Lecture 21 – MPI

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Review of Monday's Lecture
  - The fundamental concepts in MPI
- MPI Hello World
- Point-to-point communication
- Collective communication

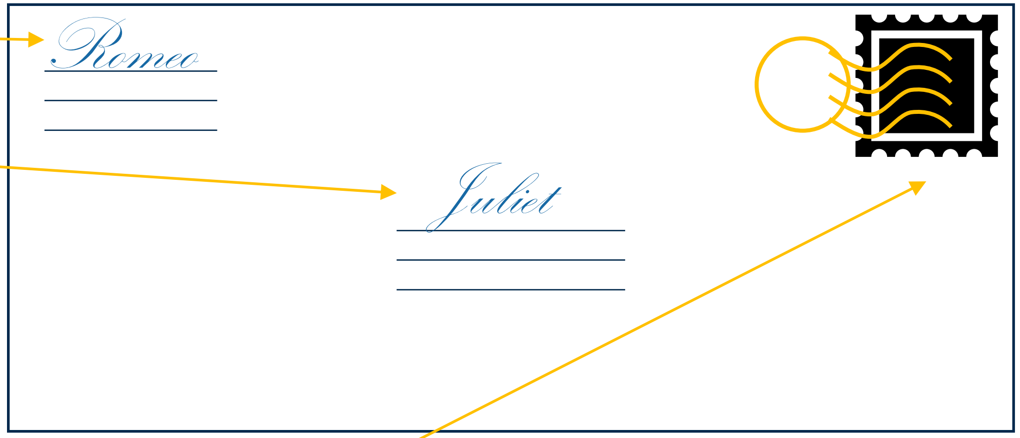
Learning Objectives: By the end of Today's Lecture you should be able to

- (*Skill*) Write and compile MPI programs
- (*Skill*) Analyze a simple MPI program
- (*Knowledge*) describe what an MPI collective is
- (*Knowledge*) be able to identify a "deadlock" in an MPI program

Review of MPI Concepts & Basics

MPI Concepts: Messages

- Sender
- Receiver
- Contents of message



- For MPI, there's also
  - messages need an identifier called a ***tag***
    (I get lots of letters from you Romeo, which one are you talking about?!)
  - ***Type*** of message (e.g. letter, flowers, candy)
  - ***Size*** of message (e.g. 10 pages, a dozen roses, a box of chocolates)

MPI Concepts: Communicators

- Communicators solve the problem of organizing groups and contexts
  - Groups name processes
  - Contexts are like systems of post offices (think different countries, states, zip codes)
    - These facilitate the use of software libraries

- MPI provides two default communicators
  - `MPI_COMM_WORLD`
  - `MPI_COMM_SELF`

- Capability exists to define and create your own communicators
  - But this is more advanced and we’re not going to say anything more about it.

The Fundamental MPI Routines: Send/Recv

```
MPI_Send(variable_address, size, datatype, destination, tag, communicator)
```

```
MPI_Recv(variable_address, max_size, datatype, source, tag, communicator, status)
```

status: needed for knowing what happened (e.g. did it work?)

MPI Program Basics (The original 6)

Routine	Purpose
MPI_Init	Initialize MPI
MPI_Comm_size	Find out how many processes there are
MPI_Comm_rank	Find out which process I am
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Finalize	Terminate MPI

Compiling and Running MPI Programs

Compiling

- MPI installs with "compiler wrappers"
  - These are simple programs that call your normal compilers with the extra options for compiling and linking against the MPI library.
- These are being standardized

Wrapper	Compiler
mpicc	C
mpicxx, mpic++, mpiCC	C++
mpifort, mpif77, mpif90	Fortran

Running

- `mpiexec [options] <executable>`
  - `-np` <number_of_processors>
  - `-f` <machinefile>
- Try to avoid
  - `mpirun` (deprecated)
  - `mpif77` and `mpif90` (deprecated in OpenMPI)
  - `mpiCC` (some file systems are not case sensitive)

MPI Hello World!

/gpfs/accounts/ners570f20_class_root/ners570f20_class/shared_data/Lecture21/hello.F90

Point-to-Point Communication

MPI Point-to-Point Communication Routines

- Point-to-Point communication involves 2 processors.
- Basic calls:

```
MPI_Send(variable_address,
         size,
         datatype,
         destination,
         tag, communicator)

MPI_Recv(variable_address,
         max_size,
         datatype,
         source,
         tag, communicator, status)
```

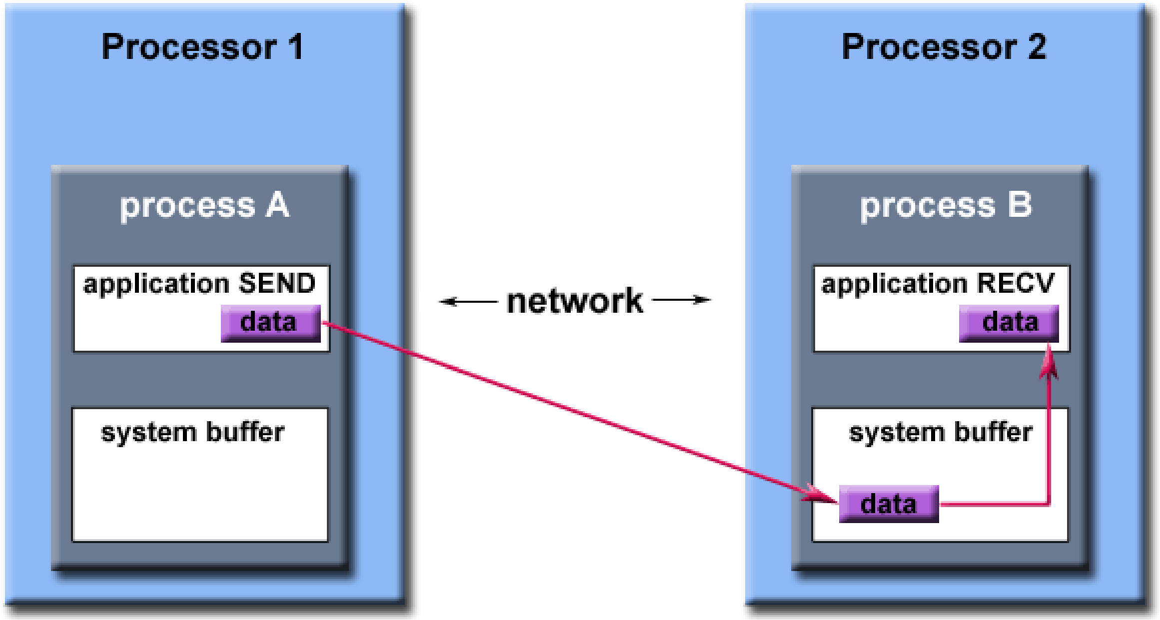
- Many variations (communication modes):
  - **Standard** mode – a send will not block even if a receive for that message has not occurred (except for lack of resources, e.g. out of buffer space at sender or receiver)
  - **Buffered** mode (MPI_Bsend) – same as standard mode, except return is always immediate, i.e., returns an error code as opposed to waiting for resources)
  - **Synchronous** mode (MPI_Ssend) – will only return when matching receive has started. No extra buffer copy needed, but can't do any computation while waiting.
  - **Ready** mode (MPI_Rsend) – will only work if matching receive is already waiting. Best performance, but can fail badly if not synchronized.
  - **Immediate** mode (MPI_Isend, etc.) – starts a standard-mode send but returns immediately. No extra buffer copy needed, but the sender should not modify any part of the send buffer until the send completes.
  - Also a combined **sendrecv**

MPI Point-to-point communication

- So many choices, which one is best?
  - The standard `send` and `recv` are good for learning MPI, but are generally not used in production application codes.
  - Buffered `send` and `recv` require more effort on the part of the application programmer to manage the buffer.
  - Synchronous send and recv are often same as standard send and recv
- Some form of non-blocking `send` and `recv` is often best for performance.
  - `MPI_Isend` and `MPI_Irecv`
  - Does require additional checking for completion
  - There are limits to the number of simultaneous messages

Message Buffers

- An MPI implementation may (not the MPI standard) decide what happens to data in these types of cases.
  - Typically, a **system buffer** area is reserved to hold data in transit.
- System buffer space is:
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - Able to exist on the sending side, the receiving side, or both
  - Something that may improve program performance because it allows send - receive operations to be asynchronous.



Path of a message buffered at the receiving process

MPI Ping Pong!

/gpfs/accounts/ners570f20_class_root/ners570f20_class/shared_data/Lecture21/pingpong.c

Collective Communication

MPI Collectives (1)

- These involve all MPI processes in a *communicator*
- Collectives can always be implemented with point-to-point routines
  - But it is often better to use the routines provided by MPI
- Common collective operations include:
  - Broadcast
  - Reduce
  - Scatter
  - Gather
  - Scan
  - Alltoall

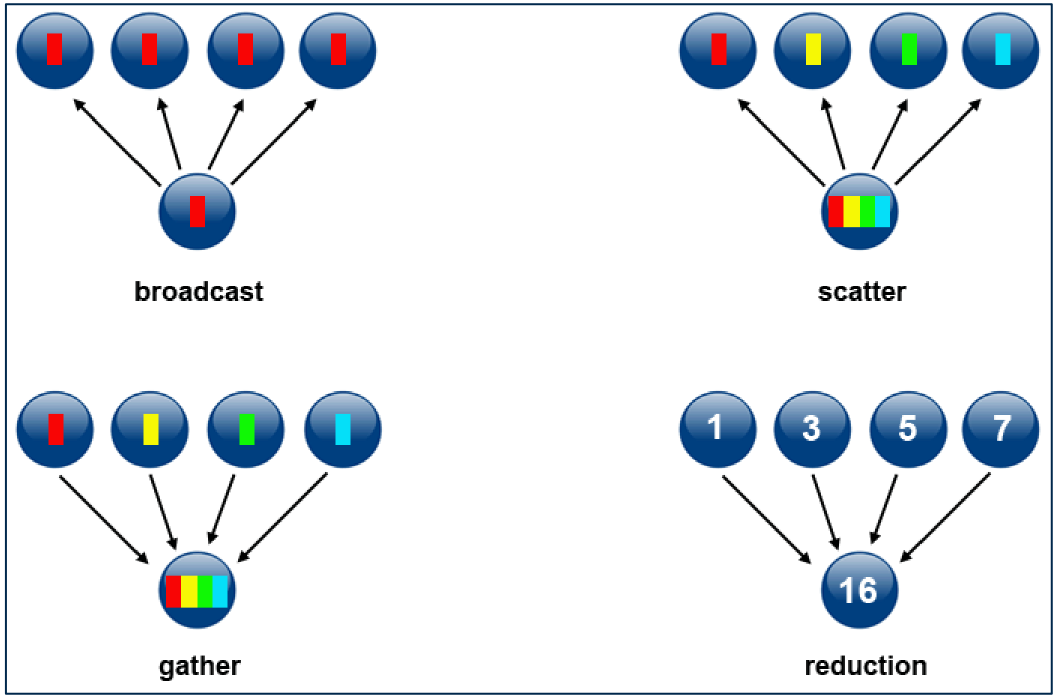


Figure from: https://computing.llnl.gov/tutorials/parallel_comp/

MPI Collectives (2)

Notable Variations

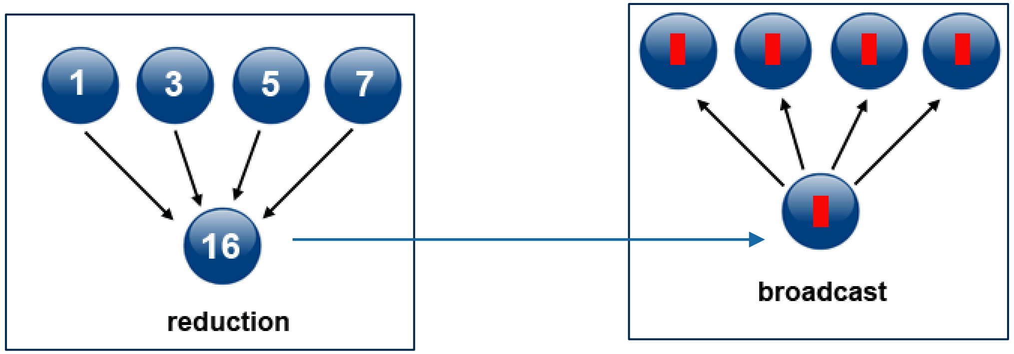
- The "`v`" suffix
  - Stands for vector
  - Means the *size of data may be different* for different processors
  - Gatherv & Scatterv, Alltoallv
- The "`All`" prefix
  - Means the *result of the operation is the same for all processors* in communicator
  - Allreduce & Allgather

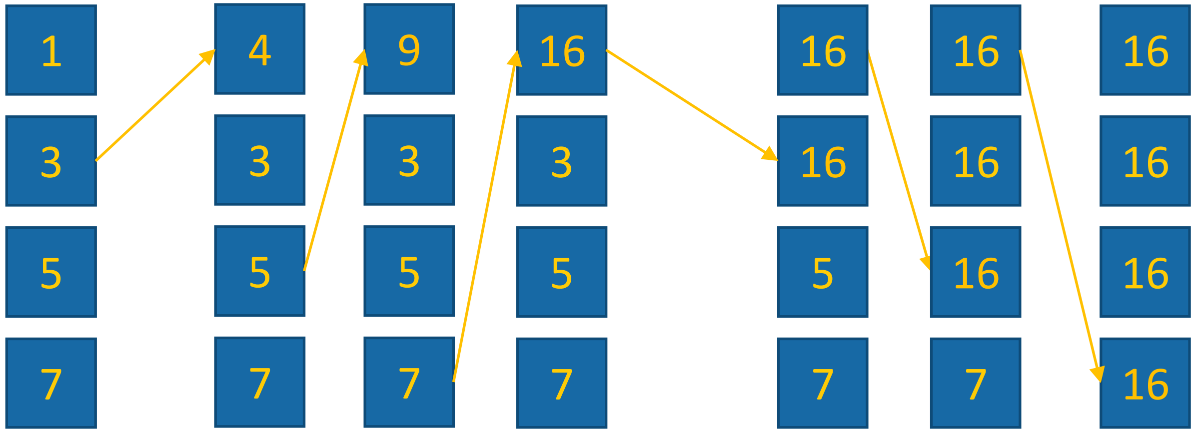
Types of reduction operations

- Arithmetic
  - MPI_SUM
  - MPI_PROD
- Relation Operators (Mins & Maxes)
  - MPI_MAX
  - MPI_MIN
  - MPI_MAXLOC
  - MPI_MINLOC
- Logical Operators
  - MPI_LAND
  - MPI_LOR
  - MPI_LXOR
- Bit-wise operators also supported

Example: MPI_Allreduce Algorithm

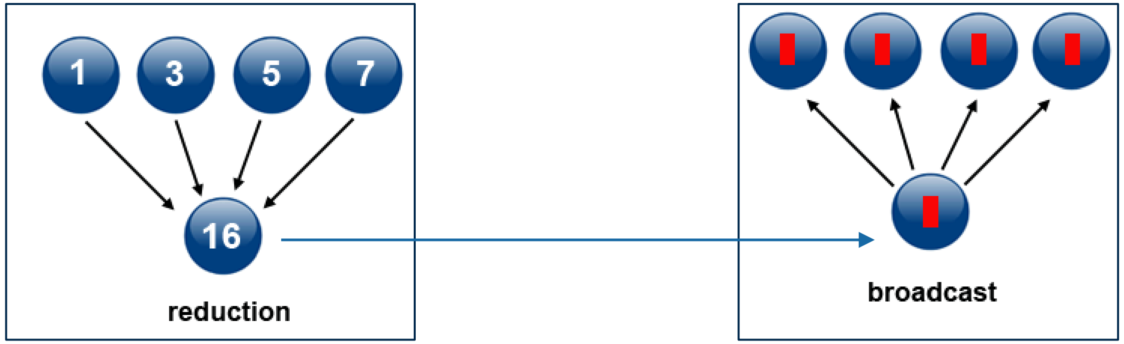
- Reduce + broadcast
- Reduce performed sequentially
  - P-1 steps
- Broadcast performed sequentially
  - Also P-1 steps
- Total of 6 steps

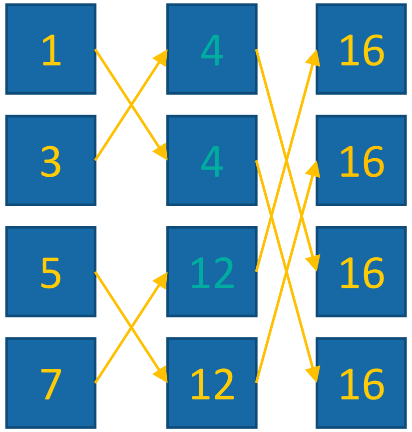




Example:Better Allreduce

- Use a binomial tree
  - Completed in $\lceil \log p \rceil$ steps
- Scales much better to higher number of processors





Even More Advanced Allreduce

- What about long messages?
  - Reduce_scatter + Allgather
- Different algorithms perform better under certain conditions

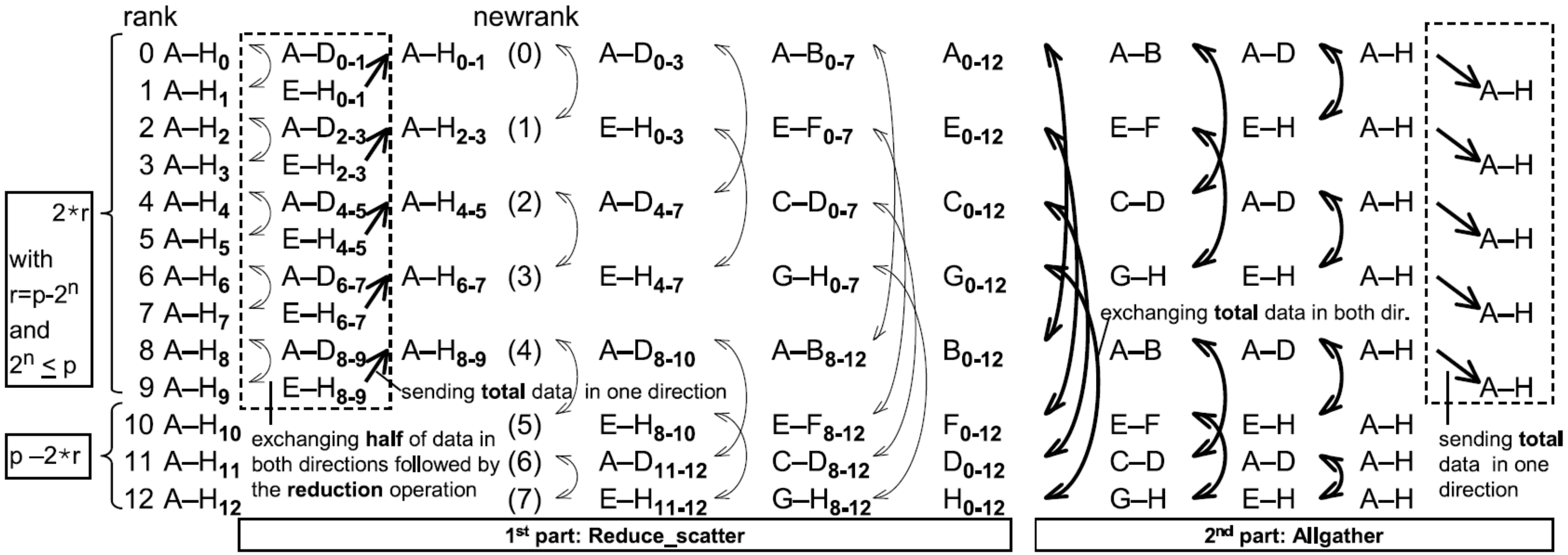


Figure 12: Allreduce using the recursive halving and doubling algorithm. The intermediate results after each communication step, including the reduction operation in the reduce-scatter phase, are shown. The dotted frames show the additional overhead caused by a non-power-of-two number of processes.

Source: http://www.mcs.anl.gov/~thakur/papers/ijhpca-coll.pdf

Summary of Collectives

- Provided as a convenience to the programmer
  - Collectives perform “common” operations that arise in programming
  - Often implemented with more complex and higher performing algorithms
    - Than what a beginner would implement.
- They represent a synchronization point in the program
- Always, always, always involves all processors *within communicator*
  - Otherwise, it causes a deadlock

Deadlock

Problem

- Symptoms
  - Code will run for a while
  - Then code will "hang".
  - Code just sits... and sits... and sits.

```
IF(MOD(myRank,2) == 0) THEN
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, mpierr)
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
ELSE
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, mpierr)
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
ENDIF
IF(MOD(myRank,2) == 0) &
  CALL MPI_Reduce(sbuf,rbuf,n,MPI_DOUBLE_PRECISION, MPI_SUM, &
    0, MPI_COMM_WORLD, mpierr)
```

Solution

- Investigate where your calls to communication are made.
  - Usually will happen around branching constructs.
- Think about how it would execute with 2 processors.

```
IF(MOD(myRank,2) == 0) THEN
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, mpierr)
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
ELSE
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, mpierr)
ENDIF
```