CS395T: Introduction to Scientific and Technical Computing

MPI

Instructors

Dr. Victor Eijkhout, Research Scientist, TACC
Dr. Karl W. Schulz, Research Associate, TACC

Parallel Processing

- Different models:
  - Producer/consumer: one task produces data, another consumes (processes) it
  - Master/slave: slaves work independently and the master assembles the result; for instance processing of individual pixels in graphics image. Other example: SETI@Home; email used as `communication library'.
  - SPMD: Single Program Multiple Data: every process runs the same code, but on different data; there will be frequent communication to coordinate work or exchange data. Communication through message passing.

Message Passing Overview

- What is message passing?
  - Commonly used in distributed systems: Lonestar, or workstations through ethernet
  - Literally, the sending and receiving of messages between tasks
  - 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.
- System takes care of sockets, buffering, data copying, et cetera.

Alternatives to message passing

- Message passing relies on explicit user action
- OpenMP and PGAS (parallel global address space) languages
- Language extensions (UPC, Global Arrays) for data that pretends to be globally addressable.
- OS-based approaches
- ==> hybrids with Message Passing are possible.

What is MPI? -- I

- MPI: Message Passing Interface
  - An agreed upon library specification: a standard
  - Not a language, not an implementation
  - Uses the message passing model
- Commonly supported on SMPs, clusters, and other heterogeneous memory and networked computers.

What is MPI? -- II

- Can be used from basic (6 functions) functionality to advanced and complex models (125 functions)
- Designed to provide access to advanced parallel hardware
  - end users: application scientists
  - library writers & parallel tool developers: higher level constructs that are based on MPI

Why learn MPI?

- MPI is a standard
  - public domain versions easy to install
  - vendor-optimized versions available on most communication hardware and architectures
- Therefore MPI applications are 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 a good way to learn about parallel computing
- MPI is "assembly language of parallel processing": low level but efficient

MPI References and Documentation

- Web
  - http://www.mcs.anl.gov/mpi/ (other mirror sites)
  - http://www.mpi-forum.org/
- Freely Available Implementations
  - http://www.mcs.anl.gov/mpi/mpich
  - http://www.lam-mpi.org/
- Books
  - *Using MPI*, by Gropp, Lusk, and Skjellum
  - *MPI Annotated Reference Manual,* by Marc Snir, *et al*
  - *Parallel Programming with MPI*, by Peter Pacheco
  - *Using MPI-2*, by Gropp, Lusk and Thakur
- Newsgroup
  - comp.parallel.mpi

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

Initialization and Termination

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

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

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

Setting up *Communicators* I

- Communicators define collections of processes that are allowed to communicate with each other
  - multiple communicators can co-exist
  - every function that communicates takes a communicator as an argument
- `MPI_COMM_WORLD` is the default communicator, encompassing all processes.
- Each communicator answers two fundamental questions
  - how many processes exist in this communicator?
  - which process am I?

Setting up *Communicators*—C

```
#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();
}
```

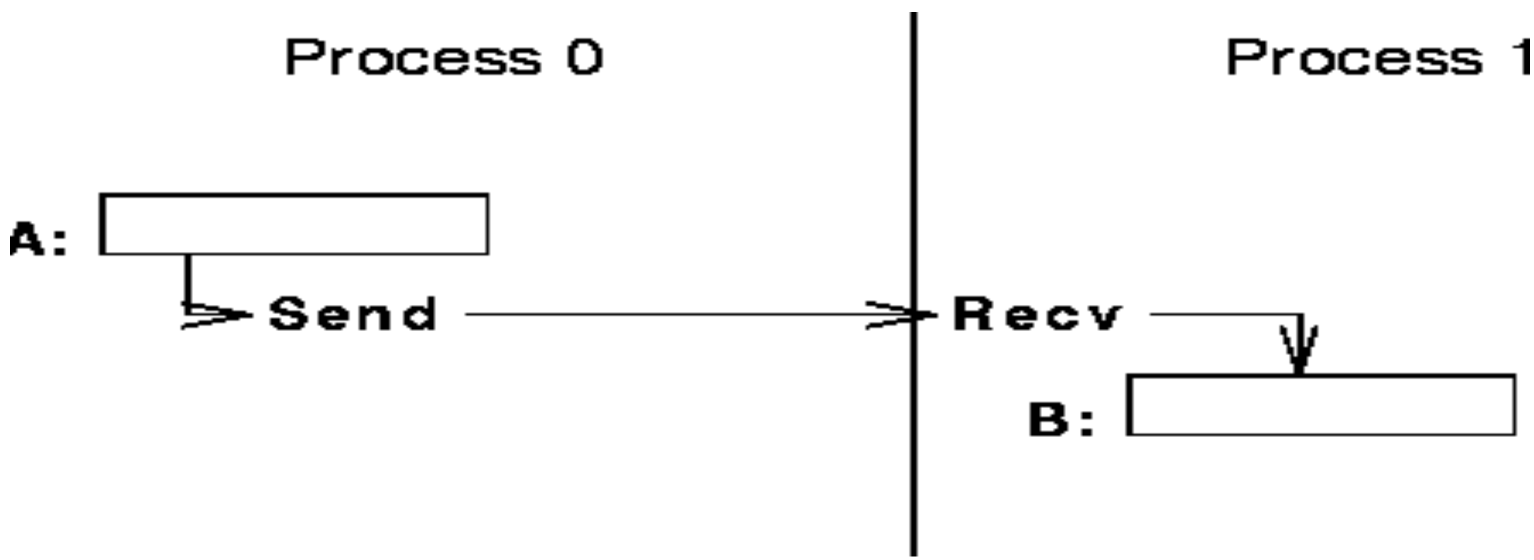
Setting up *Communicators*—FORTRAN

```
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

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



Point to Point Communication

- `MPI_Send()`: A blocking call which returns only when data has been sent from its buffer (often: when the corresponding receive call has finished)
- `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
- Lots of conditionals needed…..

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

Point to Point Communication

- 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
  - `int ierr`: place to store error code (Fortran only. In C/C++ this is the return value of the function call)
  - `MPI_Status* status`: returns information on the message received (receiving only), e.g. which source if using `MPI_ANY_SOURCE`

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

TACC

Point to Point Communication

```
#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

```
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
```

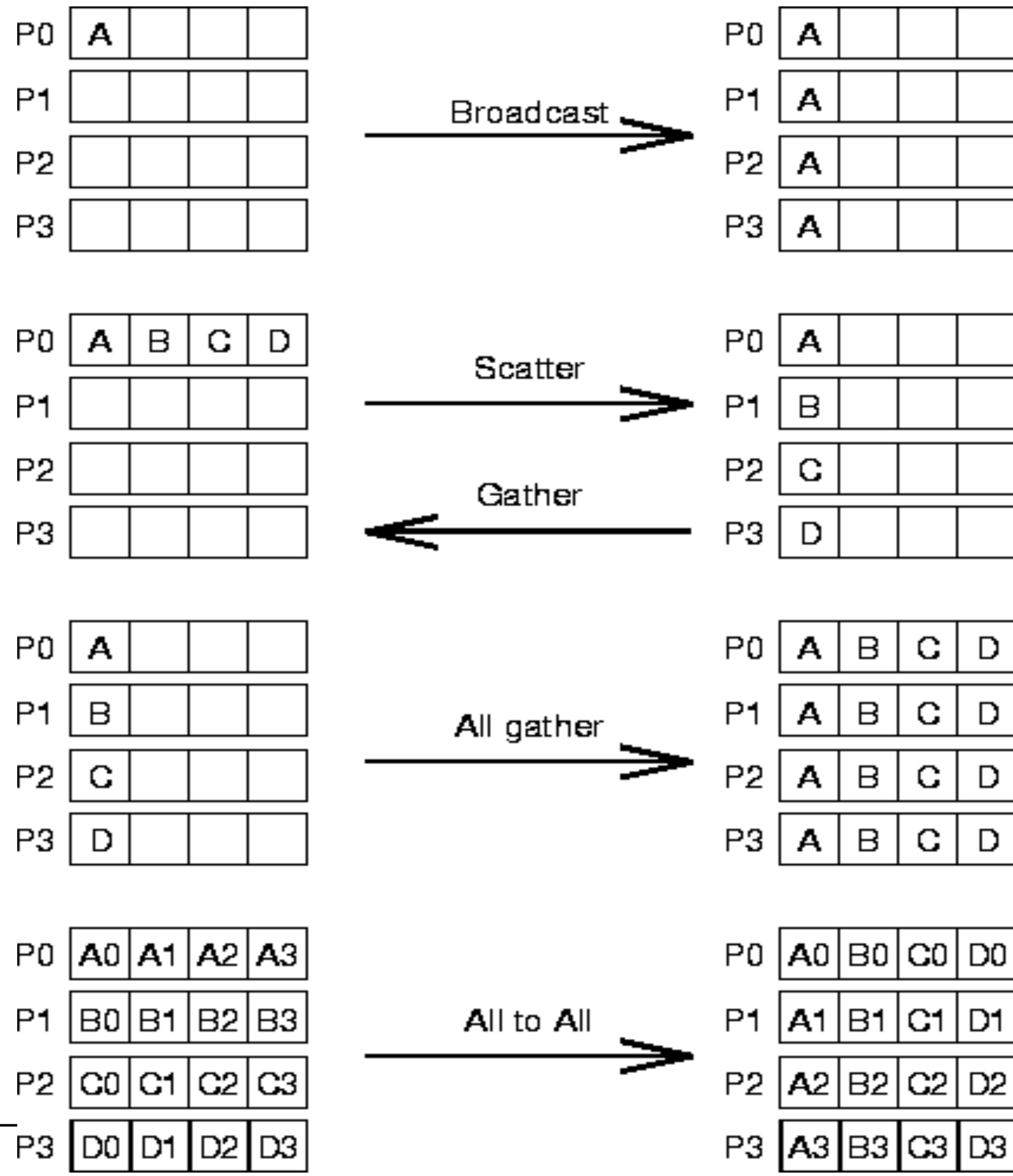
Synchronization

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

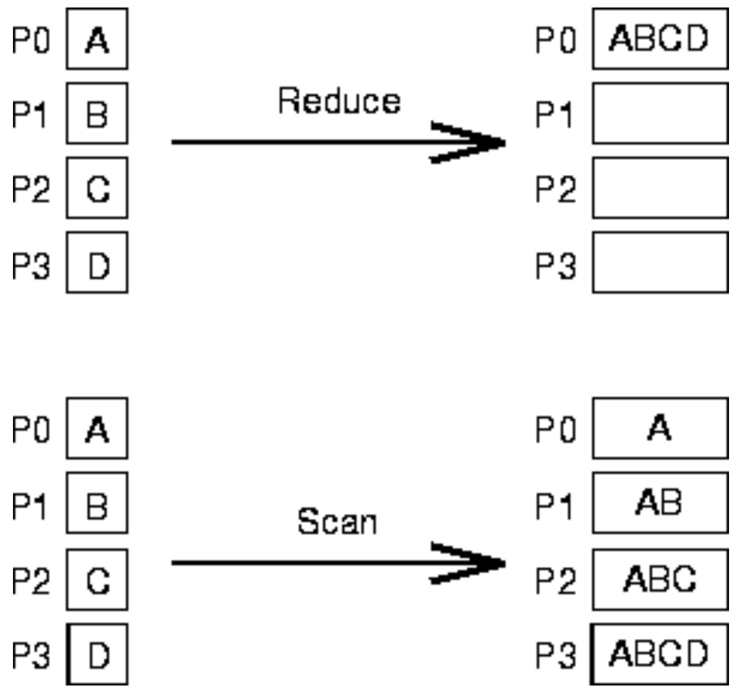
Collective Communications

- Collective Communication: a communication pattern that involves all processes within a communicator
- Collective communication calls are all blocking
- Collective communications do not use message tags
- There are three types of collective communications:
  - Synchronization
  - Data movements
  - Computations

Collective Data Movements



Collective Computation Patterns



Broadcast: the naïve approach

- You have all the tools….

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

- Broadcast data from 0 to all other processors
- Too primitive: leaves no room for the OS, MPI stack, or network hardware to optimize
- Too slow: calls may wait for completion

Broadcast

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

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

Reduce

```
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

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

```
program sum2all
include 'mpif.h'
  icomm = MPI_COMM_WORLD
  cnt = 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,cnt,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

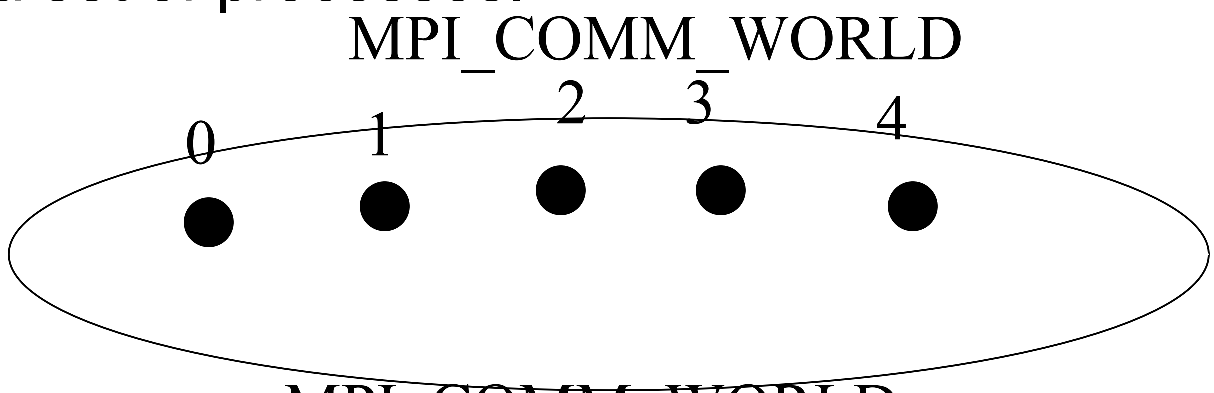
- Building simple MPI programs, using MPICH
  - `mpicc –o first first.c`
  - `mpif77 –o firstf firstf.f`
- Some MPI specific compiler options
  - **-mpilog** -- Generate log files of MPI calls
  - **-mpitrace** -- Trace execution of MPI calls
  - **-mpianim** -- Real-time animation of MPI (not available on all systems)
- **Note**: compiler/linker names are specific to MPICH. On IBM Power systems, they are *mpcc* and *mpxlf* respectively.

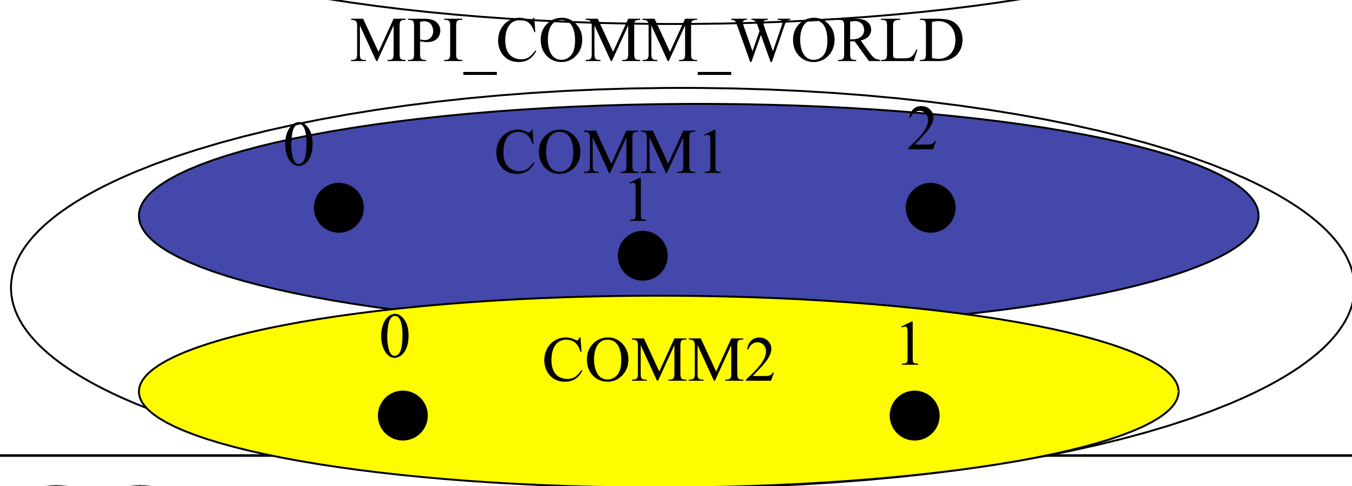
Running MPI Programs

- To run a simple MPI program using MPICH
  - mpirun/mpiexec –np 2 first
- Some MPI specific running options
  - -t -- shows the commands that mpirun would execute
  - -help -- shows all options for *mpirun*
- Note: *mpirun* is not part of the standard, but a similar command is common with several MPI implementations. On IBM, *poe;* on Lonestar *ibrun;* read the appropriate manual.

Communicators and Groups

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

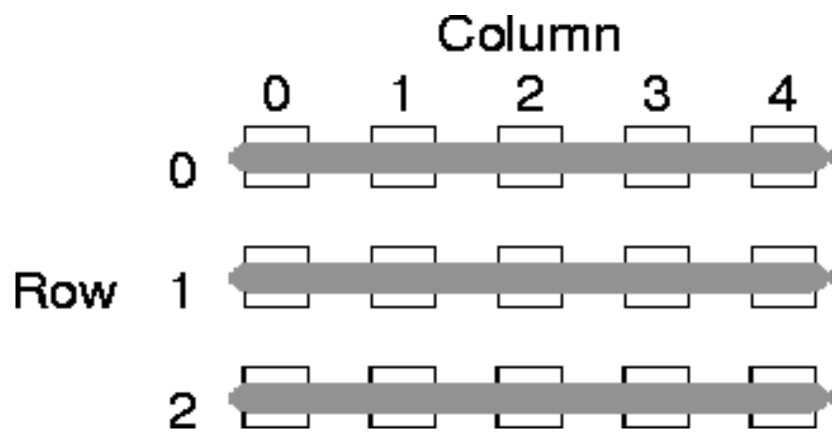




TACC

Communicators and Groups

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



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

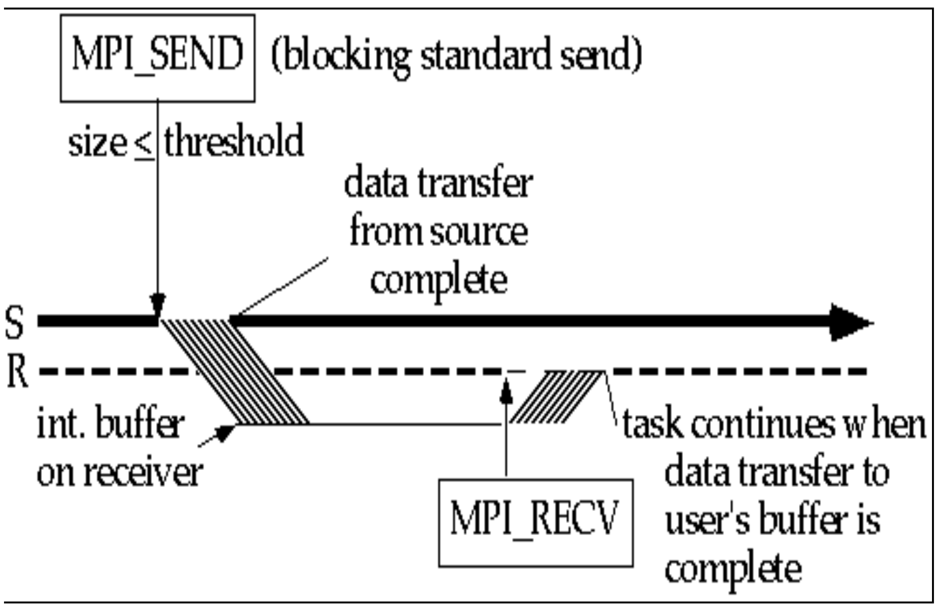
```
MPI_Comm row_comm;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
myrow = int(rank/ncol);
MPI_Comm_split(MPI_COMM_WORLD,myrow,rank,&row_comm);
```

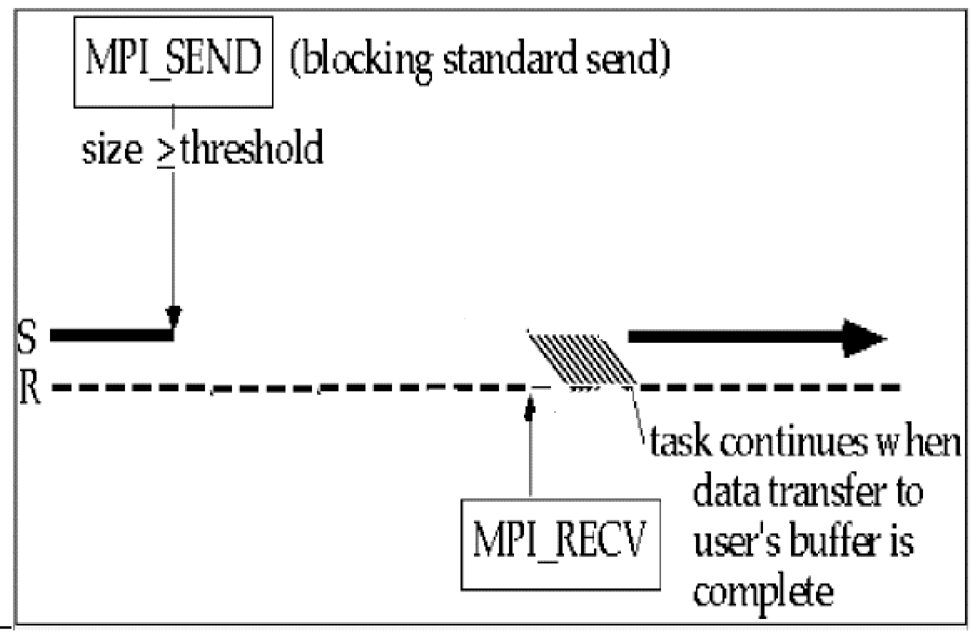
Communicators and Groups

- There is a more general mechanism using groups
- MPI_Comm_group: extract group from communicator
- Create new groups
- MPI_Comm_create: communicator from group
- Group commands: union, difference, intersection, range in/exclude

Point to Point Comm Details

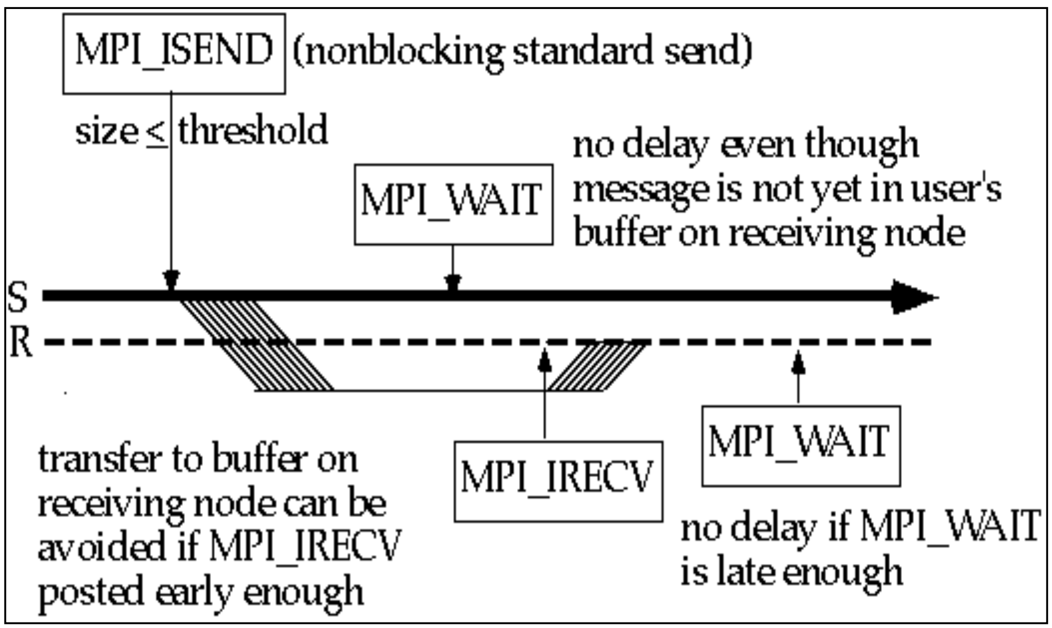
- Blocking send/receive
- MPI_Send, does not return until buffer is safe to reuse: either when buffered, or when actually received. (implementation / runtime dependent)
- Rule of thumb: send completes only if receive is posted/executed





Point to Point: Non-blocking

- Nonblocking communication: calls return, system handles buffering
- `MPI_Isend`
  - user must check status before using for same send (tag/receiver)
  - buffer can be reused for different tag/receiver (but don't modify it!)
- `MPI_Irecv`
  - user must check for completion



Non-blocking example

- Blocking operations can lead to deadlock
- Actual user code:
- Problem: all sends are waiting for corresponding receive: nothing happens

```
C   SEND THE DATA
       LM=6*NES+2
       DO 2 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
     2 CONTINUE

C   RECEIVE THE DATA
       LM=6*100+2
       DO 4 I=2,NUMPRC
         CALL MPI_RECV(NWS,LM,MPI_INTEGER,
      &    MPI_ANY_SOURCE,MSGTAG,MPI_COMM_WORLD,IERR)
C  do something with data
4      continue
```

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
      call mpi_waitall(nproc-1,sendreq,sendstat)
```

Note: this requires multiple send buffers

Solution using non-blocking send/recvs

```
      real*8 sendbuf(d,np-1), recvbuf(d,np-1)
      MPI_Request sendreq(np-1),recvreq(np-1)
      MPI_Status sendstat(np-1),recvstat(np-1)
      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)
      call mpi_waitall(nproc-1,recvreq,recvstat)
      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.

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 ….
MPI_Waitall( <make sure all sends finish> )
```

References

- Using MPI by Gropp, Lusk and Skjellum
- Using MPI-2 by Gropp, Lusk and Thakur
- www.nersc.gov/vendor_docs/ibm/pe
- http://www.llnl.gov/asci/purple/benchmarks/limited/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)