Message Passing Interface (MPI)

Steve Lantz
Center for Advanced Computing
Cornell University

Workshop: Parallel Computing on Stampede, Oct. 23, 2013

Based on materials developed by CAC and TACC

Outline

- Overview
- Basics
  - Hello World in MPI
  - Compiling and running MPI programs (LAB)
- MPI messages
- Point-to-point communication
  - Deadlock and how to avoid it (LAB)
- Collective communication
  - Reduction operations (LAB)
- Releases
- MPI references and documentation

Introduction

- What is message passing?
  - Sending and receiving messages between *tasks* or *processes*
  - Includes performing operations on data in transit and synchronizing tasks
- Why send messages?
  - Clusters have distributed memory, i.e. each process has its own address space and no way to get at another's
- How do you send messages?
  - Programmer makes use of an Application Programming *Interface* (API)
  - API specifies the functionality of high-level communication routines
  - API's functions give access to a low-level *implementation* that takes care of sockets, buffering, data copying, message routing, etc.

Overview | API for Distributed Memory Parallelism

- Assumption: processes do not see each other’s memory
- Communication speed is determined by some kind of network
  - Typical network = switch + cables + adapters + software stack…
- Key: the *implementation* of MPI (or any message passing API) can be optimized for any given network
  - Expert-level performance
  - No code changes required
  - Works in shared memory, too

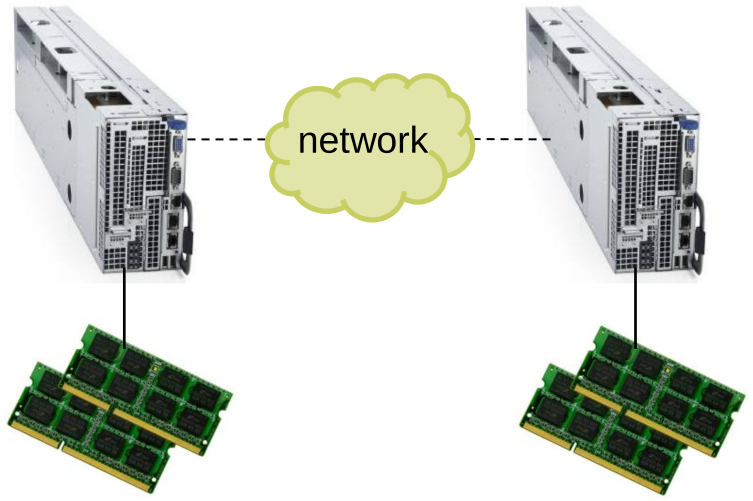


Image of Dell PowerEdge C8220X: http://www.theregister.co.uk/2012/09/19/dell_zeus_c8000_hyperscale_server/

Why Use MPI?

- MPI is a de facto standard
  - Public domain versions are easy to install
  - Vendor-optimized version are available on most hardware
- MPI is "tried and true"
  - MPI-1 was released in 1994, MPI-2 in 1996
- MPI applications can be fairly portable
- MPI is a good way to learn parallel programming
- MPI is expressive: it can be used for many different models of computation, therefore can be used with many different applications
- MPI code is efficient (though some think of it as the "assembly language of parallel processing")
- MPI has freely available implementations (e.g., MPICH)

Simple MPI

Here is the basic outline of a simple MPI program :

- Include the implementation-specific header file --

  #include <mpi.h> inserts basic definitions and types
- Initialize communications –

  MPI_Init initializes the MPI environment
  MPI_Comm_size returns the number of processes
  MPI_Comm_rank returns this process's number (rank)
- Communicate to share data between processes –

  MPI_Send sends a message
  MPI_Recv receives a message
- Exit from the message-passing system --

  MPI_Finalize

Basics | Minimal Code Example: hello_mpi.c

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{
   char message[20];
   int i, rank, size, tag = 99;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
      strcpy(message, "Hello, world!");
      for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
   } else
      MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   printf("Message from process %d : %.13s\n", rank, message);
   MPI_Finalize();
}
```

Basics

Initialize and Close Environment

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  } else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Message from process %d : %.13s\n", rank, message);
  MPI_Finalize();
}
```

Initialize MPI environment

An implementation may also use this call as a mechanism for making the usual argc and argv command-line arguments from “main” available to all tasks (C language only).

Close MPI environment

Basics

Query Environment

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  } else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Message from process %d : %.13s\n", rank, message);
  MPI_Finalize();
}
```

Returns number of processes
This, like nearly all other MPI functions, must be called after MPI_Init and before MPI_Finalize. Input is the name of a communicator (MPI_COMM_WORLD is the global communicator) and output is the size of that communicator.

Returns this process' number, or rank
Input is again the name of a communicator and the output is the rank of this process in that communicator.

Pass Messages

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  } else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Message from process %d : %.13s\n", rank, message);
  MPI_Finalize();
}
```

Send a message
Blocking send of data in the buffer.

Receive a message
Blocking receive of data into the buffer.

Compiling MPI Programs

- Generally, one uses a special compiler or wrapper script
  - Not defined by the standard
  - Consult your implementation
  - Correctly handles include path, library path, and libraries
- On Stampede, use MPICH-style wrappers (the most common)

  ```
  mpicc -o foo foo.c
  mpicxx -o foo foo.cc
  mpif90 -o foo foo.f
  ```
  (also mpif77)

  - Choose compiler+MPI with “module load” (default, Intel13+MVAPICH2)
- 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)

Running MPI Programs

- To run a simple MPI program, use MPICH-style commands

  `mpirun -n 4 ./foo` (usually mpirun is just a soft link to…)

  `mpiexec -n 4 ./foo`

- Some options for running

  `-n` -- states the number of MPI processes to launch

  `-wdir <dirname>` -- starts in the given working directory

  `--help` -- shows all options for *mpirun*

- To run over Stampede's InfiniBand (as part of a batch script)

  `ibrun ./foo`

  – The scheduler handles the rest

- Note: *mpirun*, *mpiexec*, and compiler wrappers are not part of MPI, but they can be found in nearly all implementations

  – There are exceptions: e.g., on older IBM systems, one uses *poe* to run, *mpcc_r* and *mpxlf_r* to compile

Basics | Creating an MPI Batch Script

- To submit a job to the compute nodes on Stampede, you must first create a SLURM batch script with the commands you want to run.

```
#!/bin/bash
#SBATCH -J myMPI            # job name
#SBATCH -o myMPI.o%j        # output/error file (%j = jobID)
#SBATCH -N 1                # number of nodes requested
#SBATCH -n 16               # number of MPI tasks requested
#SBATCH -p development      # queue (partition)
#SBATCH -t 00:01:00         # run time (hh:mm:ss)
#SBATCH -A TG-TRA120006     # account number

echo 2000 > input
ibrun ./myprog < input      # run MPI executable "myprog"
```

LAB: Submitting MPI Programs

- Obtain the **hello_mpi.c** source code via copy-and-paste, or by

```
tar xvf ~tg459572/LABS/IntroMPI_lab.tar
cd IntroMPI_lab/hello
```

- Compile the code using **mpicc** to output the executable **hello_mpi**
- Modify the **myMPI.sh** batch script to run **hello_mpi**
  - Do your really need the “echo” command, e.g.?
- Submit the batch script to SLURM, the batch scheduler
  - Check on progress until the job completes
  - Examine the output file

```
sbatch myMPI.sh
squeue -u <my_username>
less myMPI.o*
```

Messages — Three Parameters Describe the Data

```
MPI_Send( message, 13, MPI_CHAR, i,  type, MPI_COMM_WORLD  );

MPI_Recv( message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
```

Type of data, should be same for send and receive
MPI_Datatype type

Number of elements (items, not bytes)
Recv number should be greater than or equal to amount sent
int count

Address where the data start
void data*

Messages — Three Parameters Specify Routing

```
MPI_Send(  message, 13, MPI_CHAR,  i,  type, MPI_COMM_WORLD  );

MPI_Recv(  message, 20, MPI_CHAR,  0, type, MPI_COMM_WORLD, &status);
```

Identify process you're communicating with by rank number
int dest/src

Arbitrary tag number, must match up (receiver can specify MPI_ANY_TAG to indicate that any tag is acceptable)
int tag

Communicator specified for send and receive must match, no wildcards
MPI_Comm comm

Returns information on received message
MPI_Status status*

Messages Fortran Notes

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

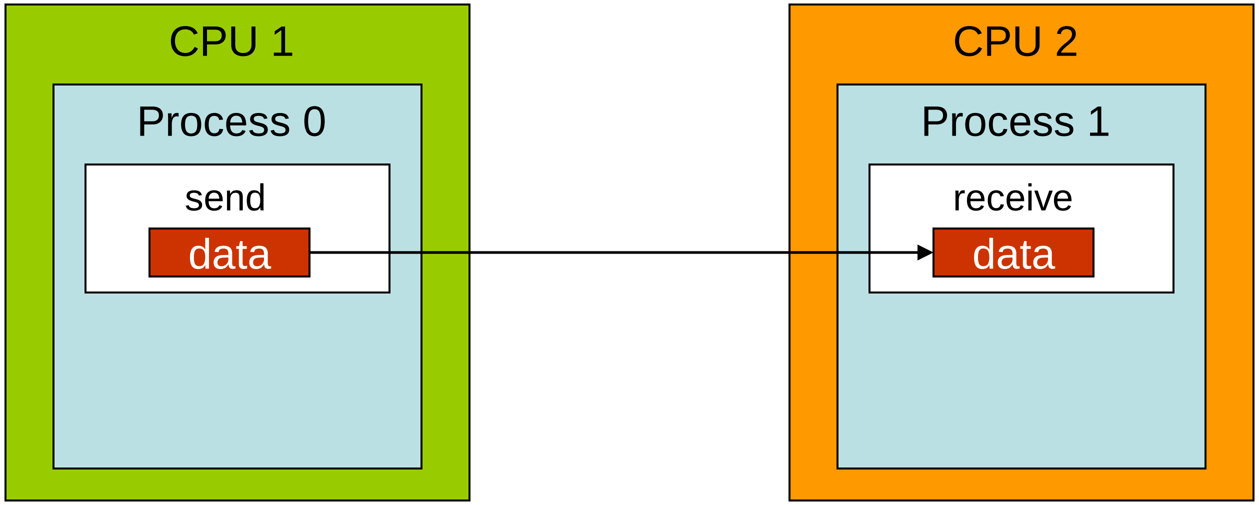
- A few Fortran particulars
  - All Fortran arguments are passed by reference
  - *INTEGER ierr:* variable to store the error code (in C/C++ this is the return value of the function call)

- Wildcards are allowed in C and Fortran
  - *src* can be the wildcard MPI_ANY_SOURCE
  - *tag* can be the wildcard MPI_ANY_TAG
  - *status* returns information on the source and tag
  - Receiver might check *status* when wildcards are used

Point to Point Topics

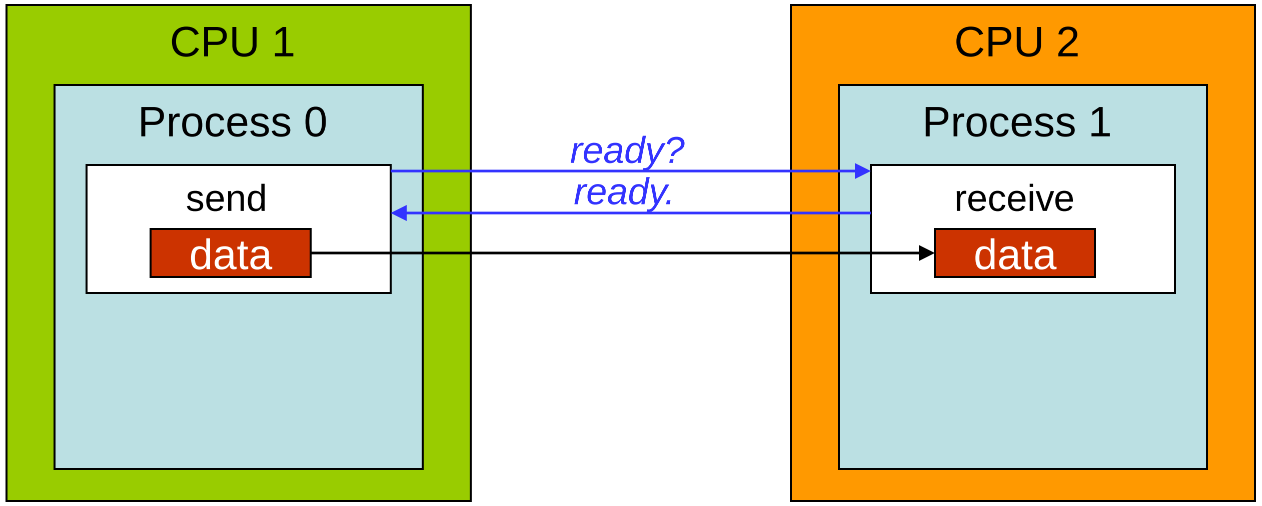
- MPI_Send and MPI_Recv: how simple are they really?
- Synchronous vs. buffered (asynchronous) communication
- Reducing overhead: ready mode, standard mode
- Combined send/receive
- Blocking vs. non-blocking send and receive
- Deadlock, and how to avoid it

Point to Point Send and Recv: Simple?



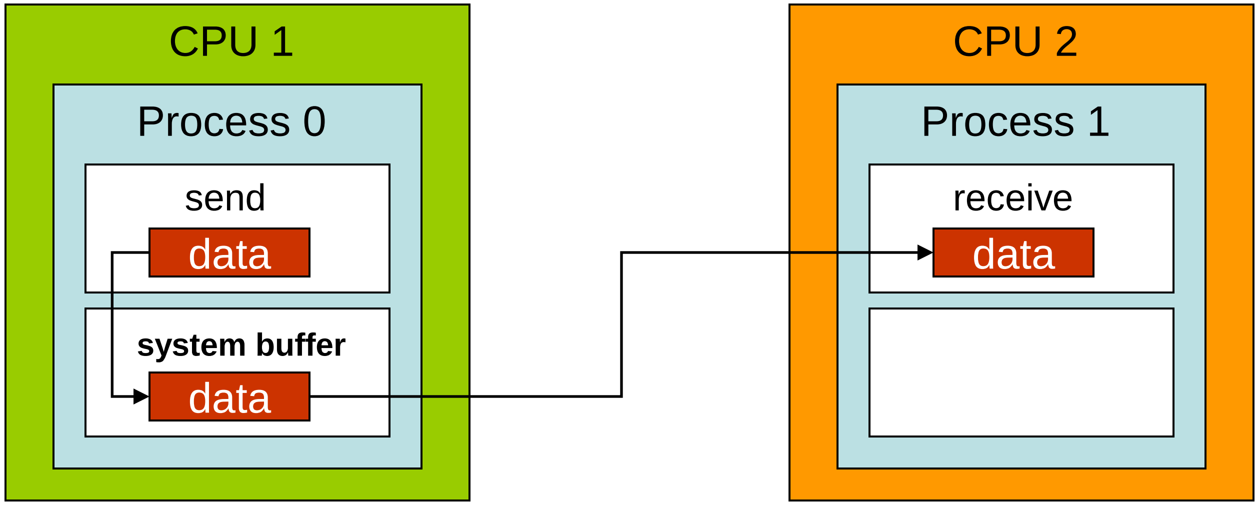
- Sending data ***from*** one point (process/task)
  to another point (process/task)
- One task sends while another receives
- But what if process 1 isn't **ready** for the message from process 0?…
- MPI provides different communication modes in order to help

Point to Point | Synchronous Send, MPI_Ssend



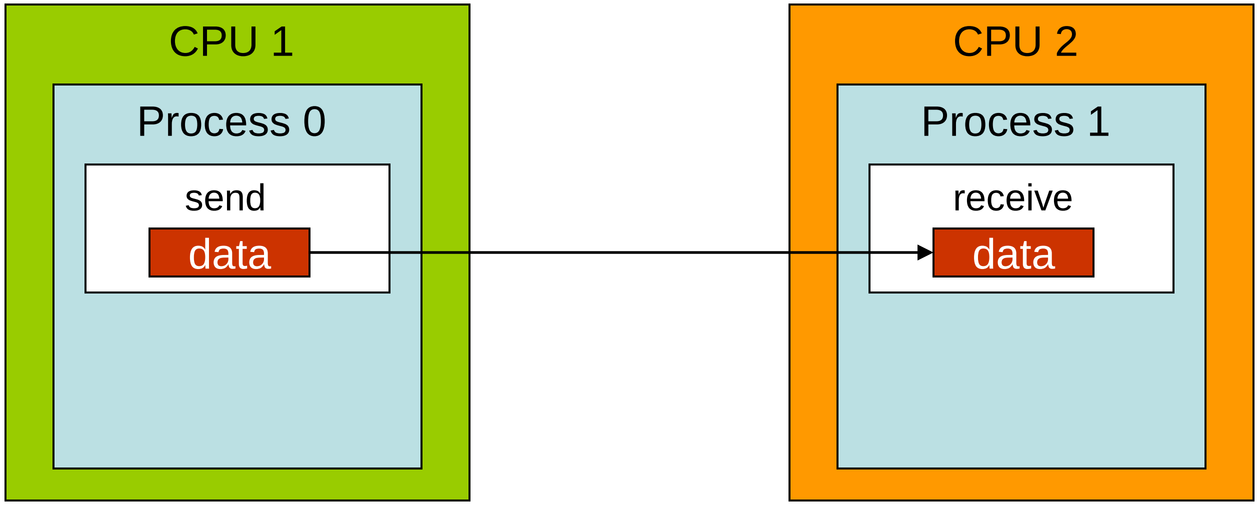
- Handshake procedure ensures both processes are ready
- It’s likely that one of the processes will end up waiting
  - If the *send* call occurs first: sender waits
  - If the *receive* call occurs first: receiver waits
- Waiting and an extra handshake? – this could be slow

Point to Point Buffered Send, MPI_Bsend



- Message data are copied to a system-controlled block of memory
- Process 0 continues executing other tasks without waiting
- When process 1 is ready, it fetches the message from the remote system buffer and stores it in the appropriate memory location
- Must be preceded with a call to MPI_Buffer_attach

Point to Point | Ready Send, MPI_Rsend

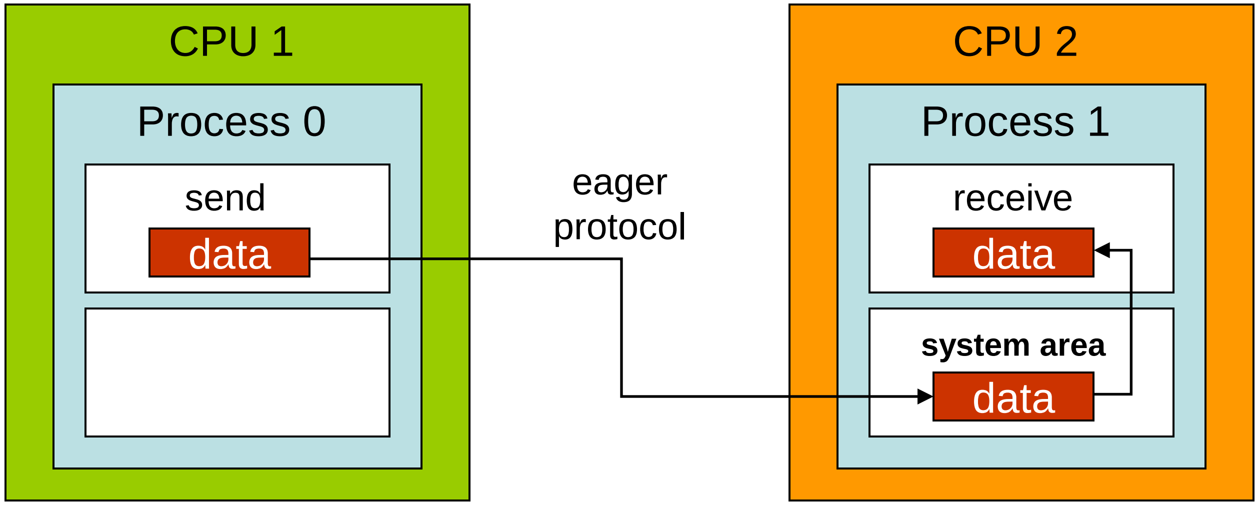


- Process 0 just assumes process 1 is ready! The message is sent!
- Truly simple communication, no extra handshake or copying
- But an error is generated if process 1 is unable to receive
- Only useful when logic dictates that the receiver *must* be ready

Point to Point Overhead

- **System overhead**
  Buffered send has more system overhead due to the extra copy operation.
- **Synchronization overhead**
  Synchronous send has no extra copying but more waiting, because a handshake must arrive before the send can occur.
- **MPI_Send**
  Standard mode tries to trade off between the types of overhead.
  - Large messages use the "rendezvous protocol" to avoid extra copying: a handshake procedure establishes direct communication.
  - Small messages use the "eager protocol" to avoid synchronization cost: the message is quickly copied to a small system buffer on the receiver.

Point to Point | Standard Send, Eager Protocol



- Message goes a system-controlled area of memory *on the receiver*
- Process 0 continues executing other tasks; when process 1 is ready to receive, the system simply copies the message from the system buffer into the appropriate memory location controlled by process
- *Does not* need to be preceded with a call to MPI_Buffer_attach

Point to Point MPI_Sendrecv

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

- Good for two-way communication between a pair of nodes, in which each one sends and receives a message
- However, destination and source need not be the same (ring, e.g.)
- Equivalent to blocking send + blocking receive
- Send and receive use the same communicator but have distinct tags

Point to Point | Send and Recv: So Many Choices

The communication mode indicates how the message should be *sent*.

Communication Mode	Blocking Routines	Non-Blocking Routines
Synchronous	MPI_Ssend	MPI_Issend
Ready	MPI_Rsend	MPI_Irsend
Buffered	MPI_Bsend	MPI_Ibsend
Standard	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv
	MPI_Sendrecv	
	MPI_Sendrecv_replace	

Note: the receive routine does not specify the communication mode -- it is simply blocking or non-blocking.

Point to Point Blocking vs. Non-Blocking

MPI_Send, MPI_Recv

A ***blocking*** call suspends execution of the process until the message buffer being sent/received is safe to use.

MPI_Isend, MPI_Irecv

A ***non-blocking*** call just initiates communication; the status of data transfer and the success of the communication must be verified later by the programmer (MPI_Wait or MPI_Test).

Point to Point | One-Way Blocking/Non-Blocking

- Blocking send, non-blocking recv

```
IF (rank==0) THEN
  ! Do my work, then send to rank 1
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
  ! Do stuff that doesn't yet need recvbuf from rank 0
  CALL MPI_WAIT (req,status,ie)
  ! Do stuff with recvbuf
ENDIF
```

- Non-blocking send, non-blocking recv

```
IF (rank==0) THEN
  ! Get sendbuf ready as soon as possible
  CALL MPI_ISEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
  ! Do other stuff that doesn’t involve sendbuf
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
ENDIF
CALL MPI_WAIT (req,status,ie)
```

Point to Point Two-Way Communication: Deadlock!

- **Deadlock 1**

```
IF (rank==0) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
```

- **Deadlock 2**

```
IF (rank==0) THEN
  CALL MPI_SSEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_SSEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

  – MPI_Send has same problem for count*MPI_REAL > 12K (the MVAPICH2 “eager threshold”; it’s 256K for Intel MPI)

Point to Point | Deadlock Solutions

- Solution 1

```
IF (rank==0) THEN
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
```

- Solution 2

```
IF (rank==0) THEN
  CALL MPI_SENDRECV (sendbuf,count,MPI_REAL,1,tag, &
                recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_SENDRECV (sendbuf,count,MPI_REAL,0,tag, &
                recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

Point to Point More Deadlock Solutions

- Solution 3

```
IF (rank==0) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
CALL MPI_WAIT (req,status)
```

- Solution 4

```
IF (rank==0) THEN
  CALL MPI_BSEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_BSEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

Point to Point Two-way Communications: Summary

	CPU 0	CPU 1
Deadlock 1	Recv/Send	Recv/Send
Deadlock 2	Send/Recv	Send/Recv
Solution 1	Send/Recv	Recv/Send
Solution 2	Sendrecv	Sendrecv
Solution 3	Irecv/Send, Wait	Irecv/Send, Wait
Solution 4	Bsend/Recv	Bsend/Recv

Basics — LAB: Deadlock

- Compile the C or Fortran code to output the executable **deadlock**
- Create a batch script including no #SBATCH parameters:

```
cat > sr.sh
#!/bin/sh
ibrun ./deadlock
```
[ctrl-D to exit cat]

- Submit the job, specifying parameters on the command line

```
sbatch -N 1 -n 8 -p development -t 00:01:00 –A TG-TRA120006 sr.sh
```

  – *Pop quiz: what are some real reasons for <16 tasks on a 16-core node?*

- Check job progress with **squeue**; check output with **less.**
- The program will not end normally. Edit the source code to eliminate deadlock (e.g., use **sendrecv**) and resubmit until the output is good.

Answer to Pop Quiz

Pop quiz: what are some real reasons for wanting to use fewer than 16 tasks on a 16-core node?

- Memory is insufficient
- Processes are multithreaded
  - Parallelized just for shared memory, OpenMP
  - Hybrid code, MPI + OpenMP
- Program is not parallel at all
  - Use `-N 1 -n 1 -p serial` (& no ibrun)

Collective Motivation

- What if one task wants to send to *everyone*?

```
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 more efficient algorithms
- Too slow

Collective Topics

- Overview
- Barrier and Broadcast
- Data Movement Operations
- Reduction Operations

Collective Overview

- Collective calls involve ALL processes within a communicator
- There are 3 basic types of collective communications:
  - Synchronization (MPI_Barrier)
  - Data movement (MPI_Bcast/Scatter/Gather/Allgather/Alltoall)
  - Collective computation (MPI_Reduce/Allreduce/Scan)
- Programming considerations & restrictions
  - Blocking operation
  - No use of message tag argument
  - Collective operations within subsets of processes require separate grouping and new communicator
  - Can only be used with MPI predefined datatypes

Collective | Barrier Synchronization, Broadcast

- *Barrier* blocks until all processes in comm have called it
  - Useful when measuring communication/computation time

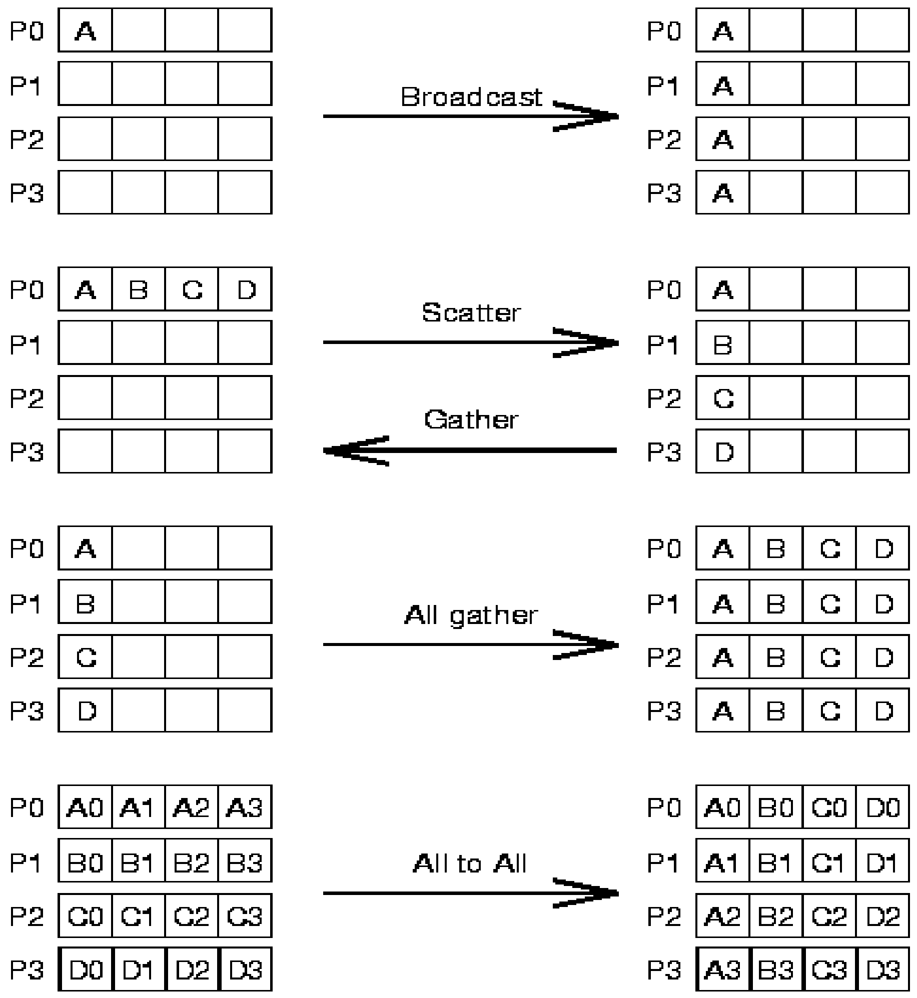
```
mpi_barrier(comm, ierr)
MPI_Barrier(comm)
```

- *Broadcast* sends data from root to all processes in comm
  - Again, blocks until all tasks have called it

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

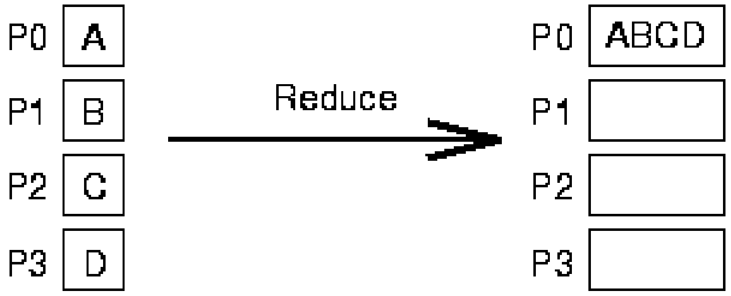
Collective Data Movement

- Broadcast
- Scatter/Gather
- Allgather
- Alltoall

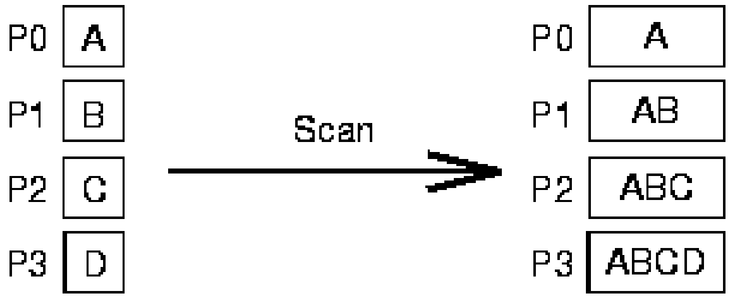


Collective Reduction Operations

- Reduce



- Scan (Prefix)



Collective Reduction Operations

Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical xor
MPI_BXOR	Logical xor
MPI_MAXLOC	Max value and location
MPI_MINLOC	Min value and location

LAB: Allreduce

- In the call to MPI_Allreduce, the reduction operation is wrong!
  - Modify the C or Fortran source to use the correct operation
- Compile the C or Fortran code to output the executable **allreduce**
- Submit the **myall.sh** batch script to SLURM, the batch scheduler
  - Check on progress until the job completes
  - Examine the output file

```
sbatch myall.sh
squeue -u <my_username>
less myall.o*
```

- Verify that you got the expected answer

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
- This presentation has covered just MPI-1 routines

MPI-2

- MPI-2 includes features left out of MPI-1
  - One-sided communications
  - Dynamic process control
  - More complicated collectives
  - Parallel I/O (MPI-IO)
- Implementations of MPI-2 came along only gradually
  - Not quickly undertaken after the reference document was released (in 1997)
  - Now OpenMPI, MPICH2 (and its descendants), and the vendor implementations are nearly complete or fully complete
- Most applications still rely on MPI-1, plus maybe MPI-IO

References

- MPI-1 and MPI-2 standards
  - http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
  - http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm
  - http://www.mcs.anl.gov/mpi/ (other mirror sites)
- Freely available implementations
  - MPICH, http://www.mcs.anl.gov/mpi/mpich
  - LAM-MPI, 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

Extra Slides

MPI_COMM MPI Communicators

- Communicators
  - Collections of processes that can communicate with each other
  - Most MPI routines require a communicator as an argument
  - Predefined communicator MPI_COMM_WORLD encompasses all tasks
  - New communicators can be defined; any number can co-exist
- Each communicator must be able to answer two questions
  - *How many processes exist in this communicator?*
  - MPI_Comm_size returns the answer, say, N_p
  - *Of these processes, which process (numerical rank) am I?*
  - MPI_Comm_rank returns the rank of the current process within the communicator, an integer between 0 and N_p-1 inclusive
  - Typically these functions are called just after MPI_Init

MPI_COMM C Example: param.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();
}
```

MPI_COMM C++ Example: param.cc

```
#include "mpif.h"
[other includes]
int main(int argc, char *argv[]){
   int np,  mype,  ierr;
 [other declarations]
            :
          MPI::Init(argc, argv);
   np   = MPI::COMM_WORLD.Get_size();
   mype = MPI::COMM_WORLD.Get_rank();
            :
        [actual work goes here]
            :
          MPI::Finalize();
}
```

MPI_COMM Fortran Example: param.f90

```
program param
  include 'mpif.h'
  integer ierr, np, mype

  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 Modes

Mode	Pros	Cons
Synchronous – sending and receiving tasks must 'handshake'.	- Safest, therefore most portable - No need for extra buffer space - SEND/RECV order not critical	Synchronization overhead
Ready- assumes that a 'ready to receive' message has already been received.	- Lowest total overhead - No need for extra buffer space - Handshake not required	RECV *must* precede SEND
Buffered – move data to a buffer so process does not wait.	- Decouples SEND from RECV - No sync overhead on SEND - Programmer controls buffer size	Buffer copy overhead
Standard – defined by the implementer; meant to take advantage of the local system.	- Good for many cases - Small messages go right away - Large messages must sync - Compromise position	Your program may not be suitable

Point to Point C Example: oneway.c

```
#include "mpi.h"
main(int argc, char **argv){
  int ierr, mype, myworld; double a[2];
  MPI_Status status;
  MPI_Comm icomm = MPI_COMM_WORLD;
  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(icomm, &mype);
  ierr = MPI_Comm_size(icomm, &myworld);
  if(mype == 0){
    a[0] = mype; a[1] = mype+1;
    ierr = MPI_Ssend(a,2,MPI_DOUBLE,1,9,icomm);
  }
  else if (mype == 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 Fortran Example: oneway.f90

```
program oneway
  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) = dble(mype); a(2) = dble(mype+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",i2," received A array =",2f8.4)',mype,A
  endif
  call mpi_finalize(ierr)
end program
```

Collective C Example: allreduce.c

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

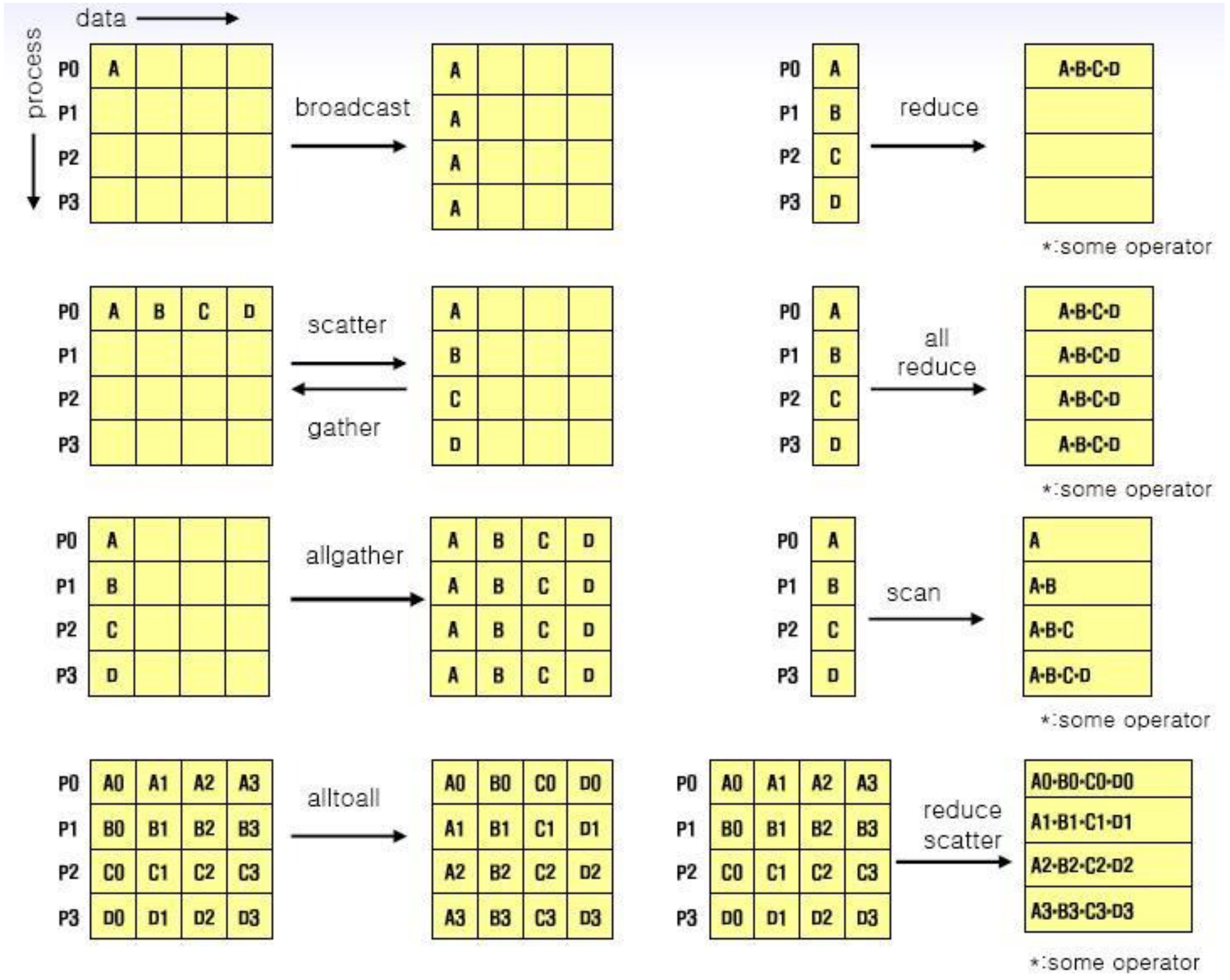
Collective Fortran Example: allreduce.f90

```
program allreduce
  include 'mpif.h'
  double precision :: val, sum
  icomm = MPI_COMM_WORLD
  knt = 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,knt,MPI_REAL8,MPI_SUM,icomm,ierr)

  ncalc = (npes-1 + mod(npes,2))*(npes/2)
  print '(" pe#",i5," sum =",f5.0, " calc. sum =",i5)', &
          mype, sum, ncalc
  call mpi_finalize(ierr)
end program
```

The Collective Collection!


data
process
P0
P1
P2
P3
A
broadcast
reduce
A·B·C·D
*:some operator
B
C
D
scatter
gather
all reduce
allgather
scan
A·B
A·B·C
alltoall
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
D0
D1
D2
D3
reduce scatter
A0·B0·C0·D0
A1·B1·C1·D1
A2·B2·C2·D2
A3·B3·C3·D3