Parallel Programming with MPI

Lars Koesterke

04/01/2014: HPC Workshop at ASU

04/04/2014: HPC Workshop at CSU

Outline

- Message Passing Overview
- Compiling and running MPI programs
- Point-to-Point Communication
- Collective Communication

OVERVIEW

Message Passing Overview

- What is message passing?

Simple Answer: The sending and receiving of messages between diverse computational resources.

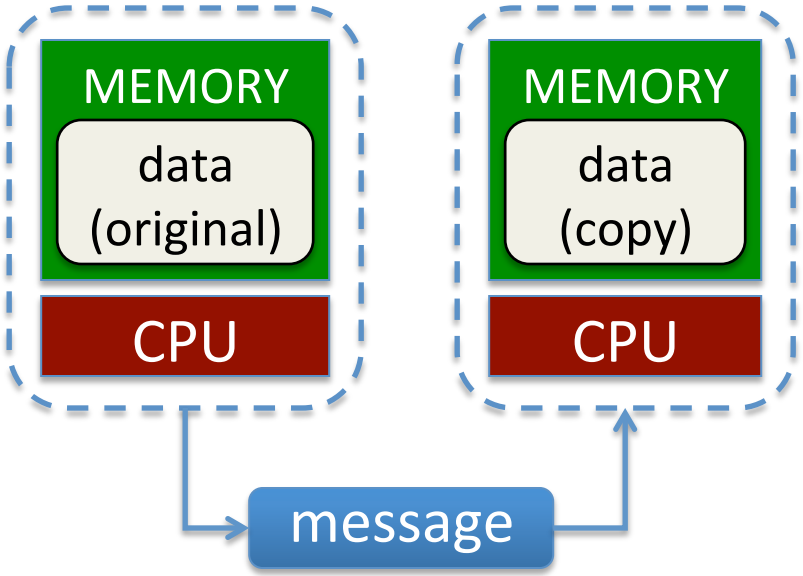
Message Passing Overview

- Messages may be used for
  - sending data
  - performing operations on data
  - synchronization between tasks

- Why do we need to send/receive messages?
  - On clusters, each node has its own address space, and no way to get at another's, except over the network

Message Passing Model

- Tasks send and receive messages to exchange data.
- Data transfer requires a cooperative operation to be performed by each process.
- The programmer is responsible for determining all parallelism.
- Message Passing Interface (MPI) was first released in 1994. (MPI-2 in 1996.)
- MPI is the de facto standard for message passing.



http://www-unix.mcs.anl.gov/mpi/

What is MPI?

- MPI *is*
  - An acronym for **M**essage **P**assing **I**nterface
  - A standard Application Programming Interface (API)

- MPI *is not*
  - A language
  - An implementation
  - Specific to a particular machine

MPI Fundamentals

- Subsets of functionality
  - basic (about 6 functions)
  - intermediate
  - advanced (up to 125 functions)

- One goal of MPI is to provide access to advanced parallel hardware for application scientists (not just programmers and computer scientists)

- Many high-level application libraries are based on MPI
  - PETSc
  - SAMRAI
  - Cactus
  - FFTW
  - PLAPACK

Why learn MPI?

- MPI is a standard
  - Public domain version easy to install
  - Vendor-optimized version available on most communication hardware
- MPI applications are 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 the theory of parallel computing.

Compiling MPI Programs

- Building simple MPI programs, using MPICH

```
% mpicc  -o first  first.c
% mpif90 -o firstf firstf.f
```
(also mpif77)

- These are simply shell script wrappers for system compilers.
-
- Some MPI specific compiler options
  - `-mpilog` : Generate log files of MPI calls
  - `-mpitrace` : Trace execution of MPI calls

Compiling MPI Programs

- The names of the mpiXXX compiler scripts are *not* specified by the MPI standard.
- Examples:
  - IBM: `mpcc_r`, `mpxlf_r`
  - Kraken (A Cray system in the Teragrid): `cc`, and `ftn`

Running MPI Programs

- To run a simple MPI program using MPICH

```
% mpirun -np 2 <progname>
```

- Some MPI specific running options
  - `-t` : shows the commands that `mpirun` would execute
  - `-help` : shows all options for `mpirun`

- The name "mpirun" is not part of the standard, other names include
  - IBM SP: `poe`
  - Lonestar/Stampede: `ibrun`
  - Mpich2: `mpiexec`

MPI BASICS

Outline

- Basic MPI code structure
- Point-to-point communication
- Collective communication

MPI Initialization & Termination

- All processes must initialize and finalize MPI (each is a collective call).
  - `MPI_Init` : starts up the MPI runtime environment
  - `MPI_Finalize` : shuts down the MPI runtime environment
- Must include header files – provides basic MPI definitions and types.
  - Header File

Fortran 77	Fortran 90	C/C++
`include 'mpif.h'`	`use mpi`	`#include "mpi.h"`

  - Format of MPI calls

Fortran 77/90 binding	C/C++ binding
`CALL MPI_XYYY(parameters…, ierr)`	`ierr = MPI_Xyyy(parameters…)`

Communicators

- MPI uses `MPI_Comm` objects to define subsets of processors which may communicate with one another.
- Most MPI routines require you to specify a communicator as an argument.
- Default communicator: `MPI_COMM_WORLD` – a predefined communicator that includes all of your processes.
- In MPI terminology, a processor's "Rank" is:
  - A unique ID within a communicator
  - Assigned by the system when the communicator is created
  - Part of a contiguous integer range which begins at zero
  - The way one specificies the source and destination of messages

Communicators

- Two common functions for interacting with an `MPI_Comm` object are:
- `MPI_Comm_size(MPI_Comm_World, int *np)`
  - Gets the number of processes in a run, *NP*
- `MPI_Comm_rank(MPI_Comm_World, int *rank)`
  - Gets the rank of the current process
  - returns a value between 0 and *NP*-1 inclusive

- Both are typically called just after `MPI_Init`.

Sample MPI code (C)

```
#include <mpi.h>
 [other includes]

int main(int argc, char **argv){
   int ierr, np, rank;
    [other declarations]

   ierr = MPI_Init(&argc, &argv);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);
   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                :
    [actual work goes here]
                :
   MPI_Finalize();
}
```

Sample MPI code (C++)

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

Sample MPI code (F90)

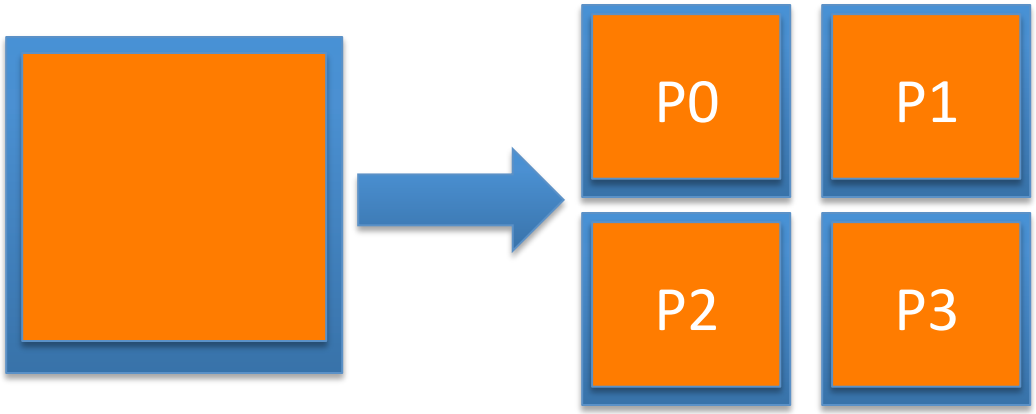
```
program samplempi
  use mpi
 [other includes]

  integer :: ierr, np, rank
     [other declarations]

  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, np, ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, rank, ierr)
                :
       [actual work goes here]
                :
  call mpi_finalize(ierr)
end program
```

MPI Execution

- Every process gets a copy of the executable: ***S**ingle **P**rogram, **M**ultiple **D**ata* (SPMD).
- They all start executing it.
- Each looks at its own rank to determine which part of the problem to work on.
- Each process works **completely independently** of the other processes, except when communicating.

Why do we need to communicate? What is communicated?

- Example: Stencil update; $y_{i,j} = a(x_{i,j} + x_{i-1,j} + x_{i+1,j} + x_{x,j-1} + x_{x,j+1})$
- Simple 2d domain decomposition

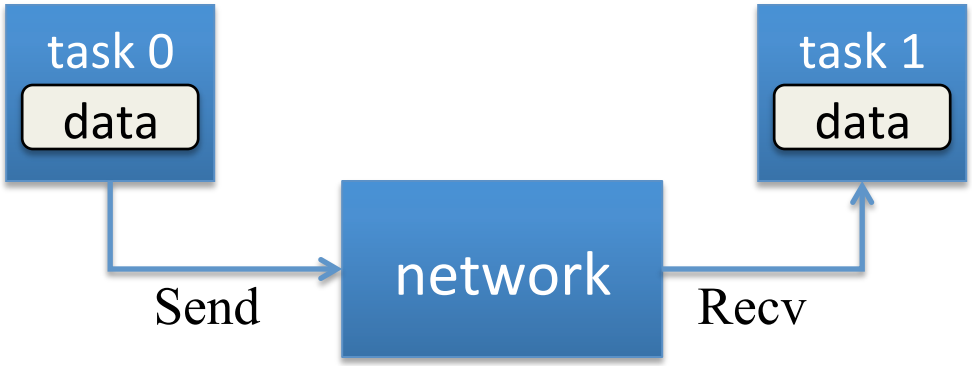


Transfer ghost layers between different processes

POINT-TO-POINT COMMUNICATION

Point-to-Point Communication

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



P2P Communication: Send

```
MPI_Send(void *buf,
         int count,
         MPI_Datatype datatype,
         int dest,
         int tag,
         MPI_Comm comm);
```

Argument	Description
buf	initial address of send/receive buffer
count	number of items to send
datatype	MPI data type of items to send/receive
dest	MPI rank or task receiving the data
tag	message ID
comm	MPI communicator where the exchange occurs

P2P Communication: Receive

```
MPI_Recv(void *buf,
         int count,
         MPI_Datatype datatype,
         int source,
         int tag,
         MPI_Comm comm,
         MPI_Status *status)
```

Argument	Description
buf	initial address of send/receive buffer
count	number of items to send
datatype	MPI data type of items to send/receive
source	MPI rank of task sending the data
tag	message ID
comm	MPI communicator where the exchange occurs
status	returns information on the message received

Summary: MPI_Send & MPI_Recv

```
MPI_Send(buf, count, datatype, dest,   tag, comm);
MPI_Recv(buf, count, datatype, source, tag, comm, status);
```

- In the `status` object, the system can return details about the message received. Can pass default `MPI_STATUS_IGNORE` object instead.
- These calls are "blocking"
- This means that program flow does not return to the calling function until the send/recv pair is completed.
- Can lead to a condition known as "deadlock" in case a Send is not paired with a matching Receive.

MPI_SendRecv

```
MPI_SendRecv(/*send arguments*/
             sendbuf, sendcount, sendtype,
             dest, sendtag,
             /*receive arguments*/
             recvbuf, recvcount, recvtype, source,
             recvtag, comm, status);
```

- Union of MPI_Send and MPI_Recv commands
- Executes a **blocking** send & receive operation
- Send and Receive stages use the same communicator, but have distinct tags
- Useful for communications patterns where each node both sends and receives messages (two-way communication)

A quick overview of MPI's send modes

MPI has a number of different "send modes." These represent different choices of *buffering* (where is the data kept until it is received) and *synchronization* (when does a send complete). In the following, I use "send buffer" for the user-provided buffer to send.

- MPI_Send: will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- MPI_Bsend: May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- MPI_Ssend: will not return until matching receive posted
- MPI_Rsend: May be used ONLY if matching receive already posted. User responsible for writing a correct program.
- MPI_Isend: Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free). Note also that while the **I** refers to *immediate*, there is no performance requirement on MPI_Isend. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.
- And then there is: MPI_Ibsend, MPI_Issend, and MPI_Irsend

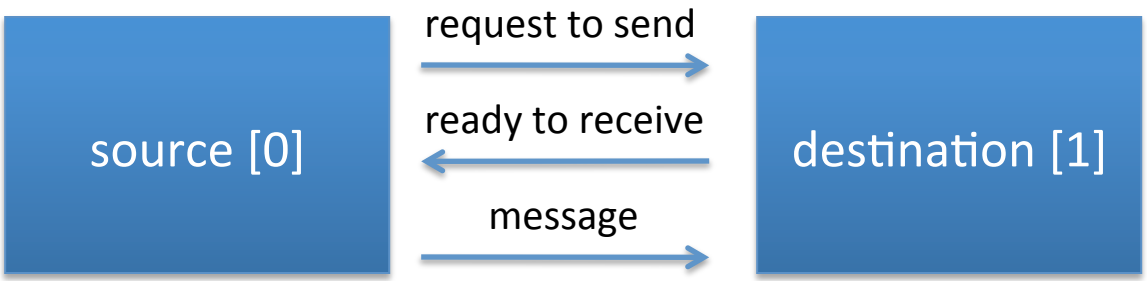
A quick overview of MPI's send modes

Recommendations

- The best performance is likely if you can write your program so that you could use just MPI_Ssend; in that case, an MPI implementation can completely avoid buffering data. Use MPI_Send instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data. (Unfortunately, one vendor has chosen to have MPI_Send emphasize buffering over performance; on that system, MPI_Ssend may perform better.) If nonblocking routines are necessary, then try to use MPI_Isend or MPI_Irecv. Use MPI_Bsend only when it is too inconvienent to use MPI_Isend. The remaining routines, MPI_Rsend, MPI_Issend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

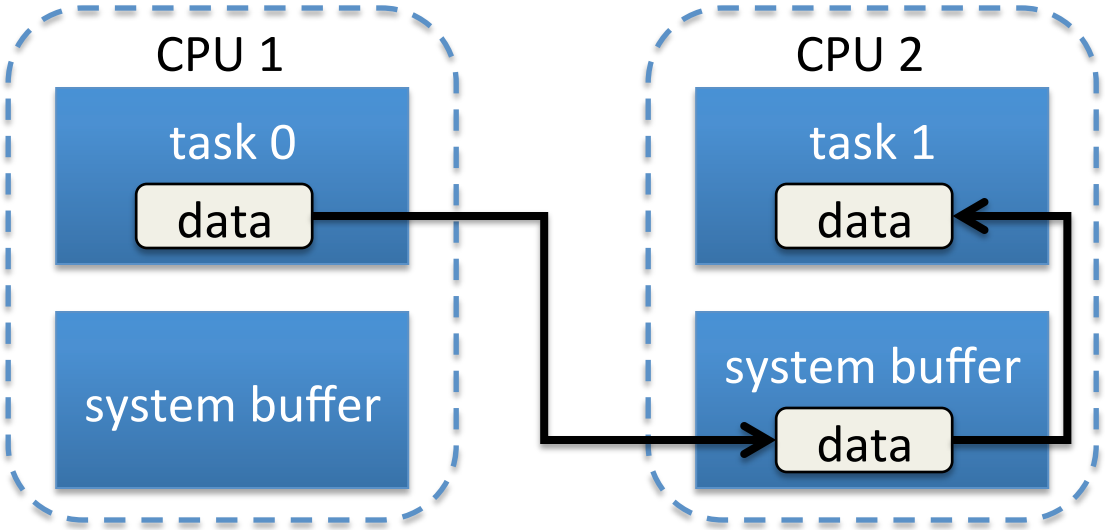
from: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

Synchronous Communication



- Process 0 waits until process 1 is ready
- *Handshaking* occurs between send & receive tasks to confirm a safe send.
- Blocking send/receive
- This is rarely the case in real world.
- Need to be able to deal with storing data when multiple tasks are out of sync.
- MPI_Ssend & MPI_Srecv

Buffered Communication



- The contents of the message is copied into a system-controlled block of memory (system buffer)
- Process 0 continues executing other tasks; when process 1 is ready to receive, the system simply copies the buffered message into the appropriate memory location controlled by process 1.
- MPI_Bsend & MPI_Brecv

Blocking vs. Non-blocking

Blocking

- A blocking send routine will only return after it is *safe* to modify the buffer.
- *Safe* means that modification will not affect the data to be sent.
- *Safe* does not imply that the data was actually received.
- MPI_Send, MPI_Recv

Non-blocking

- Send/receive routines return immediately.
- Non-blocking operations request that the MPI library perform the operation “when possible”.
- It is **unsafe** to modify the buffer until the requested operation has been performed. There are *wait* routines used to do this (MPI_Wait).
- Primarily used to overlap computation with communication.
- MPI_Isend, MPI_Irecv

Blocking/Non-Blocking Routines

Description	Syntax for C bindings
Blocking send	`MPI_Send(buf, count, datatype, dest, tag, comm)`
Blocking receive	`MPI_Recv(buf, count, datatype, source, tag, comm, status)`
Non-blocking send	`MPI_Isend(buf, count, datatype, dest, tag, comm, request)`
Non-blocking receive	`MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

- `MPI_Request` objects are used by non-blocking send and receive calls.
- `MPI_Wait()`/`MPI_Waitall()`
  - Blocking functions
  - Pause program execution until outstanding Isend/Irecv calls have completed.

Send/Recv Pairs in Code

- **Blocking Send & Blocking Recv**

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

- **Non-blocking Send & Blocking Recv**

```
IF (rank==0) THEN
  CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
ENDIF
CALL MPI_WAIT(req, wait_status)
```

Deadlock Example

```
! The following code contains a deadlock... can you spot it?
IF (rank==0) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
   CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
ELSEIF (rank==1) THEN
   CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
   CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
ENDIF

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

Alternative Deadlock Solutions

```
! Solution using sendrecv
IF (rank==0) THEN
   CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 1, sendtag,
                     recvbuf, count, MPI_REAL, 1, recvtag,
                     MPI_COMM_WORLD, status, ierr)
ELSEIF (rank==1) THEN
   CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 0, sendtag,
                     recvbuf, count, MPI_REAL, 0, recvtag,
                     MPI_COMM_WORLD, status, ierr)
ENDIF
```

```
! Another possible solution (using all non-blocking calls)
IF (rank==0) THEN
   CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req1,ierr)
   CALL MPI_IRECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req2,ierr)
ELSEIF (rank==1) THEN
   CALL MPI_ISEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req1,ierr)
   CALL MPI_IRECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req2,ierr)
ENDIF
CALL MPI_WAIT(req1, wait_status, ierr)
CALL MPI_WAIT(req2, wait_status, ierr)
```

COLLECTIVE COMMUNICATION

Collective Communication

- Defined as communication between > 2 processors
  - One-to-many
  - Many-to-one
  - Many-to-many

- A collective operation requires that all processes within the communicator group call the *same* collective communication function with matching arguments.

Naïve collective communication

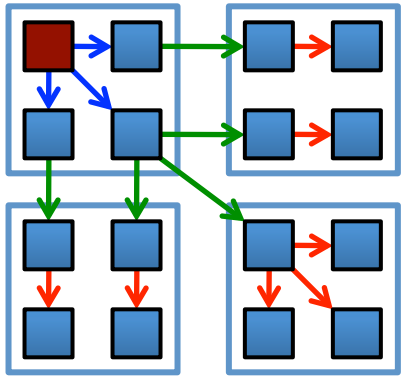
- With MPI_Send/Recv, you have all the tools…

```
if (rank == 0 ) {
  for (int id=1; id<np; id++) {
    MPI_Send( ..., /* dest= */ id, ... );
  }
} else {
  MPI_Recv( ..., /* source= */ 0, ... );
}
```

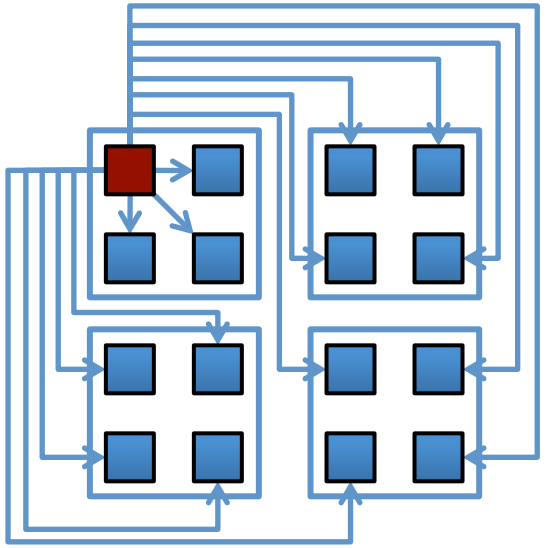
- This code "broadcasts" information from proc. 0 to all other processors, but:
  - It's too primitive: no room for the OS/hardware to optimize
  - It uses blocking communications (deadlocks!)

Broadcast Implementation

Provided by MPI



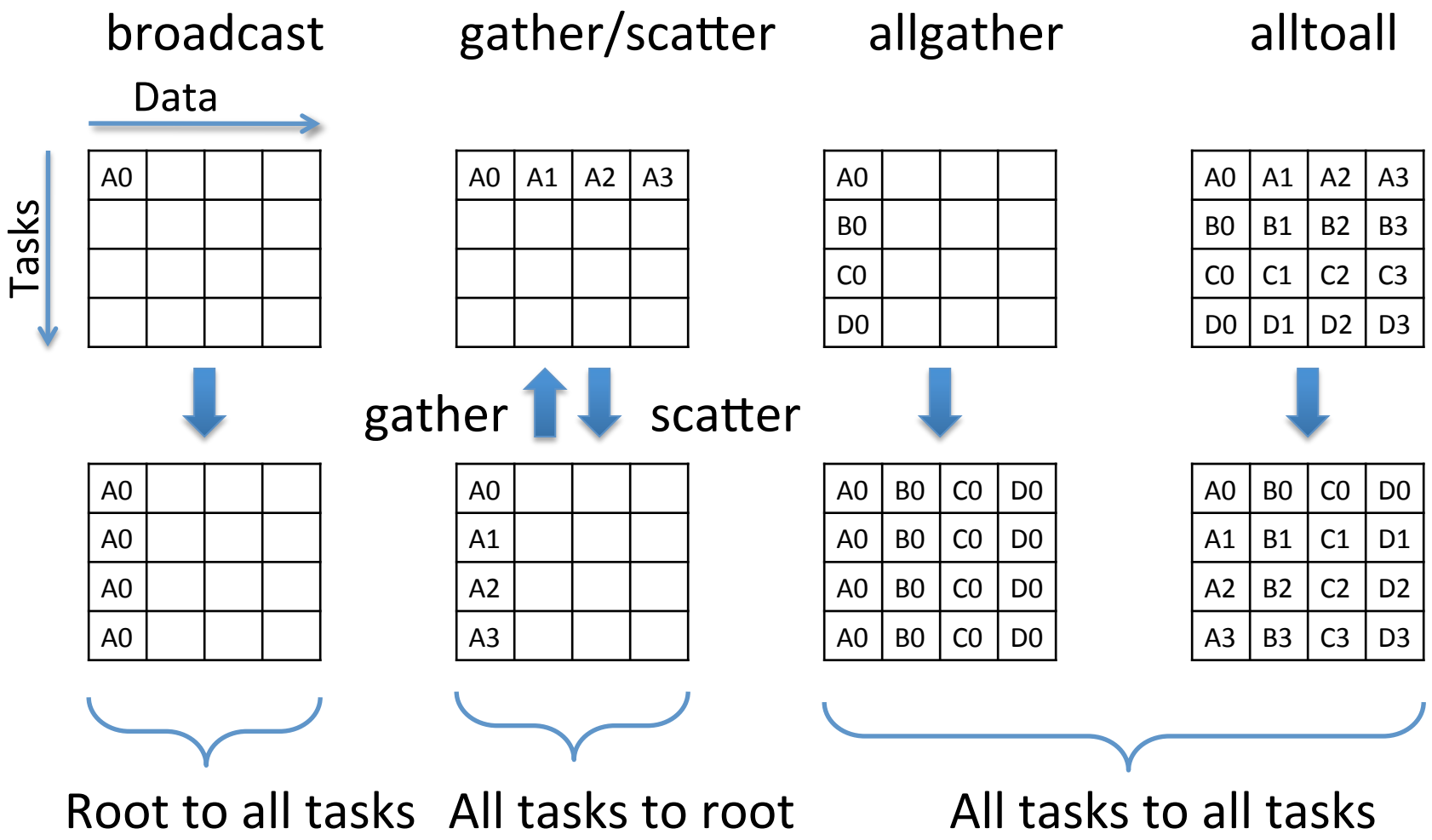
Naïve Implementation



The Basics of Collective Communications

- Involve **ALL** processes within a communicator.
- It is the programmer's responsibility to ensure that all processors call the **same** collective communication at the same time.
- Type of collective operations
  - **Synchronization** (MPI_Barrier)
  - **Data Movement** (MPI_Bcast/Scatter/Gather/Allgather/AlltoAll)
  - **Computation** (MPI_Reduce/Allreduce/Scan)
- Programming considerations & restrictions
  - Collective Communications are **blocking** operations
  - Collective operations on subsets of processes require separate grouping/ new communicators
  - The size of data sent must exactly match the size of data received, not the case for P2P communications

Collective Operation Visualization



Barrier

MPI_BARRIER(comm, ierr)		
IN	comm	Communicator

- Each task waits in the Barrier until all tasks in the communicator have reached it.
- Can be used when measuring communication/computation time and for debugging.
- Caution must be exercised to avoid over-synchronization: This will unnecessarily slow down program execution.

Broadcast

MPI_BCAST(buf, count, datatype, root, comm, ierr)		
INOUT	buf	starting address of buffer
IN	count	number of entries in buffer
IN	datatype	data type of buffer
IN	root	rank of broadcast root
IN	Comm	communicator

- Broadcast a message from the process with rank “root” to all the processes in the group (in the communicator).

MPI_Scatter

```
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
            recvtype, root, comm)
```

IN	sendbuf	address of send buffer
IN	sendcount	number of elements sent to each process
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	Recvcount	number of elements in receive buffer
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process
IN	comm	communicator

- The root process divides its send buffer into *n* equal segments and sends.
- Each process receives a segment from the root and places it in its receive buffer.
- The reverse of MPI_Gather

MPI_Gather

```
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
           recvtype, root, comm)
```

IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements in send buffer
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	Recvcount	number of elements for any single receive
IN	recvtype	data type of receive buffer elements
IN	root	rank of receiving process
IN	comm	Communicator

- Each process sends the contents of its send buffer to the root process.
- Root stores them in its receive buffer according to the ranks of the senders.
- The reverse of MPI_Scatter

MPI_Allgather

```
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
              recvtype, comm)
```

IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements in send buffer
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements received from any process
IN	recvtype	data type of receive buffer elements
IN	comm	communicator

- An MPI_Gather whose result ends up on all processors.
- Each task in the group, in effect, performs a one-to-one broadcasting operation within the group.

MPI_Alltoall

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm)
```

IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements sent to each process
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements received from any process
IN	recvtype	data type of receive buffer elements
IN	Comm	communicator

- Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order, by index.

MPI_Reduce

```
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
```

IN	sendbuf	address of send buffer
OUT	recvbuf	address of receive buffer on root process
IN	count	number of elements in send buffer
IN	datatype	data type of elements of send buffer
IN	op	reduce operation
IN	root	rank of root process
IN	comm	communicator

- Applies a reduction operation on all tasks in the group and places the result in the receive buffer on the root process.
- Possible operations are MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD, ...

MPI_Allreduce

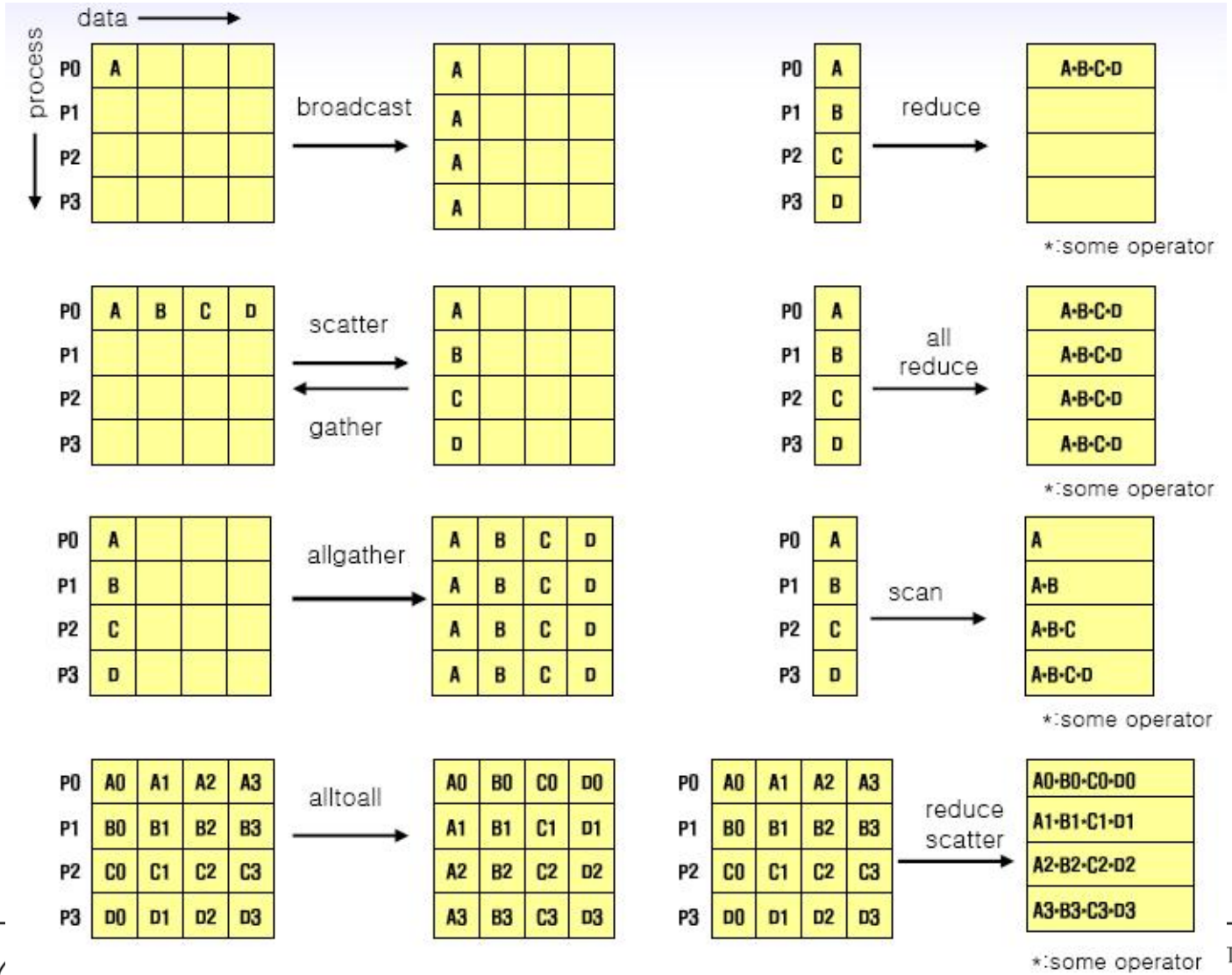
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)		
IN	sendbuf	starting address of send buffer
OUT	recvbuf	starting address of receive buffer
IN	count	number of elements in send buffer
IN	datatype	data type of elements of send buffer
IN	op	operation
IN	comm	communicator

- Applies a reduction operation and places the result in *all* tasks in the group.
- Equivalent to an MPI_Reduce followed by MPI_Bcast.

Full List of 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	Bit-wise Xor
MPI_MAXLOC	Max value and location
MPI_MINLOC	Min value and location

Collective Communication: Summary



Final Thoughts

- Before jumping headfirst into MPI, pause and consider
  - Are you implementing something others have already implemented and put into libraries
  - Should you re-use code or develop new code from the ground up
  - Map the development cycle and requirements, make sure MPI is the right answer for you

- If MPI is the right answer
  - Survey existing tools, libraries and functions, code re-use saves time, effort and frustration
  - Map application requirements, procedures and algorithm to existing parallel/ distributed algorithms, avoid reinventing the wheel
  - Include “smart” debugging statements in your code, ideally some form of tracing to be able to track how things go wrong (because chances are, they will go wrong)

Thank you very much

lars@tacc.utexas.edu

Please participate in our survey

http://bit.ly/CSUXSEDE