Introduction to Parallel Programing with MPI
Part 1

by

Prasad Maddumage
Research Computing Center
Florida State University

October 03, 2017

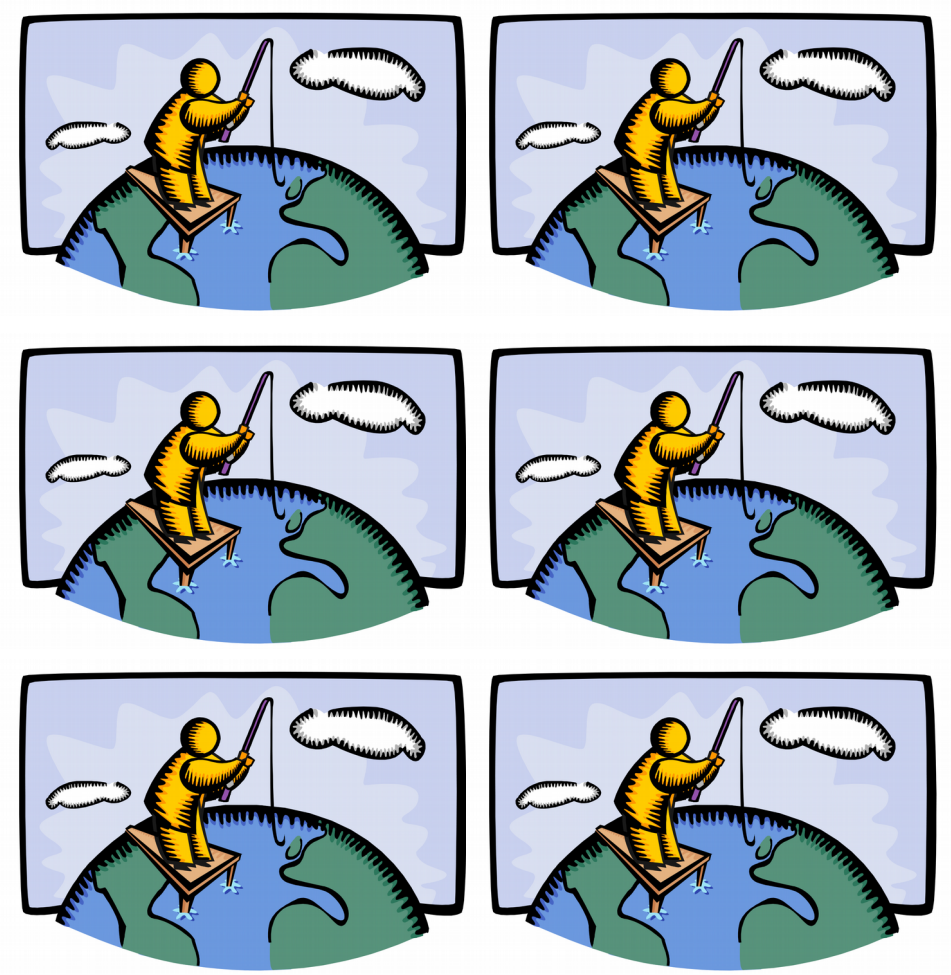
Outline

- Part 1
  - Basics of parallel programming
  - Basics of MPI
  - Point to point communication
  - Blocking vs. non-blocking calls
  - Collective communication
- Part 2
  - MPI file I/O
  - Custom data types
  - Communicators
  - One sided communication

Parallelism

Parallelism means doing multiple things at the same time: you can get more work done at the same time

Less Fish



More Fish

Terminology

- **Threads**: Execution sequences that share a single memory area (address space)
- **Process**: An execution sequence with its own address space
- **Shared memory parallelism/ multithreading:** parallelism via multiple threads
- **Distributed memory parallelism/ multiprocessing**: parallelism via multiple processes
- **High Performance Computing** (**HPC**): Combining the computing power of multiple machines to solve larger problems

Distributed Parallelism

- Suppose you want to do a jigsaw puzzle of 1000 pieces
- Lets assume it take you one hour to finish

Distributed Parallelism

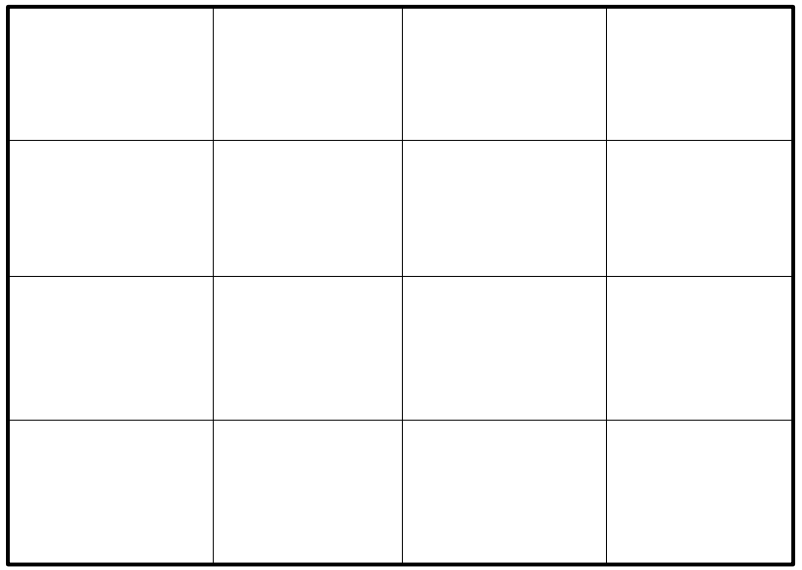
- Now, we divide the 1000 pieces into two tables
- Let Casey work in the other table on his half of the puzzle
- Both of you work independently: up to 2X speedup
- Communication at the end is expensive: you need to move two tables

Distributed Parallelism

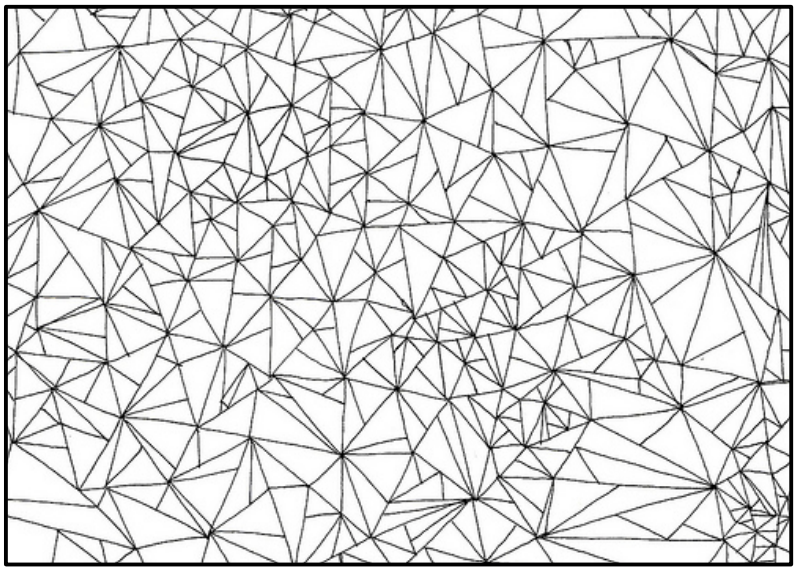
- More people joins!
- You can solve the puzzle faster
- Initial splitting of puzzle pieces may be harder (**load balancing**)
- Need more communication to finish the puzzle (**communication overhead**)

Load Balancing

- Distribute the work evenly among workers
  - Processes should not wait for others to finish: less efficient
- If a problem scales linearly, load balancing is easy
- Load balancing can be easy/hard depending on the problem



Easy



Hard

Communication Overhead

- Input data, parameter initialization, etc. need to be communicated to each node
- Outputs generated by each node should be gathered
- Communications during the calculation stage
  - Information about “boundaries” (eg. weather simulations)
  - N-body codes needs position of all other particles to calculate the total force on one particle
- There are ways to programatically minimize communication overhead in addition to use faster interconnects (hardware)

Why bother with all this?

- This seems like a lot of extra work!
- Why bother?
  - Done right, HPC can run your code much faster (few hours compared to week(s) on your desktop!) → get more done
  - You can solve bigger problems → better, exciting science
  - Today's HPC will be your desktop in 10 to 15 years! → stay ahead of your time

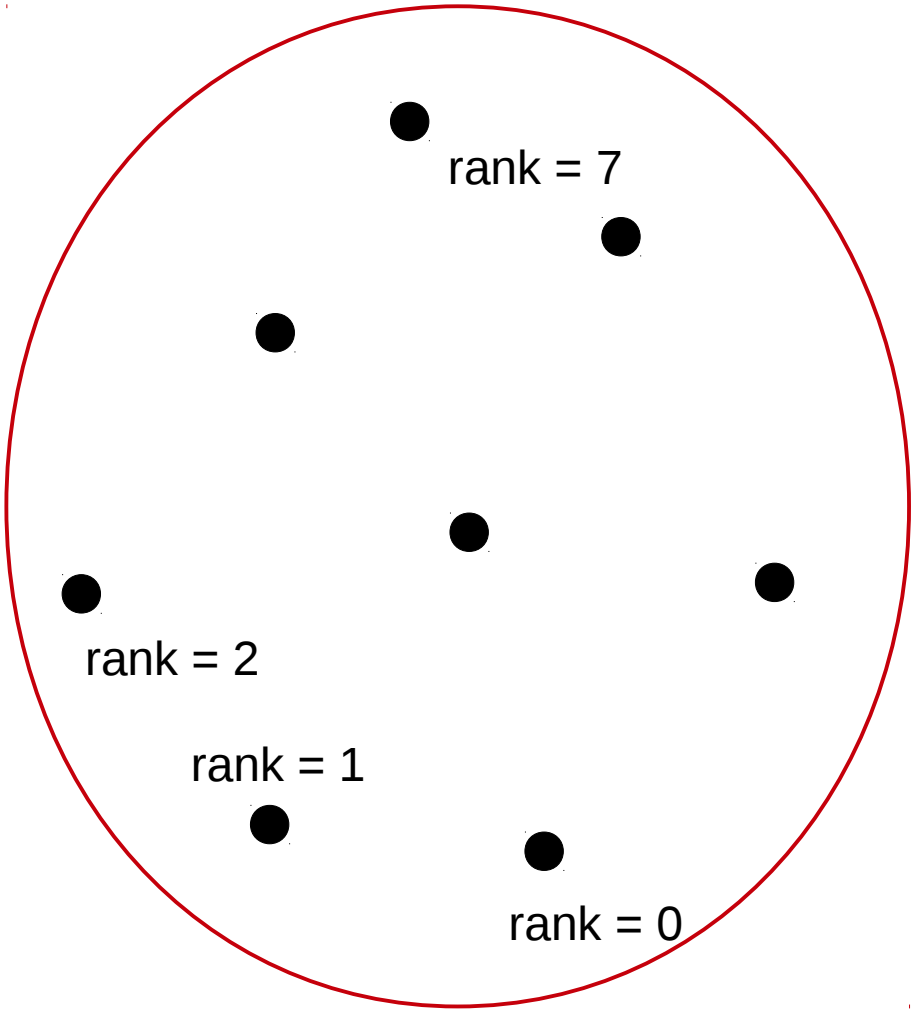
MPI
The Message Passing Interface

What is MPI?

- MPI is a language-independent communications protocol
  - Implements the framework (software) for communication between processes
- MPI is an API (Application Programming Interface)
  - Only the structure and behavior of each routine is defined
  - Implementation of routines are platform specific
- Different implementations: **OpenMPI**, MPICH, **MVAPICH2**, Intel MPI, …
- MPI Consists of a **header file**, set of **library routines**, and a **runtime environment**
- An MPI code usually has **server (master)** and **client (slave)** sections
  - Only one node executes server sections
  - All nodes except server runs client sections

MPI basics

- Start and stop
  - MPI_Init
  - MPI_Finalize
- Environment awareness
  - MPI_Comm_size
  - MPI_Comm_rank



Communicator (MPI_COMM_WORLD)

MPI Program Structure (F90)

```
program my_mpi_test
    IMPLICIT NONE
    include “mpif.h”
    [other includes]

    integer :: my_rank, n_procs, mpi_err
    [other declarations]

    call MPI_Init(mpi_err)
    call MPI_Comm_Rank(MPI_COMM_WORLD, my_rank, mpi_err)
    call MPI_Comm_size(MPI_COMM_WORLD, n_procs, mpi_err)

    [do your work here]

    call MPI_Finalize(mpi_err)
end program my_mpi_test
```

header file

MPI Program Structure (C)

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

Int main (int argc, char* argv[])
{
    int my_rank, n_procs, mpi_err;
    mpi_err = MPI_Init(&argc, &argv);
    mpi_err = MPI_Comm_Rank(MPI_COMM_WORLD, &my_rank);
    mpi_err = MPI_Comm_size(MPI_COMM_WORLD, &n_procs);

    [do your work here]

    mpi_err = MPI_Finalize();
}
```

Optional list of arguments to be sent to all processes

Compiling

- GNU, Intel, and PGI OpenMPI and MVAPICH2 compilers available at HPC
- Use modules to load selected compiler
  - Eg: `module load gnu-openmpi`
- MPI compiler wrapper scripts are used for compiling

Language	Script	Compiler
C	mpicc	gcc/icc/pgcc
C++	mpiCC mpic++ mpicxx	g++/icpc/pgCC
Fortran	mpif90	gfortran/ifort/pgf90
	mpif77	g77

Point to point calls

- MPI_SEND
  - (F) MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)
  - (C) MPI_Send (&buf, count, datatype, dest, tag, comm)

Data to be sent

Number of elements (Integer)

Destination (Integer)

Communicator

Tags (Integer) should match between send and receive

- MPI_RECV
  - (F) MPI_RECV(buf, count, datatype, source, tag, comm, status, ierr)
  - (C) MPI_Recv(&buf, count, datatype, source, tag, comm, &status)

Stores source & tag information

Point to point calls

- MPI_SENDRECV
  - (F) MPI_SENDRECV(sendbuf, sendcount, sendtype, dest,
    sendtag, recvbuf, recvcount, recvtype,
    source, recvtag, comm, status, ierr)
  - (C) MPI_Sendrecv(&sendbuf, sendcount, sendtype, dest,
    sendtag, &recvbuf, recvcount, recvtype,
    source, recvtag, comm, &status)
  - Can be used with individual SEND/RECV calls or another SENDRECV command

MPI Data types

- Type of data sent or received should be explicitly specified

C Data Types		Fortran Data Types	
MPI_CHAR	char	MPI_CHARACTER	character
MPI_INT	int	MPI_INTEGER	integer
MPI_FLOAT	float	MPI_REAL	real
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision

- Above is not a complete list. MPI standard also allows creation of user defined data types.

MPI Hello World

Codes available at
`https://github.com/prasadhmd/MPIworkshop/tree/master/part1`

Determinism

- MPI is nondeterministic: the arrival order of messages sent from two processes, A and B, to a third process, C, is not defined
- **`Source`** variable determines from which process a message is received. Using **MPI_ANY_SOURCE** as the source will let it receive from any sender
- **`Tag`** variable specifies which message from a given source to receive. Using **MPI_ANY_TAG** as the tag will let it receive messages with any tag

Deadlock Conditions

```
if (rank == 0) {
      MPI_Send(..., 1, tag1, MPI_COMM_WORLD);
      MPI_Recv(..., 1, tag2, MPI_COMM_WORLD, &status);
 } else if (rank == 1) {
      MPI_Send(..., 0, tag2, MPI_COMM_WORLD);
      MPI_Recv(..., 0, tag1, MPI_COMM_WORLD, &status);
 }
```

- Above code will NEVER complete! SEND from server will wait for a RECV request from rank 1 process while the it waits for a RECV request from server
- Following is one possible fix

```
if (rank == 0) {
      MPI_Send(..., 1, tag1, MPI_COMM_WORLD);
      MPI_Recv(..., 1, tag2, MPI_COMM_WORLD, &status);
 } else if (rank == 1) {
      MPI_Recv(..., 0, tag1, MPI_COMM_WORLD, &status);
     MPI_Send(..., 0, tag2, MPI_COMM_WORLD);
 }
```

MPI Communication types

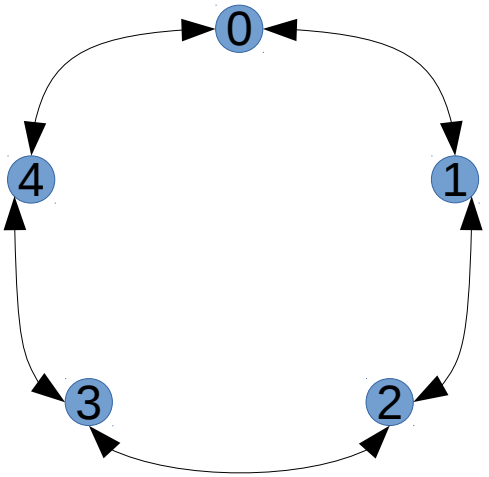
- Blocking Calls
  - A call will not “return” until it is “complete”
    - A blocking call will pause the program execution on the sender process(es) until the receiving process(es) receive the complete message
  - “Safe” but can slow down the program execution
- Non-Blocking Calls
  - Only instructs MPI to carry out the communication: No waiting for the data transfer to be complete
  - “Fast” but synchronization can be tricky

Non-Blocking point to point calls

- Immediate send
  - (F) MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierr)
  - (C) MPI_Isend(&buf, count, datatype, dest, tag, comm, &request)

    request (Integer type in Fortran, MPI_Request type in C): a handle for the communication to retrieve the status later

- Immediate receive
  - (F) MPI_IRECV(buf, count, datatype, source, tag, comm, request, ierr)
  - (C) MPI_Irecv(&buf, count, datatype, source, tag, comm, &request)



Exchange data between nearest neighbors

```
program ring
implicit none
include 'mpif.h'

Integer :: numtasks, rank, next, prev, buf(2), ierr
Integer :: stats(MPI_STATUS_SIZE,4), reqs(4)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

prev = rank - 1
next = rank + 1
if (rank == 0)  prev = numtasks - 1
if (rank == numtasks - 1) next = 0

call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, 1, MPI_COMM_WORLD, reqs(1), ierr)
call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, 2, MPI_COMM_WORLD, reqs(2), ierr)

call MPI_ISEND(rank, 1, MPI_INTEGER, prev, 2, MPI_COMM_WORLD, reqs(3), ierr)
call MPI_ISEND(rank, 1, MPI_INTEGER, next, 1, MPI_COMM_WORLD, reqs(4), ierr)

call MPI_WAITALL(4, reqs, stats, ierr)

call MPI_FINALIZE(ierr)

end program ring
```

wait until all transfers are completed
this is a **BLOCKING** command

Codes available at
`https://github.com/prasadhmd/MPIworkshop/tree/master/part1`

Non-Blocking tests

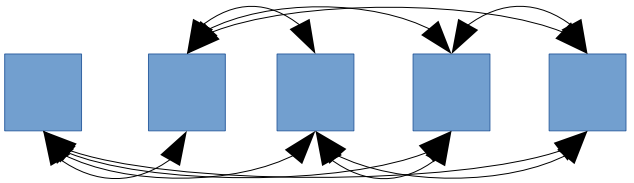
- It is sometimes useful to have a non-blocking way to check the status of a communication
  - (F) MPI_TEST(request, flag, status, ierr)
  - (C) MPI_Test(&request, &flag, &status)

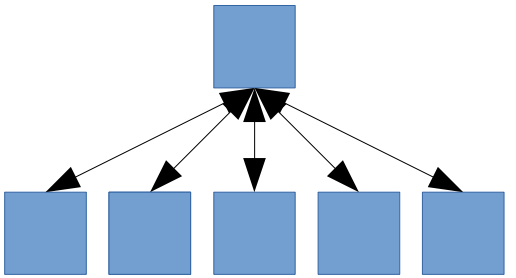
False/True depending on in/completion

- `MPI_TESTALL`, `MPI_TESTANY` are two variants of `MPI_TEST` which checks an array of communications

Collective Communication

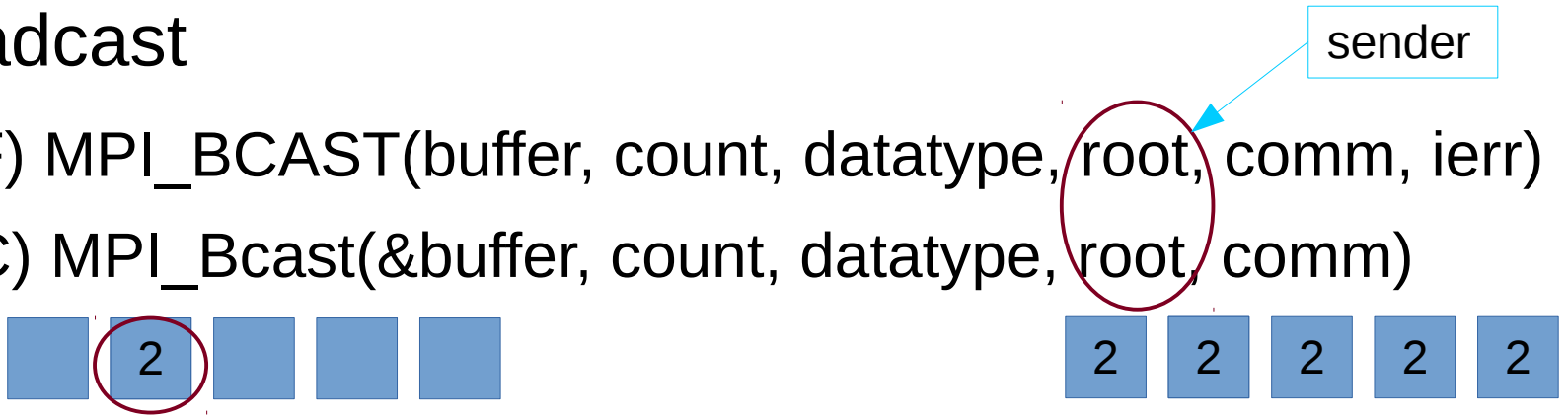
- Sometimes it is necessary to communicate between groups of processes
  - Data Movement: transfer initial values, gather individual results, etc.
  - Collective Computation: gather partial results and combine to get the final result
  - Synchronization: keep everyone in the same page
- Types of collective calls
  - One to all and all to one
  - All to all



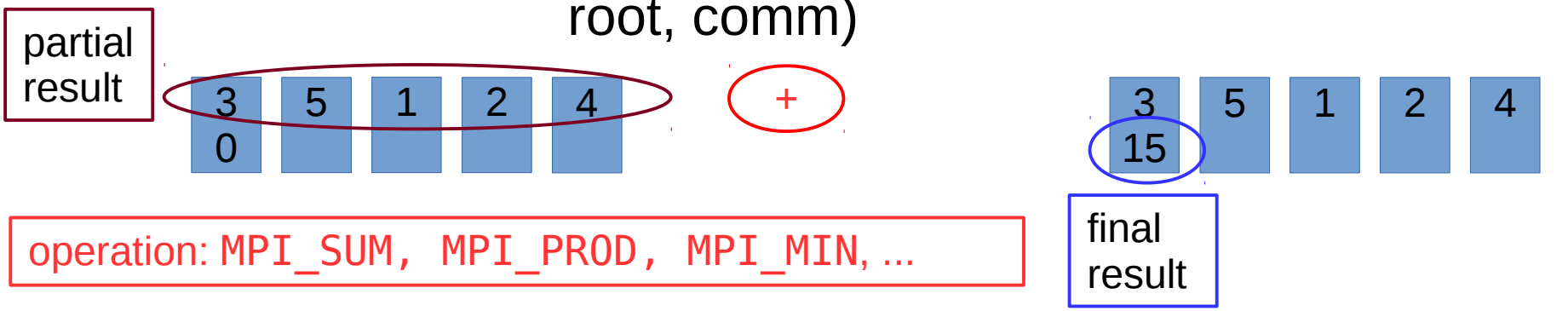


Collective Communication

- Broadcast
  - (F) MPI_BCAST(buffer, count, datatype, root, comm, ierr)
  - (C) MPI_Bcast(&buffer, count, datatype, root, comm)



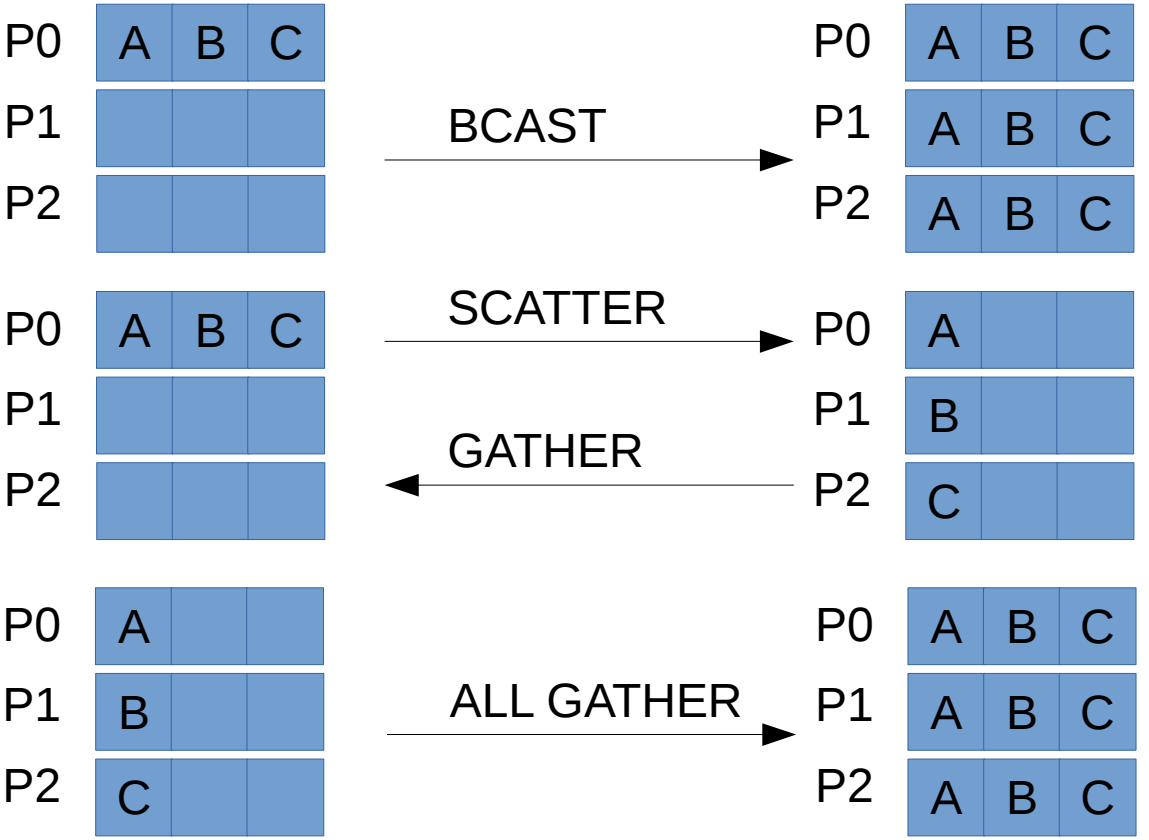
- Reduction
  - (F) MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
  - (C) MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, comm)



MPI Reduction Example

Codes available at
`https://github.com/prasadhmd/MPIworkshop/tree/master/part1`

Collective Communication



Collective Communication

- Collective communication calls can be faster than a set of send-receive calls
- Very useful for data parallelism
  - Same set of instructions operate on different sub sets of data
- There is significant communication overhead
- These are blocking calls
  - Non-blocking collective calls are also available

MPI Reduce example
MPI Scatter/Gather example

Codes available at
`https://github.com/prasadhmd/MPIworkshop/tree/master/part1`

Parallel Programming Strategies

Parallel Programming Strategies

- Client-server
- Data Parallelism
- Task Parallelism
- Pipeline

Client-Server

- Server (master) decides what clients (slaves) do
- **Embarrassingly parallel**: The problem can easily be broken into roughly equal amounts of work per process and has very little communication (low communication overhead)
- Has near linear speedup and easy to program
- Eg: Monte Carlo methods - widely used to simulate a physical phenomena or calculate an integral
  - Randomly generate large number of samples (**realizations**) of a phenomenon/equation and take the average over all samples
  - Simulation stops when the average value converges

Client-Server example
Calculating Pi

Data Parallelism

- Each process does exactly same operations on a unique subset of data
- Most scientific problems involve calculus: solving differential equations etc.
- Numerically solving these equations over a large domain is very common
- Data parallelism can be applied to parallelize this type of problems
- Eg: CFD, Heat transfer, Weather prediction, etc.

Task Parallelism

- Each process does different operations on exactly same set of data
- Task parallelism is a widely used technique
- N body problem
  - N objects interacting with each other via forces: stars under gravity, molecules under electrostatic force etc.
  - Send properties of each object to all processes and let each process find the total force on a subset of particles
  - After each time step, use MPI_Allreduce
  - Applications: Cosmology, structural biology, machine learning

Pipeline Parallelism

- Each process does its work, passes its set of data to next process and receives next set of data from previous process
- All processes are connected to form a data pipeline
- Every process execute same tasks and results are passed to the “next” process and more data is received from the “previous” process
- Workers can be connected in a circular (closed) loop or linear (open)
- Matrix multiplication can be done in a pipeline