PITTSBURGH SUPERCOMPUTING CENTER

Practical MPI

Shawn T. Brown, PhD.

Director of Public Health Applications

Pittsburgh Supercomputing Center, Carnegie Mellon University

Compiling MPI programs

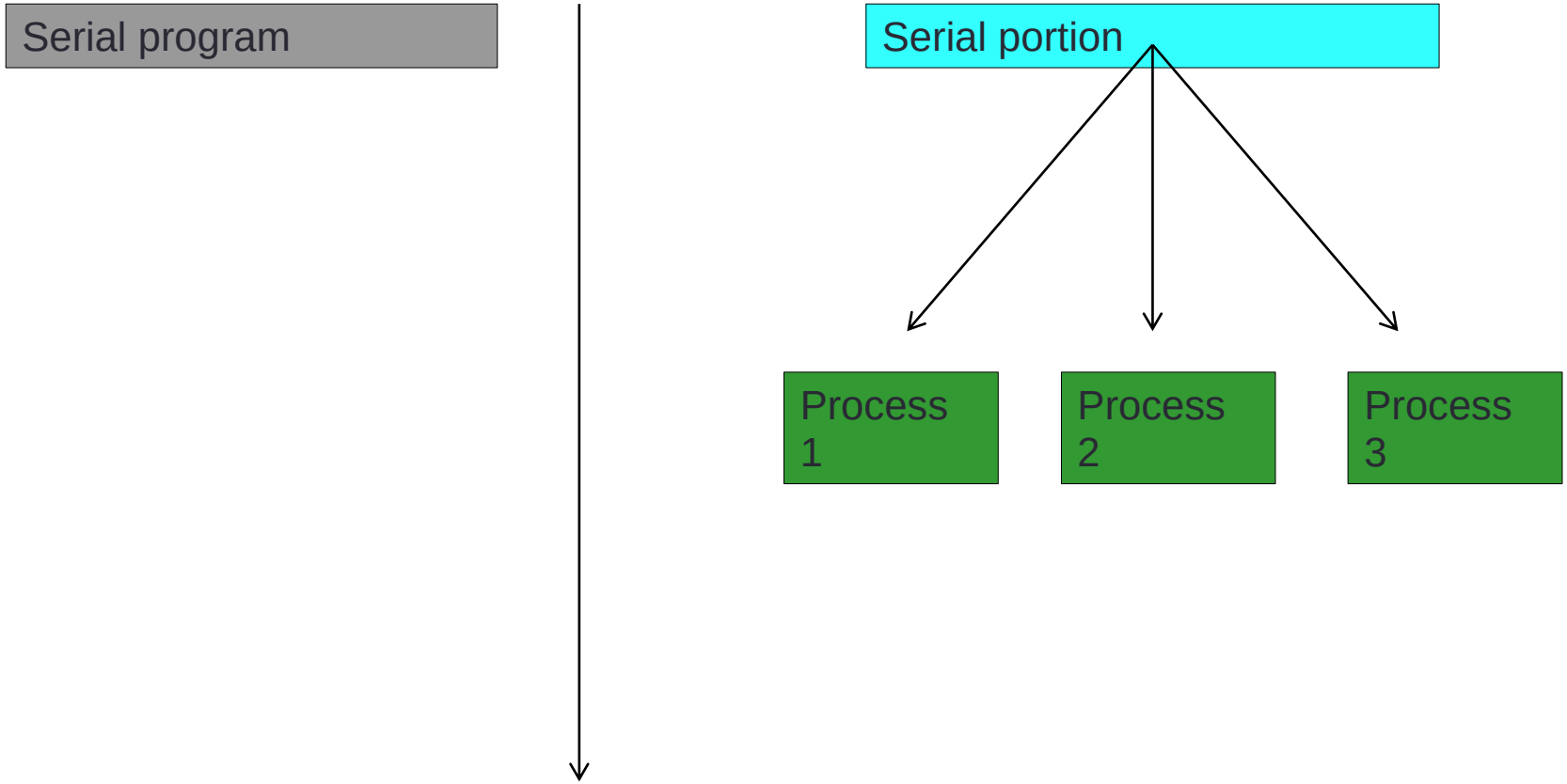
- The MPI-1 standard gives no requirement on how to compile an MPI program.
  - Most implementations use a compiler wrapper to compile
  - For MPICH, one could...
    - gcc -I/usr/local/mpich/include -L/usr/local/mpich/lib -lmpich prog.c -o prog
  - Or
    - mpicc prog.c -o prog
  - For Fortran: mpif77 or mpif90
  - For C++: mpicxx
  - Be careful, called something different on various architectures
    - Mpcc, mpiCC, CC, ftn, etc.

Running MPI programs

- Once again, there is no definition as to how one is to run an MPI program.
- Most implementations use a launcher to do this
  - mpirun is the most common mechanism
  - Syntax
    - mpirun -np <number of processors> <executable name>
  - Other options may include list of machines or other architecture dependend parameters, check the documentation.

Programming in MPI

- Basic Idea

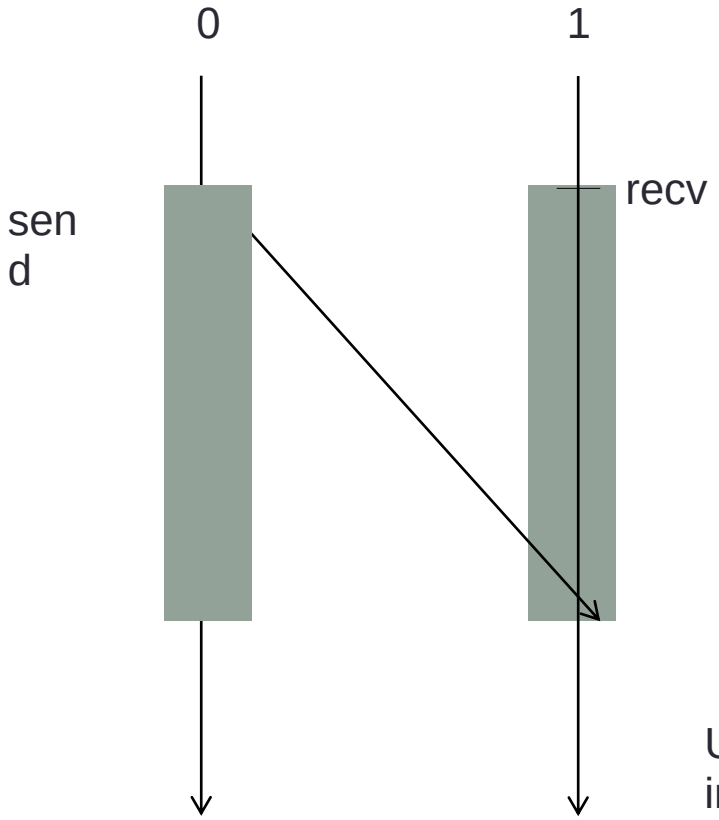


Programming in MPI

- Most MPI implementations
  - Provide a library to link into your code
  - C/C++ and FORTRAN interfaces
- Basically four types of MPI functions
  - Initialization and managment calls
  - Point-to-Point communication
    - Communication between 2 processes
  - Collective operations
    - Communication between groups of processes
  - Data type creation

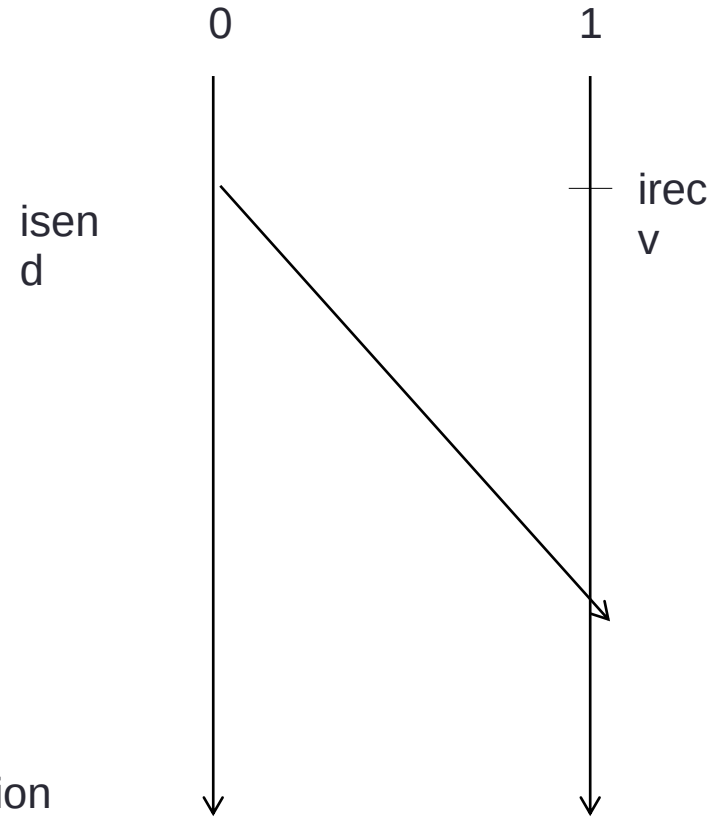
Point-to-Point communication

- In MPI standard, point-to-point communication is facilitated by sends and recieves



Blockin
g

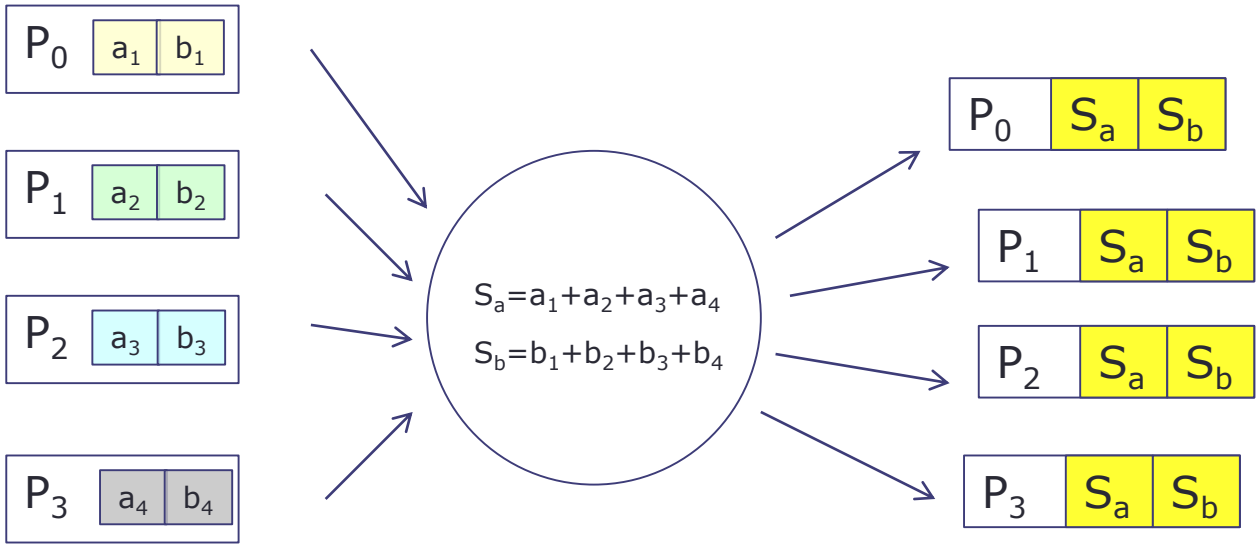
Up to implementation how to implement these concepts



Non-
Blocking

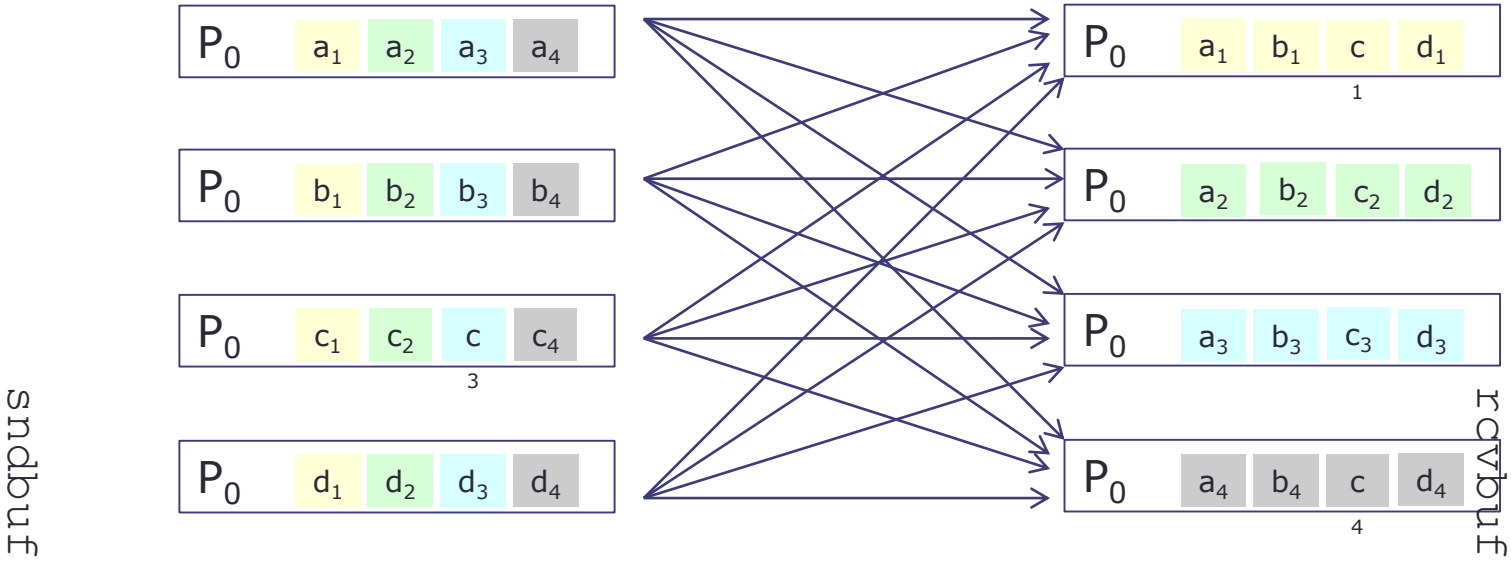
Collective communication

- Reduction
  - Reduce disparate arrays on a group of processors by some defined operations
    - e.g. summation



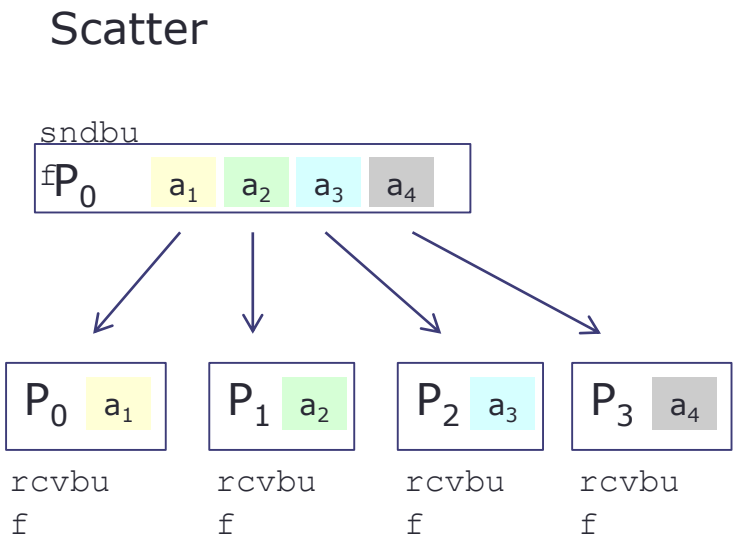
Collective communications

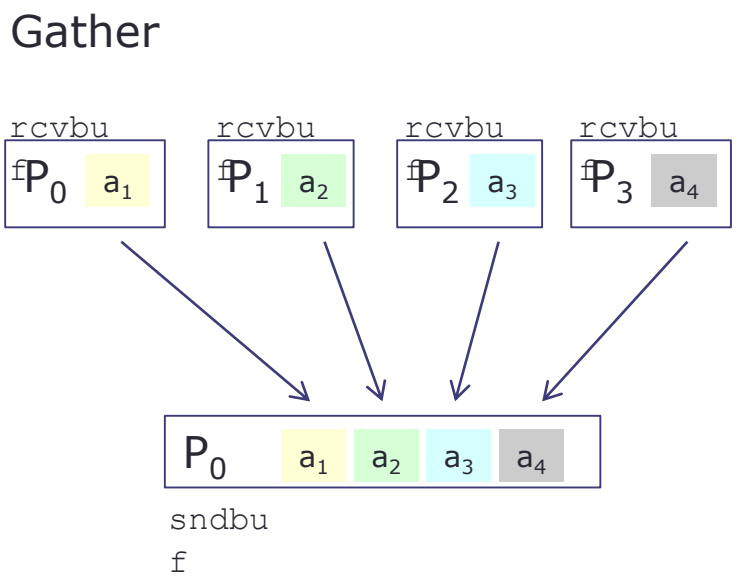
- All To All operations
  - Broadcasting



Collective Communications

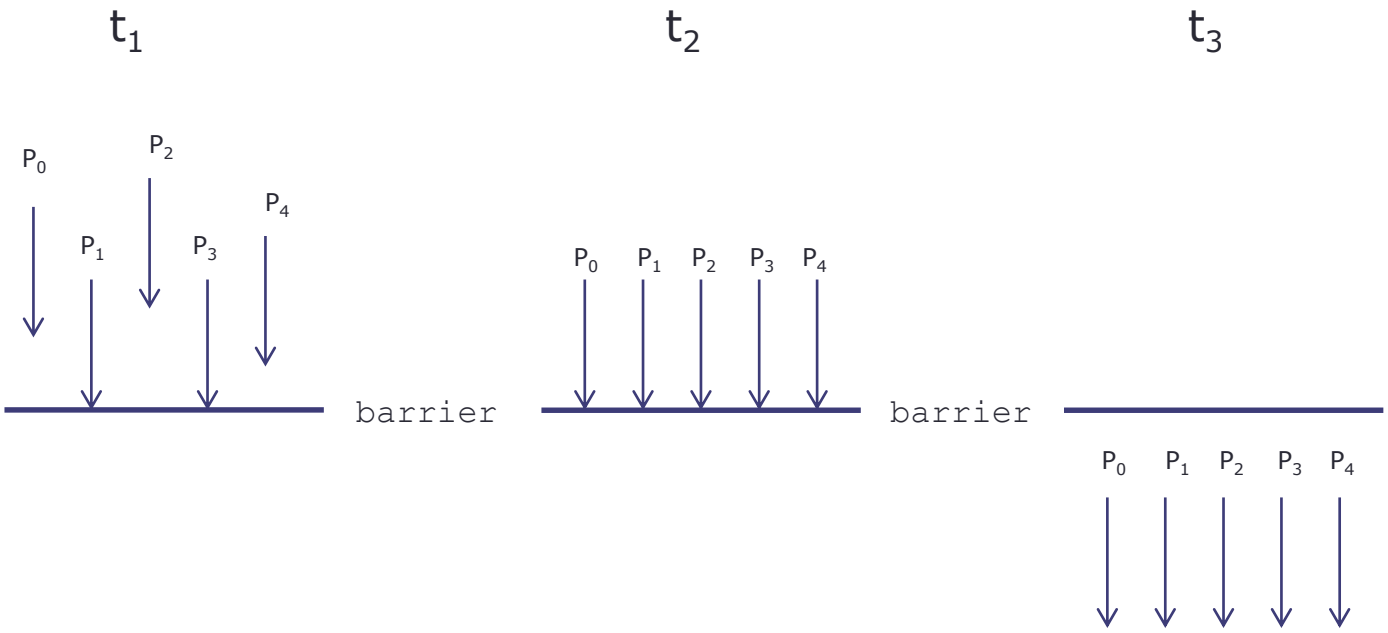
- Gather-Scatter (One-to-All)





Synchronization

- Barriers
  - Define an explicit barrier to hold an execution point.



Communicators

- A communicator is a datastructure that defines a group of processes
  - Each communicator has:
    - Size: the number of processes associated with it
    - Ranks: indexing to define the processes within the communicator
- MPI_COMM_WORLD
  - Created for every MPI program
  - The communicator that defines all of the processes in the current program.
  - Created by MPI_Init

How much MPI do you need?

- One can survive knowing only 6 MPI calls.
  - MPI_INIT()
  - MPI_COMM_SIZE()
  - MPI_COMM_RANK()
  - MPI_SEND
  - MPI_RECV
  - MPI_FINALIZE

Example

- Let's write a program to send one integer from process 0 to process 1.

```
Program MPI
   Implicit None
!
   Include 'mpif.h'
!
   Integer                                    :: rank
   Integer                                    :: buffer
   Integer, Dimension( 1:MPI_status_size )    :: status
   Integer                                    :: error
!
   Call MPI_init( error )
   Call MPI_comm_rank( MPI_comm_world, rank, error )
!
   If( rank == 0 ) Then
      buffer = 33
      Call MPI_send( buffer, 1, MPI_integer, 1, 10, &
                     MPI_comm_world, error )
   End If
!
   If( rank == 1 ) Then
      Call MPI_recv( buffer, 1, MPI_integer, 0, 10, &
                     MPI_comm_world, status, error )
      Print*, 'Rank ', rank, ' buffer=', buffer
      If( buffer /= 33 ) Print*, 'fail'
   End If
   Call MPI_finalize( error )
End Program MPI
```

Anatomy of a MPI call

The simplest call:

MPI_send(buffer, count, data_type, destination,tag, communicator**)**

where:

- BUFFER: data to send
- COUNT: number of elements in buffer .
- DATA_TYPE : which kind of data types in buffer ?
- DESTINATION the receiver
- TAG: the label of the message
- COMMUNICATOR set of processors involved

Example

- Let's write a program to send one integer from process 0 to process 1.

```
Program MPI
   Implicit None
!
   Include 'mpif.h'
!
   Integer                                    :: rank
   Integer                                    :: buffer
   Integer, Dimension( 1:MPI status size )    :: status
```

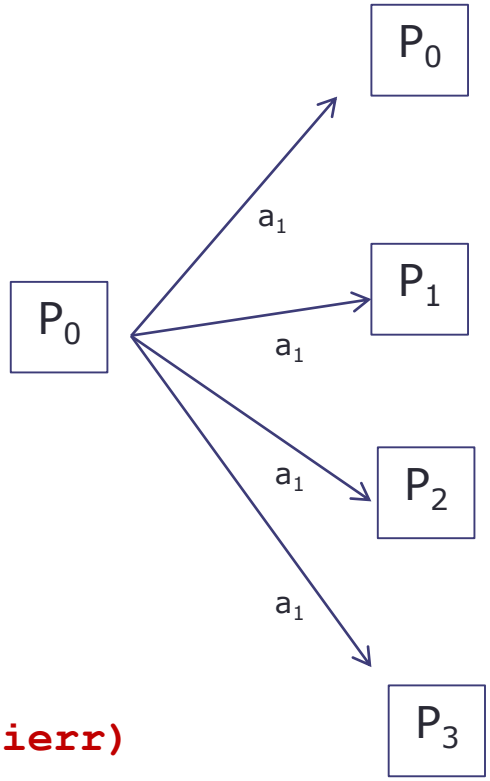
megatron.psc.edu - PuTTY

```
[stbrown@pywiak ~/programming]$ mpif90 send.f -o send
[stbrown@pywiak ~/programming]$ mpirun -np 2 send
 Rank               1   buffer=            33
[stbrown@pywiak ~/programming]$
```

```
   Print*, 'Rank ', rank, ' buffer=', buffer
   If( buffer /= 33 ) Print*, 'fail'
End If
```

Broadcast Example

```
PROGRAM broad_cast
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 IF( myid .EQ. 0 ) THEN
   a(1) = 2.0
   a(2) = 4.0
 END IF
 CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
 WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)
 CALL MPI_FINALIZE(ierr)
 END
```



MPI_BCast

One-to-all communication: same data sent from root process to all others in the communicator

Fortran:

```
INTEGER count, type, root, comm, ierr
CALL MPI_BCAST(buf, count, type, root, comm, ierr)
```

Buf array of type **type**

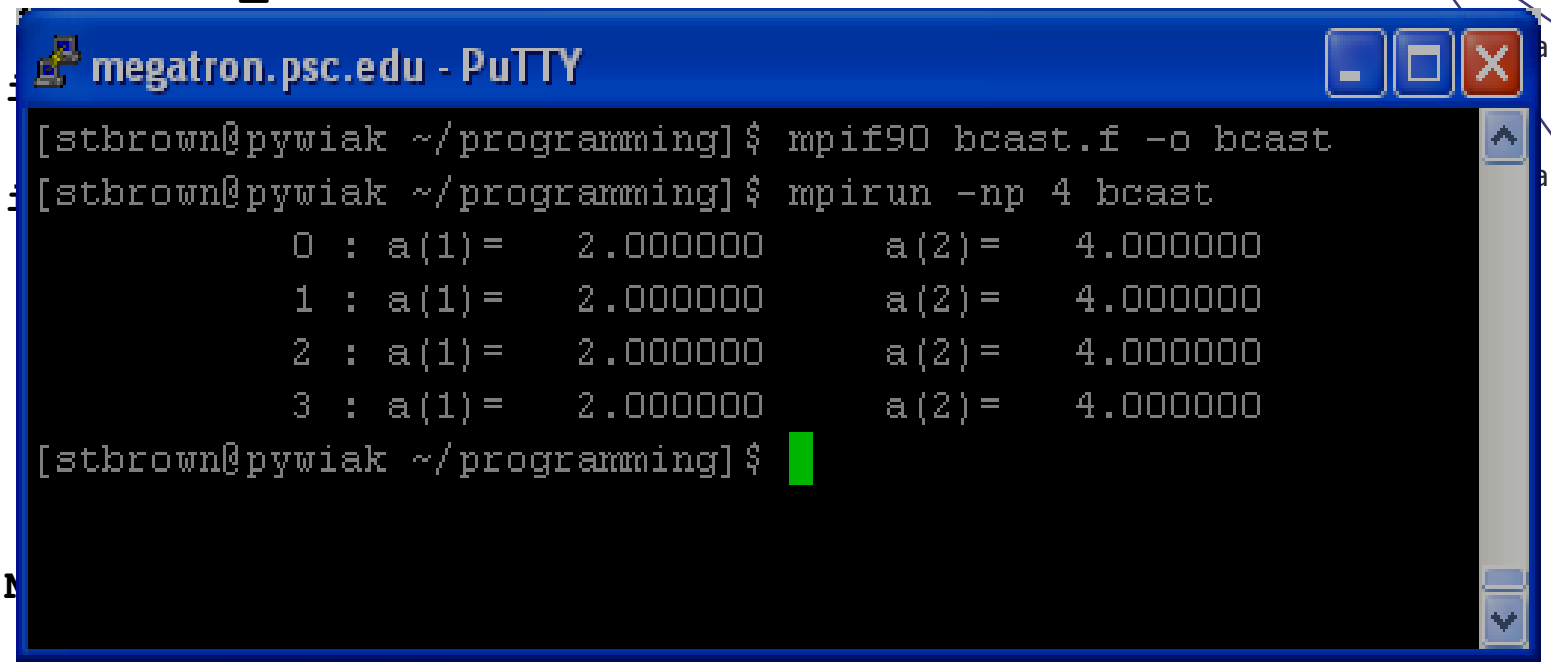
C:

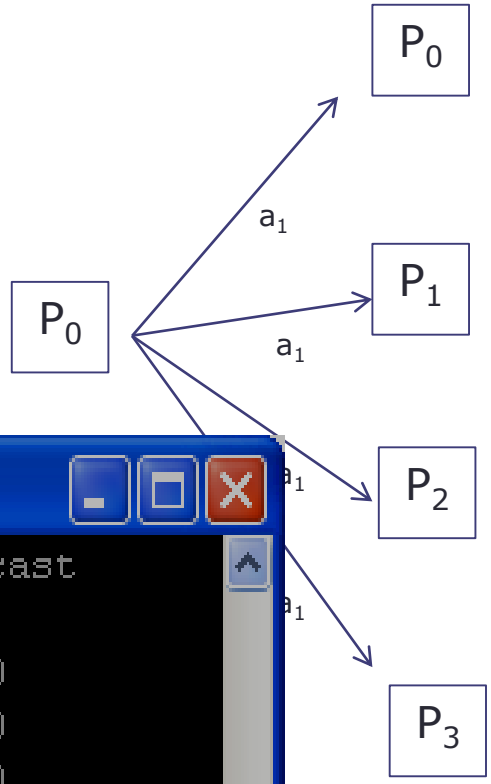
```
int MPI_Bcast(void *buf, int count, MPI_Datatype
  datatypem int root, MPI_Comm comm)
```

All processes must specify same **root**, **rank** and **comm**

Broadcast Example

```
PROGRAM broad_cast
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2)
 CALL MPI_INIT(ierr)
```






```
a(2)
```

Reduction Example

```
 PROGRAM reduce
 INCLUDE 'mpif.h'
 INTEGER ierr, myid, nproc, root
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2), res(2)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
 root = 0
 a(1) = 2.0
 a(2) = 4.0
 CALL MPI_REDUCE(a, res, 2, MPI_REAL, MPI_SUM, root,
&  MPI_COMM_WORLD, ierr)
 IF( myid .EQ. 0 ) THEN
   WRITE(6,*) myid, ': res(1)=', res(1), 'res(2)=', res(2)
 END IF
 CALL MPI_FINALIZE(ierr)
 END
```

Reduction Calls

Fortran:

`MPI_REDUCE(snd_buf,rcv_buf,count,type,op,root,comm,ierr)`

`snd_buf`	input array of type `type` containing local values.
`rcv_buf`	output array of type `type` containing global results
`count`	`(INTEGER)` number of element of `snd_buf` and `rcv_buf`
`type`	`(INTEGER)` MPI type of `snd_buf` and `rcv_buf`
`op`	`(INTEGER)` parallel operation to be performed
`root`	`(INTEGER)` MPI id of the process storing the result
`comm`	`(INTEGER)` communicator of processes involved in the operation
`ierr`	`(INTEGER)` output, error code (if `ierr=0` no error occours)

`MPI_ALLREDUCE( snd_buf,rcv_buf,count,type,op,comm,ierr)`

The argument `root` is missing, the result is stored to all processes.

In C

C:

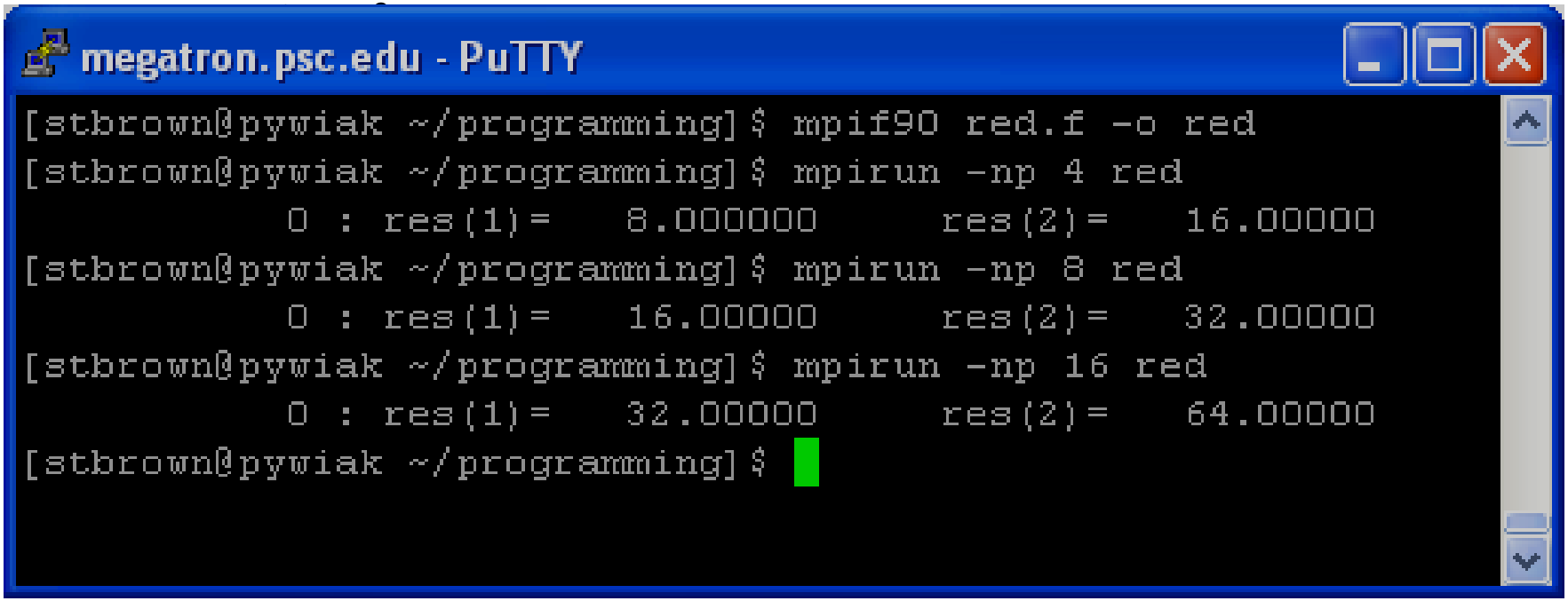
```
int MPI_Reduce(void * snd_buf, void * rcv_buf, int
   count, MPI_Datatype type, MPI_Op op, int root,
   MPI_Comm comm)

int MPI_Allreduce(void * snd_buf, void * rcv_buf,
   int count, MPI_Datatype type, MPI_Op op, MPI_Comm
   comm)
```

Predefined Reductions

MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Reduction Example

```
PROGRAM reduce
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(2), res(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
```



More information

https://computing.llnl.gov/tutorials/mpi/