Introduction to MPI

Dan Stanzione
Arizona State University
July 14th, 2008

Introduction & Outline

- Overview of message passing
- MPI: what is it and why should you learn it?
- MPI references and documentation
- MPI API
  - Point to Point Communication
  - Collective communication and computation
- Compiling and running MPI programs
- Some more advanced concepts

Message Passing Overview

- What is message passing?
  - Literally, the sending and receiving of messages between tasks
  - Commonly used in distributed systems
  - Capabilities include sending data, performing operations on data, and synchronization between tasks
- Memory model
  - each process has its own address space, and no way to get at another's, so it is necessary to send/receive data

Alternatives to message passing

- OpenMP
- PGAS (partitioned global address space) languages
  - UPC
  - Co-Array Fortran
- Libraries
  - Global Arrays
  - ...
- Hybrids with Message Passing are possible

MPI-1

- MPI-1 - Message Passing Interface (v. 1.2)
  - library standard defined by committee of vendors, implementers, and parallel programmers
  - used to create parallel SPMD codes based on explicit message passing
- Available on almost all parallel machines with C/C++ and Fortran bindings (and occasionally with other bindings)
- About 125 routines, total
  - 6 basic routines
  - the rest include routines of increasing generality and specificity

MPI-2

- Includes features left out of MPI-1
  - one-sided communications
  - dynamic process control
  - more complicated collectives
- Implementations
  - not quickly undertaken after the standard document was released (in 1997)
  - now OpenMPI, MPICH2 (and its descendants), and the vendor implemenations are pretty complete or fully complete

Why learn MPI?

- MPI is a standard
  - Public domain version easily to install
  - Vendor-optimized version available on most communication hardware
- MPI applications can be fairly portable
- MPI is expressive: MPI can be used for many different models of computation, therefore can be used with many different applications.
- MPI is `assembly language of parallel processing': low level but efficient.

Basic MPI

- Initialization and Termination
- Setting up *Communicators*
- Point to Point Communication
- Collective Communication
- In principle enough for any application, but more complicated constructs can be more efficient

MPI Basics

- **Most used MPI commands**

```
MPI_Init - start using MPI
MPI_Comm_size - get the number of tasks
MPI_Comm_rank - the unique index of this task
MPI_Send - send a message
MPI_Recv - receive a message
MPI_Finalize - stop using MPI
```

- **All algorithms can be expressed in MPI with these 6 functions**
- **…but we don't recommend it!**

Initialization and Termination

- All processes must initialize and finalize MPI (each is a collective call).
- Must include header files

```
#include <mpi.h>
main(int argc char**&argv){
int ierr;
ierr = MPI_Init(&argc, &argv);
            :
ierr = MPI_Finalize();
}
```

```
program init_finalize
include 'mpif.h'
call mpi_init(ierr)
            :
call mpi_finalize(ierr)
end program
```

Setting up *Communicators* I

- Communicators
  - collections of processes that can communicate with each other
  - Multiple communicators can co-exist
- Each communicator answers two questions:
  - How many processes exist in this communicator?
  - Which process am I?
- MPI_COMM_WORLD encompasses all processes and is defined by default

Setting up *Communicators* I

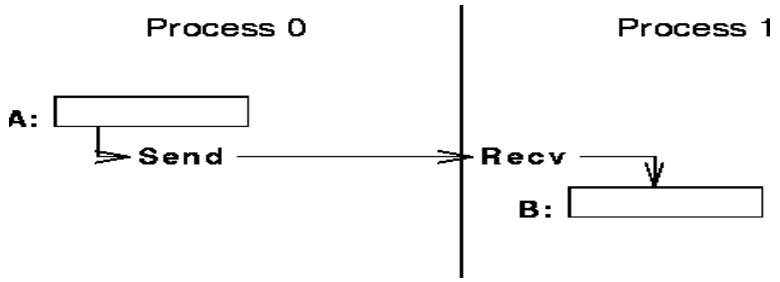
```
#include <mpi.h>
main(int argc, char **argv){
   int np,  mype,  ierr;

   ierr = MPI_Init(&argc, &argv);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);
   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &mype);
            :
   MPI_Finalize();
}
```

Setting up *Communicators* II

```
program param
  include 'mpif.h'

  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, np  ,ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, mype,ierr)
            :
  call mpi_finalize(ierr)
end program
```

Point to Point Communication I

- Sending data from one point (process/task) to another point (process/task)
- One task sends while another receives



Point to Point Communication II

- MPI_Send(): A blocking call which returns only when data has been copied to its buffer
- MPI_Recv(): A blocking receive which returns only when data has been received onto its buffer

```
mpi_send (data, count, type, dest, tag, comm, ierr)
mpi_recv (data, count, type, src,  tag, comm, status,
ierr)
```

```
MPI_Send (data, count, type, dest, tag, comm)
MPI_Recv (data, count, type, src,  tag, comm, &status)
```

Point to point code

- Recall that all tasks execute `the same' code
- Conditionals often needed

```
MPI_Comm_rank(comm,&mytid);

if (mytid==0) {
  MPI_Send( /* buffer */, /* target= */ 1, /* tag= */ 0,
            comm);
} else if (mytid==1) {
  MPI_Recv( /* buffer */, /* source= */ 0, /* tag= */ 0,
            comm);
}
```

Point to Point Communication III

- Common Parameters:
  - *void* data*: actual data being passed
  - *int count*: number of *type* values in *data*
  - *MPI_Datatype type*: data type of *data*
  - *int dest/src:* rank of the process this call is sending to or receiving from. *src* can also be wildcard MPI_ANY_SOURCE
  - *int tag*: simple identifier that must match between sender/receiver, or the wildcard MPI_ANY_TAG
  - *MPI_Comm comm*: communicator that must match between sender/receiver – no wildcards
  - *MPI_Status* status*: returns information on the message received (receiving only), e.g. which source if using MPI_ANY_SOURCE
  - *INTEGER ierr*: place to store error code (Fortran only. In C/C++ this is the return value of the function call)

```
mpi_send (data, count, type, dest, tag, comm, ierr)
mpi_recv (data, count, type, src,  tag, comm, status, ierr)
```

Point to Point Communication IV

```
#include  "mpi.h"
main(int argc, char **argv){
int ipe, ierr; double a[2];
MPI_Status status;
MPI_Comm icomm = MPI_COMM_WORLD;
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_rank(icomm, &ipe);
ierr = MPI_Comm_size(icomm, &myworld);
if(ipe == 0){
    a[0] = mype; a[1] = mype+1;
    ierr = MPI_Send(a,2,MPI_DOUBLE, 1,9, icomm);
  }
else if (ipe == 1){
    ierr = MPI_Recv(a,2,MPI_DOUBLE, 0,9,icomm,&status);
    printf("PE %d, A array= %f %f\n",mype,a[0],a[1]);
  }
MPI_Finalize();
}
```

Point to Point Communication V

```
program sr
   include "mpif.h"
   real*8,  dimension(2)                :: A
   integer, dimension(MPI_STATUS_SIZE) :: istat
   icomm = MPI_COMM_WORLD
   call mpi_init(ierr)
   call mpi_comm_rank(icomm,mype,ierr)
   call mpi_comm_size(icomm,np  ,ierr);

   if(mype.eq.0) then
    a(1) = real(ipe); a(2) = real(ipe+1)
    call mpi_send(A,2,MPI_REAL8, 1,9,icomm, ierr)
   else if (mype.eq.1) then
    call mpi_recv(A,2,MPI_REAL8, 0,9,icomm, istat,ierr)
    print*,"PE ",mype,"received A array =",A
   endif

   call mpi_finalize(ierr)
end program
```

Broadcast

- What if one processor wants to send to everyone else?

```
if (mytid == 0 ) {
  for (tid=1; tid<ntids; tid++) {
    MPI_Send( (void*)a, /* target= */ tid, … );
  }
} else {
  MPI_Recv( (void*)a, 0, … );
}
```

- Implements a very naive, serial broadcast
- Too primitive
  - leaves no room for the OS / switch to optimize
  - leaves no room for fancy algorithms
- Too slow: calls may wait for completion.

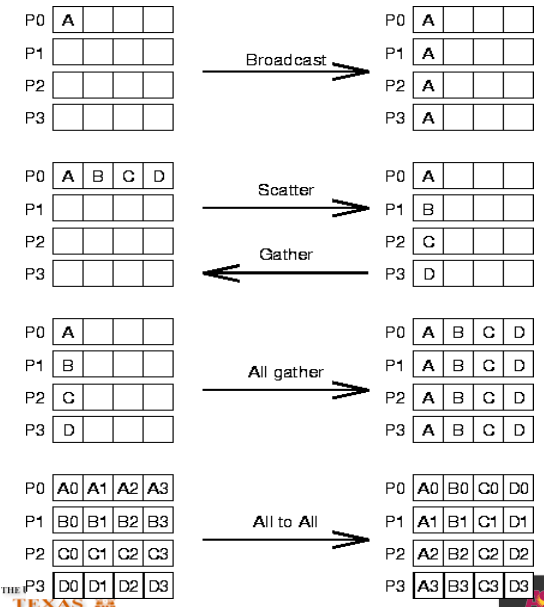
MPI Collective Communications

- Collective Communication
  - communication pattern that involves all processes within a communicator
- There are three basic types of collective communications:
  - Synchronization
  - Data movement
  - Computation w/ movement
- MPI Collectives are all blocking
- MPI Collectives do not use message tags

Synchronization

- Barrier
  - `mpi_barrier(comm, ierr)`
  - `MPI_Barrier(comm)`
- Function blocks until all tasks in *comm* call it.

Collective Data Movements

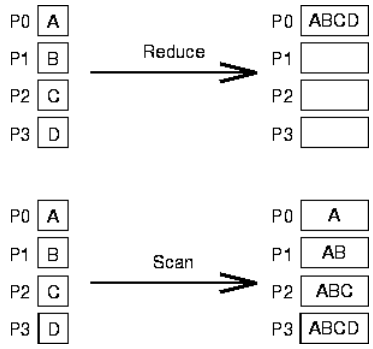


Broadcast

- `mpi_bcast(data, count, type, root, comm, ierr)`
- `MPI_Bcast(data, count, type, root, comm)`

- Sends **`data`** from the *root* process to all communicator members.

Collective Computation Patterns I



Collective Computation Patterns II

```
mpi_reduce(data, result, count, type, op, root, comm, ierr)
MPI_Reduce(data, result, count, type, op, root, comm)
```

Parameter Name	Operation
MPI_SUM	**sum**
MPI_PROD	**product**
MPI_MAX	**maximum value**
MPI_MIN	**minimum value**
MPI_MAXLOC	**max. value location & value**
MPI_MINLOC	**min. value location & value**

Collective Computation Example I

```
#include <mpi.h>
#define WCOMM MPI_COMM_WORLD
main(int argc, char **argv){
 int npes, mype, ierr;
 double sum, val; int calc, count=1;
 ierr = MPI_Init(&argc, &argv);
 ierr = MPI_Comm_size(WCOMM, &npes);
 ierr = MPI_Comm_rank(WCOMM, &mype);

 val = (double) mype;

ierr=MPI_Allreduce(&val,&sum,count,MPI_DOUBLE,MPI_SUM,WCOMM
);

 calc=(npes-1 +npes%2)*(npes/2);
 printf(" PE: %d sum=%5.0f calc=%d\n",mype,sum,calc);
 ierr = MPI_Finalize();
}
```

Collective Computation Example II

```
program sum2all
include 'mpif.h'

  icomm = MPI_COMM_WORLD
  count = 1
  call mpi_init(ierr)
  call mpi_comm_rank(icomm,mype,ierr)
  call mpi_comm_size(icomm,npes,ierr)
  val = dble(mype)

  call mpi_allreduce(val,sum,count,MPI_REAL8,MPI_SUM,
       icomm,ierr)

  ncalc=(npes-1 + mod(npes,2))*(npes/2)
  print*,' pe#, sum, calc. sum = ',mype,sum,ncalc
  call mpi_finalize(ierr)
end
```

Compiling MPI Programs

- Generally use a special compiler or compiler wrapper script
  - not defined by the standard
  - consult your implementation
  - handles correct include path, library path, and libraries
- MPICH-style (the most common)
  - C

    ```
    mpicc -o foo foo.c
    ```
  - Fortran

    ```
    mpif77 -o foo foo.f
    ```

Running MPI Programs

- MPI programs require some help to get started
  - what computers should I run on?
  - how do I access them?
- MPICH-style

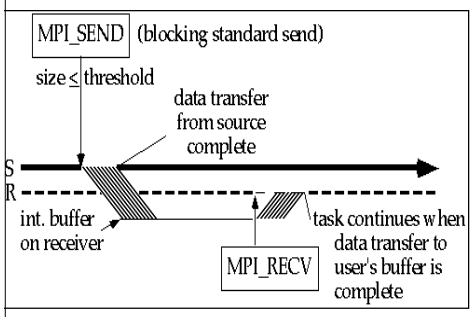
  ```
  mpirun -np 10 -machinefile mach ./foo
  ```
- When batch schedulers are involved, all bets are off
- Lonestar/Ranger (as part of an LSF/SGE script)

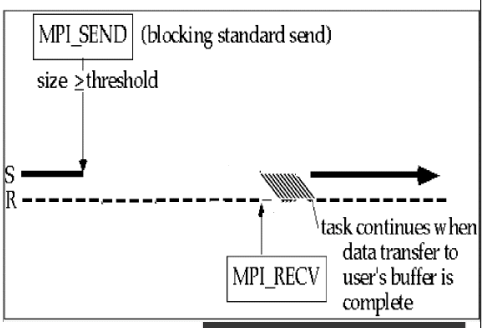
  ```
  ibrun ./foo
  ```
  - the schedulers handle the rest

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





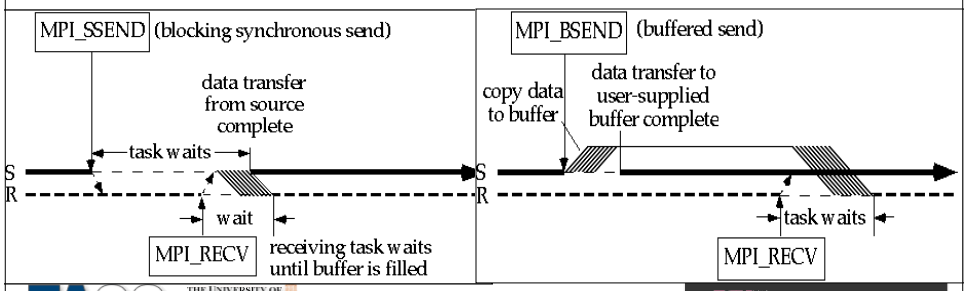
Why Care about Blocking?

- Actual user code
- Why did this code work on one machine, but not in general?
- We'll come back to this later…

```
!   SEND 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 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 sometimes with data
    END DO
```

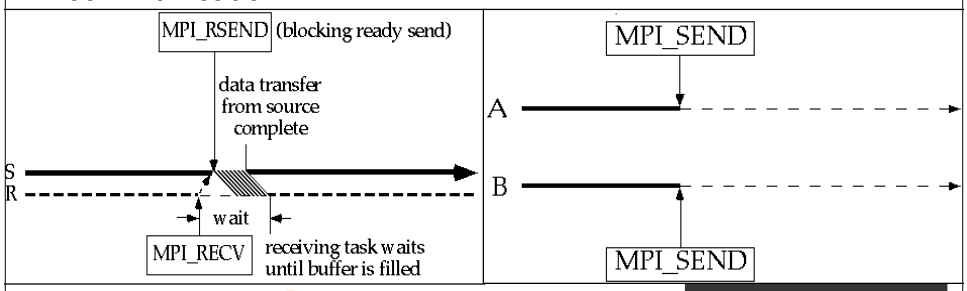
Point to Point Comm. II

- 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.

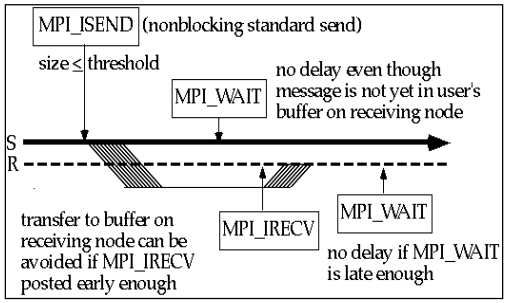


Point to Point Comm. IV

- 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 and buffer space can run out
- 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 V: 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 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 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 sometimes 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/recvs

```
      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)
      do p=1,nproc-1
C now process the incoming data
```

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

Point to Point Comm. VI

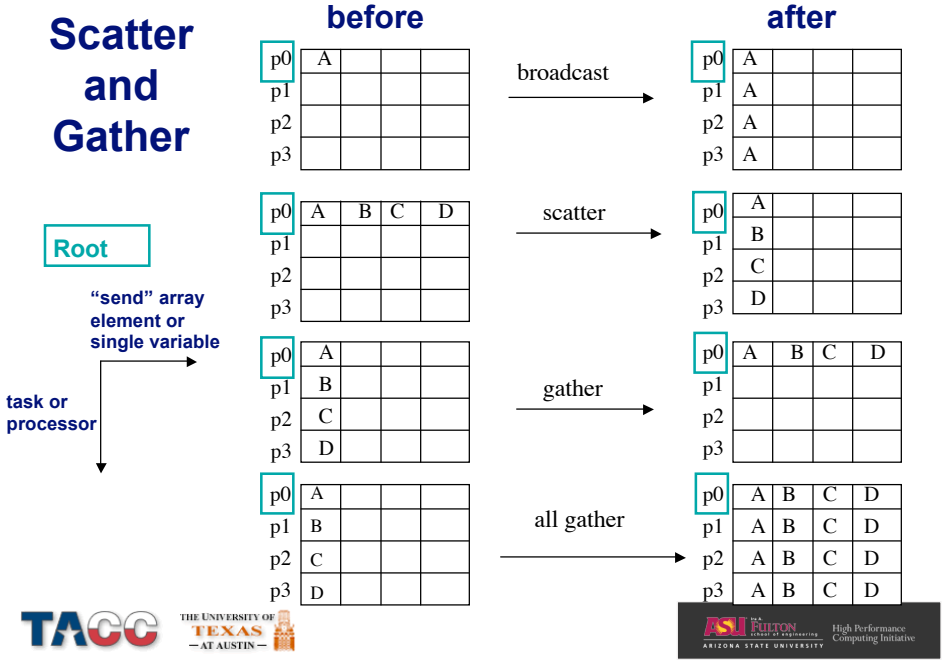
- MPI_Sendrecv : both in on call, source and destination can be the same
- Example 1: exchanging data with one other node; target and source the same
- Example 2: chain of processors
  - Operate on data
  - Send result to next processor, and receive next input from previous processor in line

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

More Collective Communications

Advanced Collective Comm. I

Scatter and Gather

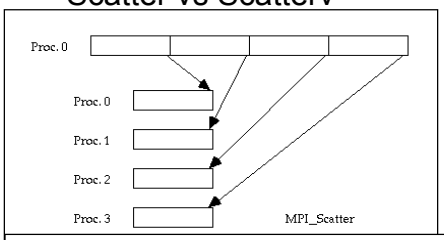


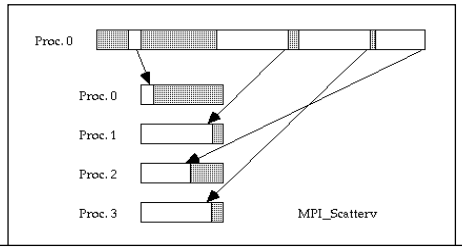
Advanced Collective Comm. II

- MPI_{Scatter,Gather,Allscatter,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
  - Somewhat harder to program (more bookkeeping)

Advanced Collective Comms. III

- 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(local_array,n,MPI_INT,result_array,
               sizes,offsets,MPI_INT,comm);
free(sizes); free(offsets);
```

Derived datatypes

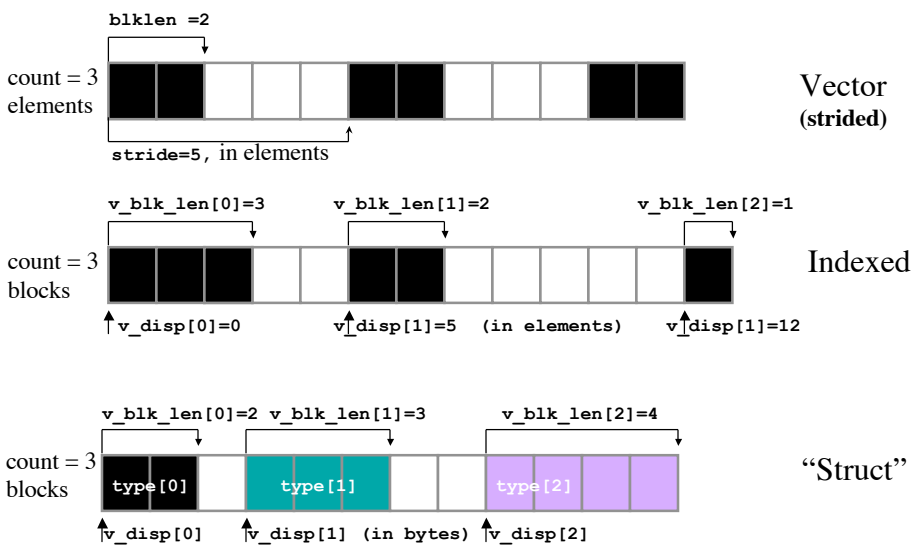
Derived Datatypes I

- 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 Data-types can be created recursively and at run-time
- 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)

nrows

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

ncols

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

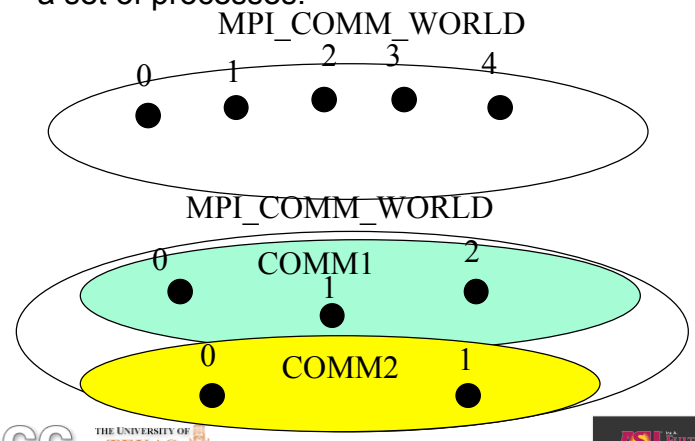
call MPI_Type_commit(vtype, ierr)

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

Communicators and Groups

Communicators and Groups I

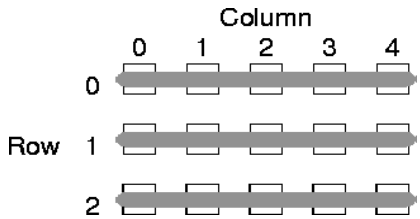
- All MPI communication is relative to a *communicator* which contains a *context* and a *group*. The group is just a set of processes.



Communicators and Groups II

- To subdivide communicators into multiple non-overlapping communicators – Approach I
  - e.g. to form groups of rows of PEs

```
                    :
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

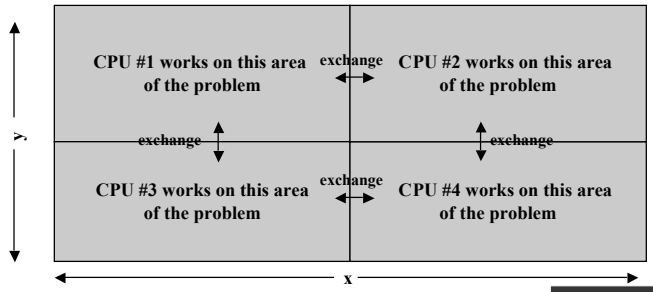
```
                                  :
MPI_Group base_grp, grp;  MPI_Comm row_comm, temp_comm;
int row_list[NCOL], irow, myrank_in_world;

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

Cartesian Topologies

- **Another use of Communicators is to organize your tasks in a more convenient way.**
- **Cartesian Communicators allow you to lay out your MPI tasks on a cartesian coordinate grid.**
- **Useful in any problem simulating a 2D or 3D grid.**



Cartesian Topologies

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

MPI::Cartcomm MPI::Intracomm::CreateCart(...)

MPI_CART_CREAT(...)

comm_old - input communicator

ndims - # of dimensions in cartesian grid

dims - integer array of size ndims specifying the number of processes in each dimension

periods - true/false specifying whether each dimension is periodic

reorder - ranks may be reordered or not

comm_cart - new communicator containing new topology.

Cartesian Shift function

```
int MPI_Cart_Shift(MPI_Comm comm, int
  direction, int disp, int *rank_source, int
  *rank_dest)
```

- direction - coordinate dimension of shift
- disp - displacement (can be positive or negative)
- rank_source and rank_dest are return values
  - use that source and dest to call MPI_Sendrecv

For More Information

- Using MPI by Gropp, Lusk, and Skjellum
- Using MPI-2 by Gropp, Lusk, and Thakur
- MPI-1 Standard
  - http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
- MPI-2 Standard
  - http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm