0,t1
if (rank==0) then
    t0=19.0
    p0=>t0
    call MPI_SEND(p0,1,MPI_REAL,1,55,MPI_COMM_WORLD,err)
else
    t1=52.0
    p1=>t1
    print *,"P:",rank,"t1 before recv",t1
    call MPI_RECV(p1,1,MPI_REAL,MPI_ANY_SOURCE,55, &
                 MPI_COMM_WORLD,status,err)
    print *,"P:",rank,"t1 after recv",t1
end if
```

# Program Output

---

```
P:1 t1 before recv  52.00000  
P:1 t1 after  recv  19.00000
```

Notable Feature: Process 0 changed the value of *t1* without actually sending a message to it, but by going through its pointer *p1*.

# C++ and MPI-1

---

- MPI-2 C++ Binding Namespace
- Reference Variable Transfer
- Structures: Byte Stream Transfer
- Object Transfer
- Transferring Inheritance traits

# MPI-2 C++ Bindings

---

- Some MPI libraries have incorporated the MPI-2 feature of providing C++ bindings for MPI functions.
- The MPI-2 strategy is to use a new namespace called MPI and put a number of high-level Class definitions in it.
- Each MPI namespace Class is required to have a default constructor, a destructor, a copy constructor, and assignment operator
- N.B., This is just a syntactical addition. The same C MPI capabilities are presented in C++ form: no new capabilities have been added
  - For example, no new routines for passing objects

# MPI Namespace

---

```
namespace MPI {  
  class Comm { ... }  
  class Intracomm : public Comm { ... }  
  class Graphcomm : public Intracomm { ... }  
  class Cartcomm : public Intracomm { ... }  
  class Intercomm : public Comm { ... }  
  class Datatype { ... }  
  class Errhandler { ... }  
  class Exception { ... }  
  class Group { ... }  
  class Op { ... }  
  class Request { ... }  
  class Prequest : public Request { ... }  
  class Status { ... }  
};
```

---

# Class Member Functions

---

- The actual C++ version of the MPI functions you want to use in your program are member functions of one of the MPI namespace classes.
- Thus, the general syntax to reference a C++ MPI function is  
MPI::`<class object>.<mpi function>`
- As an example, the C++ version of the broadcast function for the “World” communicator object would be used with the following syntax:  

```
MPI::COMM_WORLD.Bcast(data, 100, MPI::INT, 5);
```
- As evident in this example, the symbolic names for MPI constants have also changed in the C++ version

# Point-to-Point Program

---

- On the next page is shown the “classic” beginning MPI program in which one MPI process sends data to another.
- But, it has been written entirely with MPI-2 C++ versions of the necessary functions and constants.
- Reference: The MPI-2 Standard Document

## P2P C++ Program (Send)

---

```
#include <mpi.h>
#include <iostream.h>
int main(int argc, char *argv[]) {
    int rank,size;
    double num[50],data[500];
    MPI::Status Sitrep;
    MPI::Init(argc,argv);
    size=MPI::COMM_WORLD.Get_size();
    rank=MPI::COMM_WORLD.Get_rank();
    if (rank==0) {
        for (int i=0;i<50;++i)
            num[i]=i+size;
        MPI::COMM_WORLD.Send(num,50,MPI::DOUBLE,1,45);
```

---

## P2P C++ Program (Receive)

---

```
}else{
    MPI::COMM_WORLD.Recv(data, 500, MPI::DOUBLE,
        MPI::ANY_SOURCE, MPI::ANY_TAG, Sitrep);
    cout << "P:" << rank << " source=" <<
        Sitrep.Get_source() << endl;
    cout << "P:" << rank << " tag=" << Sitrep.Get_tag()
        << endl;
    int count=Sitrep.Get_count(MPI::DOUBLE);
    cout << "P:" << rank << " count=" << count << endl;
    for (int i=0;i<count;i+=10) {
        cout << "i=" << i << " data[i]=" << data[i] << endl;
    }
}
MPI::Finalize(); return 0; }
```

---

# Reference Variables

---

- Reference variables are a C++ addition to the combined languages in which one memory location can be given two names.
- The name of the declared variable is one, the name of a reference variable assigned to the “real” name is the other
- Put into the language to allow direct, readable transfer-by-reference between dummy arguments and actual arguments
- Even though there is no reference variable `MPI_Datatype`, the following code shows how the contents of a reference variable can be transferred between two MPI processes.

```
double x;
double& y=x;
if (rank==0) {
    x=3.78;
    printf("P:%d x=%f\n",rank,x);
    printf("P:%d y=%f\n",rank,y);
    MPI_Send( &y,8,MPI_BYTE,1,33,MPI_COMM_WORLD);
} else {
    MPI_Recv(&y,8,MPI_BYTE,0,33,MPI_COMM_WORLD,&state);
    printf("P:%d x=%f\n",rank,x);
    printf("P:%d y=%f\n",rank,y);
    y=8.7;
}
printf("P:%d x=%f\n",rank,x);
printf("P:%d y=%f\n",rank,y);
```

---

# Reference Program Output

---

P:0 x=3.780000 // Before Send

P:0 y=3.780000

P:1 x=3.780000 // After Recv

P:1 y=3.780000

P:0 x=3.780000 // After if-else

P:0 y=3.780000

P:1 x=8.700000

P:1 y=8.700000

# Structures as Byte Streams

---

- As you were taught in an introductory MPI course the easiest and clearest method for transferring a structure was to make your own new, derived MPI\_Datatype that matched the memory layout of the structure.
- An alternate (less elegant) approach is to just pass each data item in the C structure one-by-one as a stream of bytes.
- Both approaches **DO** the same thing: transfer the memory map of the structure variable
- The following code shows this “new” method

# Structure Program

---

```
struct particle{  
    int id;  
    double x,y,z;  
};
```

```
static struct particle atom={3492,3.45,-0.58,13.26};
```

```
if (rank==0) {  
    printf("P:%d atom id =%d\n",rank,atom.id);  
    printf("P:%d atom position %f %f %f\n",rank,atom.x  
        atom.y,atom.z);  
    MPI_Send( &atom,28,MPI_BYTE,1,23,MPI_COMM_WORLD);  
}else{
```

---

# Structure Reception

---

```
MPI_Recv(&atom, 28, MPI_BYTE, 0, 23, MPI_COMM_WORLD,  
        &sitrep);
```

```
printf("P:%d atom id =%d\n", rank, atom.id);  
printf("P:%d atom position %f %f %f\n", rank, atom.x,  
        atom.y, atom.z);
```

```
MPI_Get_count(&sitrep, MPI_BYTE, &count);  
printf("P:%d recv byte count is %d\n", rank, count);  
}
```

# Structure Program Output

---

```
P:0 atom id =3492 // Before Send
```

```
P:0 atom position 3.450000 -0.580000 13.260000
```

```
P:1 atom id =3492 // After Recv
```

```
P:1 atom position 3.450000 -0.580000 13.260000
```

```
P:1 recv byte count is 28 // Just checking ..
```

# Object Messaging Passing

---

- By far the most asked question from C++ programmers. **How can I send an object from one process to another?** (Using only MPI –1 routines)
  - Answer: just send the data members
  - When a Class is defined every object created knows the address (in its local memory) of the function members through a look-up table.
  - When a specific object calls a member function that address is accessed and the “this” pointer for that object is passed as an argument
  - The key is that the address table is known to both the sending and receiving processes.
  - In the following code and object is transferred as a data stream
-

# Object Definition

---

```
class Triangle {
    double base;
    double height;
public:
    void set(double a,double b) {
        base=a; height=b;
    }
    void display() {
        cout << "base=" << base << endl;
        cout << "height=" << height << endl;
    }
    double area() { return (0.5*base*height); }
};
```

# Object Transfer

---

```
Triangle t,s;
if (rank==0) {
    t.set(12.0,9.25);
    t.display();
    cout << "P:" << rank << " area=" << t.area() << endl;
    MPI_Send( &t,16,MPI_BYTE,1,23,MPI_COMM_WORLD);
}else{
    MPI_Recv( &s,16,MPI_BYTE,0,23,MPI_COMM_WORLD,&sitrep);
    s.display();
    cout << "P:" << rank << " area=" << s.area() << endl;
}
```

# Object Program Output

---

```
base=12          // Before Send, from Process 0
```

```
height=9.25
```

```
P:0 area=55.5
```

```
base=12          // After Recv, from Process 1
```

```
height=9.25
```

```
P:1 area=55.5
```

# Object “Datatype”

---

- Since (as shown) only the data members of an object need to be transferred, one can create a new derived MPI\_Datatype for the memory layout of the data members.
- The derived MPI\_Datatype can then be used in the message passing as an alternative to a byte stream.
- In the following code, the new MPI\_Datatype called MPI\_TRI is created for use with Triangle objects. The output is the same as the previous program.

# MPI\_TRI Program

---

```
MPI_Datatype MPI_TRI;
MPI_Type_contiguous(2,MPI_DOUBLE,&MPI_TRI);
MPI_Type_commit(&MPI_TRI);

if (rank==0) {
    t.set(12.0,9.25);
    t.display();
    cout << "P:" << rank << " area=" << t.area() << endl;
    MPI_Send( &t,1,MPI_TRI,1,23,MPI_COMM_WORLD);
}else{
    MPI_Recv( &s,1,MPI_TRI,0,23,MPI_COMM_WORLD,&sitrep);
    s.display();
    cout << "P:" << rank << " area=" << s.area() << endl;
}
```

---

# Inheritance

---

- We have just seen two methods for transferring objects, but what if the object is a member of a derived class?
- Programmers would like the received object to also inherit the capabilities of the base class.
- This will be accomplished if **ALL** of the data members in the inheritance “lineage” are transferred.
- In the following program, there is a base class and one derived class; an object of the derived class will be sent and received

# Inheritance Base Class

---

```
class Data {  
    protected:  
        double x[N],y[N];  
public:  
    void getdata() {  
        FILE *dfile;  
        dfile=fopen("data.in","r");  
        for (int i=0;i<N;++i)  
            fscanf(dfile,"%lf%lf",&x[i],&y[i]);  
        fclose(dfile);  
    }  
    void display() {  
        for (int i=0;i<N;++i)  
            cout << "i=" << i << " x=" << x[i] << " y=" <<  
            y[i] <<endl;  
    } };
```

---

# Inheritance Derived Class

---

```
class Average: public Data {
    double mux,muy;
public:
    void averagex() {
        double sum=0.0;
        for (int i=0; i<N; ++i)
            sum += x[i];
        mux=sum/N; }
    void averagey() {
        double sum=0.0;
        for (int i=0; i<N; ++i)
            sum += y[i];
        muy=sum/N; }
    void display() {
        cout << "Average x: " << mux << endl;
        cout << "Average y: " << muy << endl;
    }
};
```

---

## Inheritance MPI code

---

```
Average temp,heat;
```

```
if (rank==0) {  
    temp.getdata();  
    temp.averagex();  
    temp.averagey();  
    MPI_Send( &temp,160,MPI_BYTE,1,23,MPI_COMM_WORLD);  
}else{  
MPI_Recv( &heat,160,MPI_BYTE,0,23,MPI_COMM_WORLD,&sitrep);  
    heat.Data::display();  
    heat.display();  
}
```

# Inheritance Code Output

---

```
i=0 x=1 y=15.6 // Data::display() from P1
```

```
i=1 x=2 y=17.5
```

```
i=2 x=3 y=36.6
```

```
i=3 x=4 y=43.8
```

```
i=4 x=5 y=58.2
```

```
i=5 x=6 y=61.6
```

```
i=6 x=7 y=64.2
```

```
i=7 x=8 y=70.4
```

```
i=8 x=9 y=98.8
```

```
Average x: 5 // Average::Display() from P1
```

```
Average y: 51.8556
```

---

# Introduction to MPI Parallel I/O

---

Reading and writing data is now handled much like sending messages to the disk and receiving messages from storage.

Several key features of MPI-2 parallel I/O are:

- MPI-2 allows the user to perform parallel I/O similar to the way messages are sent from one process to another
- Not all implementations presently implement the full MPI-2 I/O
- MPI-2 supports both blocking and non-blocking I/O
- MPI-2 supports both collective and non-collective I/O

# MPI-2 File Structure

---

MPI-2 partitions data within files similar to the way that derived datatypes define data partitions within memory.

The concepts of blocksize, memory striding and offsets are fundamental to understanding the file structure.

MPI-2 I/O has some of the following characteristics:

- MPI datatypes are written and read
- Concepts to those used to define derived datatypes are used to define how data is partitioned in the file
- Sequential as well as random access to files are supported
- Each process has its own "view" of the file

# MPI-2 File Structure

---

A view defines the current set of data visible and accessible by a process from an open file. Each process has its own view of the file, defined by three quantities:

- A displacement - describing where in the file to start
- An etype - the type of data that is to be written or read
- A filetype - pattern of how the data is partitioned in the file

The pattern described by a filetype is repeated, beginning at the displacement, to define the view.

# MPI-2 File Structure

---

Views can be changed by the user during program execution.

The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI\_BYTE).

# Displacement

---

To better understand the file view, each component will be explained in more detail.

A file displacement is an absolute byte position relative to the beginning of a file.

The displacement defines the location where a view begins.

Any data prior to the position indicated by the displacement will not be accessible from that process.

# Etype

---

An etype ( elementary datatype) is the unit of data access and positioning.

It can be any MPI predefined or derived datatype.

Derived etypes can be constructed using any of the MPI datatype constructor routines

Data access is performed in etype units, reading or writing whole data items of type etype.

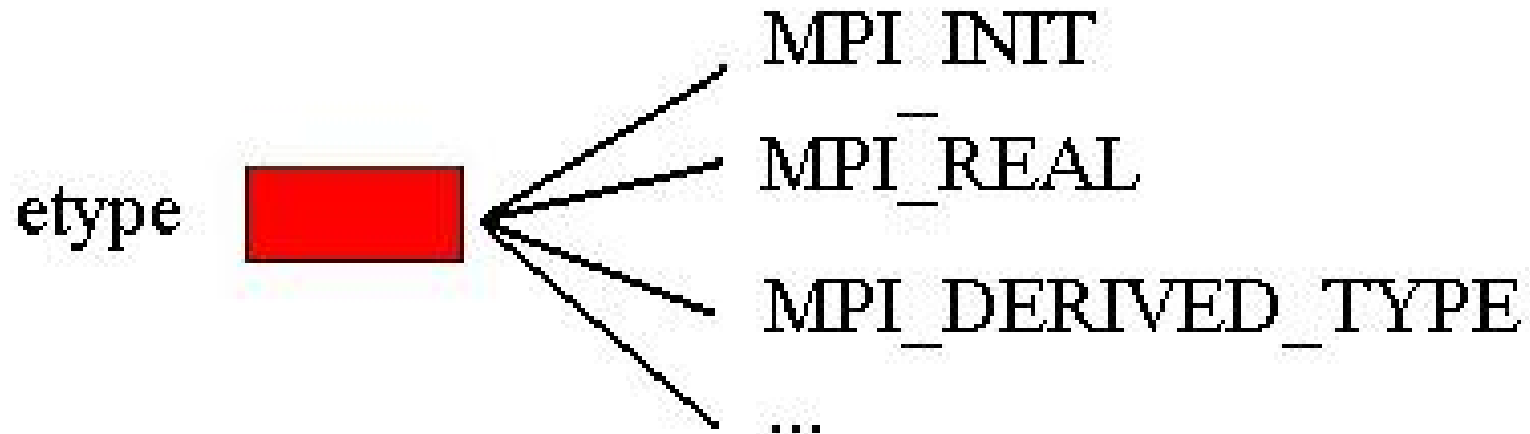
Offsets are expressed as a count of etypes

File pointers point to the beginning of etypes.

Is similar to the datatype argument in the MPI\_SEND call

# Etype

---



# Filetype

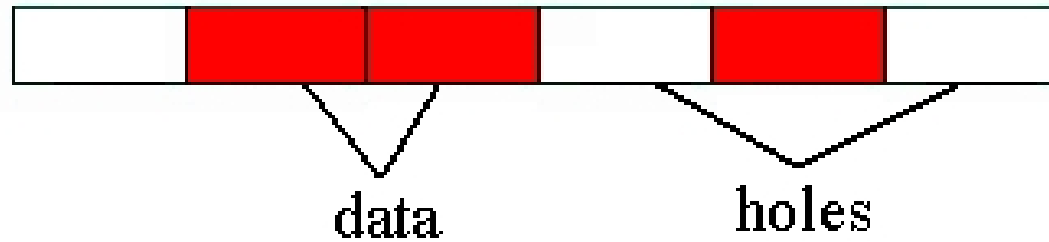
---

- A filetype is the basis for partitioning a file and defines a template for accessing the file.
- Is the basis for partitioning a file among processes
- Defines a template for accessing the file
- Is a defined sequence of etypes, which can have data or be considered blank
- Is similar to a vector derived datatype
- A filetype is repeated to fill the view of the file

# Filetype

---

filetype



# View

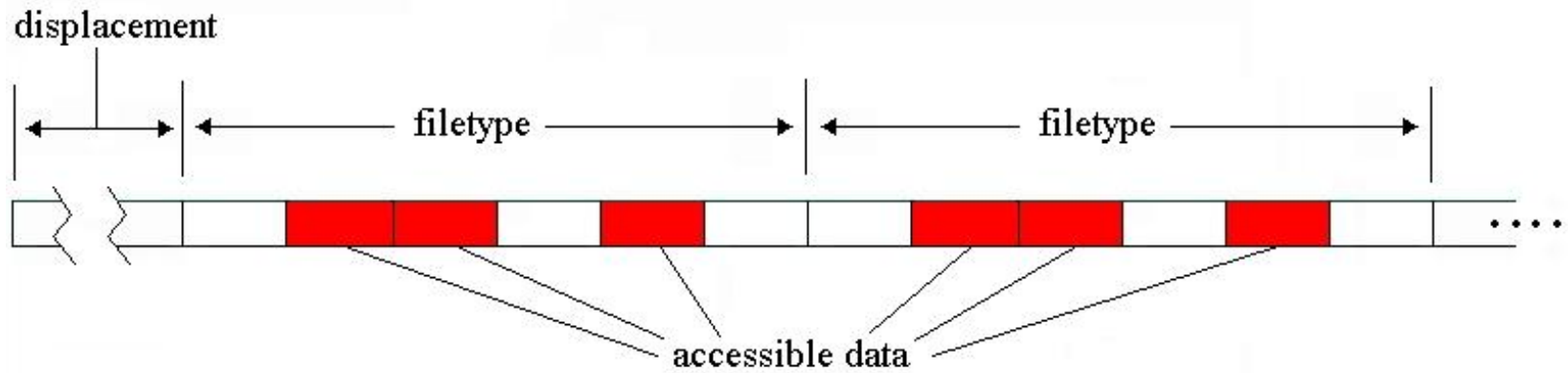
---

Each process has its own view of the shared file that defines what data it can access.

- A view defines the current set of data, visible and accessible
- Each process has its own view of the file
- View can be changed by the user during program execution
- Starting at the position defined by the displacement, the filetype is repeated to form the view
- Default view:
  - displacement = 0 ; etype = MPI\_BYTE ; filetype = MPI\_BYTE

# View

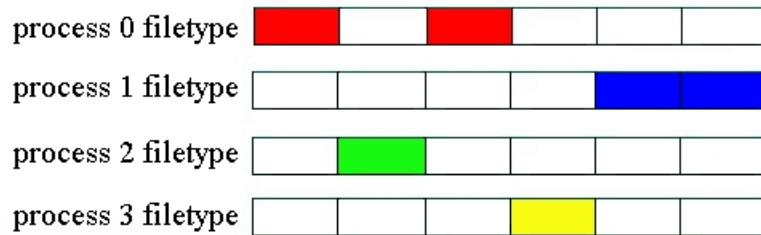
---



# Combining Views of Multiple Processes

---

A group of processes can use complementary views to achieve a global data distribution



# Initializing MPI File I/O

---

File I/O is initialized in MPI-2 with the `MPI_FILE_OPEN` call. This is a collective routine and all processes within the communicator specified must provide filenames that reference the same file.

C:

```
int MPI_File_open(MPI_Comm comm, char *filename, int
    amode, MPI_Info info, MPI_File *fh)
```

FORTRAN:

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH,
    IERROR)
```

```
CHARACTER*(*) FILENAME
```

```
INTEGER COMM, AMODE, INFO, FH, IERROR
```

# Initializing MPI File I/O

---

The arguments for this routine are:

COMM	process communicator
FILENAME	name of the file to open
AMODE	file access mode
INFO	information handle that varies with implementation
FH	new file handle by which the file can be accessed

# Initializing MPI File I/O

---

The file access mode is set from the list of following options:

- `MPI_MODE_RDONLY` - read only
- `MPI_MODE_RDWR` - reading and writing
- `MPI_MODE_WRONLY` - write only
- `MPI_MODE_CREATE` - create the file if it does not exist
- `MPI_MODE_EXCL` - error if creating file that already exists
- `MPI_MODE_DELETE_ON_CLOSE` - delete file on close
- `MPI_MODE_UNIQUE_OPEN` - file will not be concurrently opened elsewhere
- `MPI_MODE_SEQUENTIAL` - file will only be accessed sequentially
- `MPI_MODE_APPEND` - set initial position of all file pointers to end of file

# Initializing MPI File I/O

---

AMODE is a bit vector OR of all of the above features desired.

- C/C++ - users can use bit vector OR (|) to combine these constants
- Fortran 90 - users can use the bit vector IOR intrinsic
- Fortran 77 - users can use the bit vector IOR on system that support it, but it is not portable

# Defining A View

---

To define a view, the `MPI_FILE_SET_VIEW` function is used.

C:

```
int MPI_File_set_view(MPI_File fh, MPI_Offset
    disp, MPI_Datatype etype, MPI_Datatype
    filetype, char *datarep, MPI_Info info)
```

Fortran:

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE,
    DATAREP, INFO, IERROR)
INTEGER          FH, ETYPE, FILETYPE, INFO,
    IERROR
CHARACTER*(*)   DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

# Defining A View

---

The arguments for this routine are:

FH	file handle specified in MPI_FILE_OPEN
DISP	absolute displacement from the beginning of the file
ETYPE	unit of data access and positioning
FILETYPE	sequence of etypes which is repeated to define the view of a file
DATAREP	data representation (see explanation below)
INFO	information handle that varies with implementation

# Defining A View

---

- The DATAREP argument refers to the representation of the data within the file.
- MPI guarantees full inter operability within a single MPI environment, and supports increased inter operability outside that environment through the external data representation.

# Defining A View

---

While custom data representations can be defined, by default DATAREP can be one of the following values:

## **native**

- Data in this representation is stored in a file exactly as it is in memory.
- Advantages: data precision and I/O performance are not lost in type conversions with a purely homogeneous environment
- Disadvantages: the loss of transparent inter-operability within a heterogeneous MPI environment.

## **internal**

- Data in this representation can be used for I/O operations in a homogeneous or heterogeneous environment
- The implementation will perform type conversions if necessary

# Defining A View

---

## **external32**

- This data representation states that read and write operations convert all data from and to the "external32" representation
- The file can be exported from one MPI environment to another with the guarantee that the second environment will be able to read all the data
- Disadvantage: data precision and I/O performance may be lost in data type conversions

# Data Access - Reading Data

---

There are three aspects to data access:

- positioning
  - explicit offset ; individual file pointers ; shared file pointers
- synchronization
  - Blocking ; non-blocking ; split collective
- coordination
  - Collective ; non-collective

# MPI\_FILE\_READ

---

MPI\_FILE\_READ reads a file using individual file pointers.

C:

```
int MPI_File_read(MPI_File fh, void *buf, int
    count, MPI_Datatype datatype, MPI_Status
    *status)
```

Fortran:

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE,
    STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE,
    STATUS(MPI_STATUS_SIZE), IERROR
```

# MPI\_FILE\_READ

---

The arguments for this routine are:

FH	file handle
BUF	initial address of buffer
COUNT	number of elements in buffer
DATATYPE	datatype of each buffer element (handle)

# Data Access – Writing Data

---

- As an introduction to parallel data output, this section will only look at the blocking, non-collective calls. (As was done with the parallel input section).

# MPI\_FILE\_WRITE

---

MPI\_FILE\_WRITE writes a file using individual file pointers.

C:

```
int MPI_File_read(MPI_File fh, void *buf, int count,  
    MPI_Datatype datatype, MPI_Status *status)
```

FORTTRAN:

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS,  
    IERROR)
```

```
<type> BUF(*)
```

```
INTEGER FH, COUNT, DATATYPE,  
    STATUS(MPI_STATUS_SIZE), IERROR
```

# MPI\_FILE\_WRITE

---

The arguments for this routine are:

FH file handle

OFFSET file offset

BUF initial address of buffer

COUNT number of elements in buffer

DATATYPE datatype of each buffer element (handle)

STATUS status object (handle)

# Sample Program (Run with 4 MPI Processes)

---

```
integer, dimension(3) :: value=(/10,45,62/)
integer :: amode,OUT,etype,filetype,intsize,info=0
integer (KIND=MPI_OFFSET_KIND) :: disp
amode=ior(MPI_MODE_CREATE,MPI_MODE_WRONLY)
call MPI_FILE_OPEN(MPI_COMM_WORLD,'testout.dat',amode, &
                  info,OUT,err)
call MPI_TYPE_EXTENT(MPI_INTEGER,intsize,err)
disp=3*rank*intsize
etype=MPI_INTEGER
filetype=MPI_INTEGER
call MPI_FILE_SET_VIEW(OUT,disp,etype,filetype, &
                      'NATIVE', info,err)
do i=1,3
  go=(rank+1)*value(i)
  call MPI_FILE_WRITE(OUT,go,1,MPI_INTEGER,state,err)
end do
call MPI_FILE_CLOSE(OUT,err)
```

---

# Program Output

---

Since output file testout.dat is binary use the octal dump command (od) to examine its content

```
$ od -dx testout.dat
0000000  10    0   45    0   62    0   20
          000a 0000 002d 0000 003e 0000 0014 000
0000020  90    0  124    0   30    0  135
          005a 0000 007c 0000 001e 0000 0087 000
0000040  186   0   40    0  180    0  248
          00ba 0000 0028 0000 00b4 0000 00f8 000
000006
```