A Crash Introduction to Parallel Programming with MPI

Arnold Tharrington
National Center for Computational Sciences
Scientific Computing Group

October 6, 2012

Course Outline

- Motivation for parallel programming
- What is MPI?
- The MPI programming model
- MPI environmental functions
- Point to Point communications
- Collective communications
- Computing π and performance considerations

What is MPI?

- MPI stands for Message Passing Interface
- MPI is an implementation specification for parallel programming message passing libraries
  - MPI is the “de facto” standard of message passing in the HPC community

History of MPI

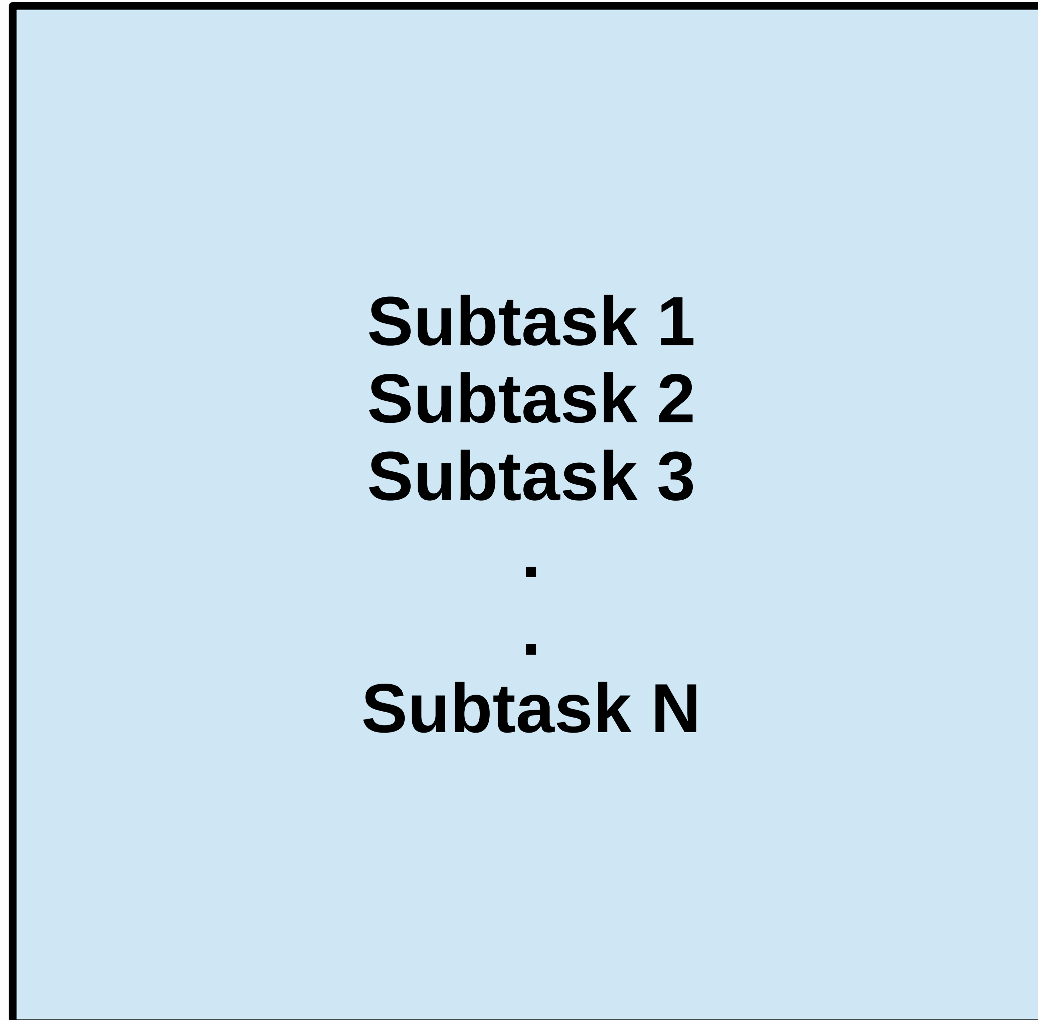
- History
  - Distributed computing develops between the 1980s and the early 1990s – a number of incompatible message passing libraries exist
  - One standard to rule them all
    - Through efforts of numerous people and organizations between 1992 and 1994, the MPI standard is created and the final draft is released in May 1994

Motivation For Parallel Programming

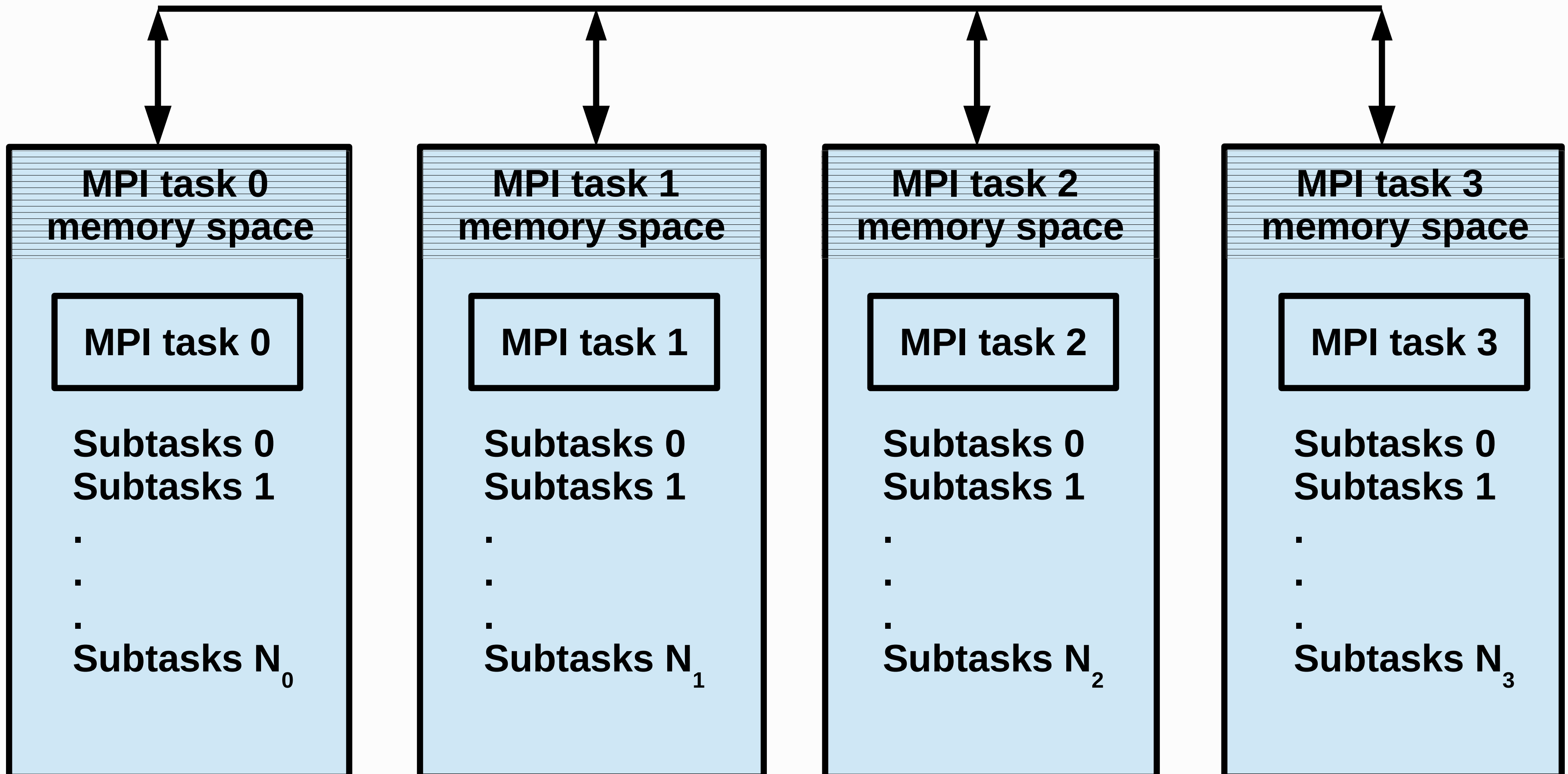
- The goal is to reduce the wall time to solution
  - Frequency Scaling
    - Limited by power consumption, $P=CV^2F$
      - P is power consumed
      - C is the switch capacitance
      - V is the supply voltage
      - F is the switching frequency
    - Intel's cancellation of Tejas and Jayhawk processors in May 2004[1]
      - Demarcates Intel's shift from single core to multi-core processors
  - Multiple processing threads
    - Divide the computational tasks between distinct processing “threads”
    - Ideally the wall time should decrease linearly with the number of execution threads

1. ***Tejas and Jayhawk*** <http://en.wikipedia.org/wiki/Tejas_and_Jayhawk>

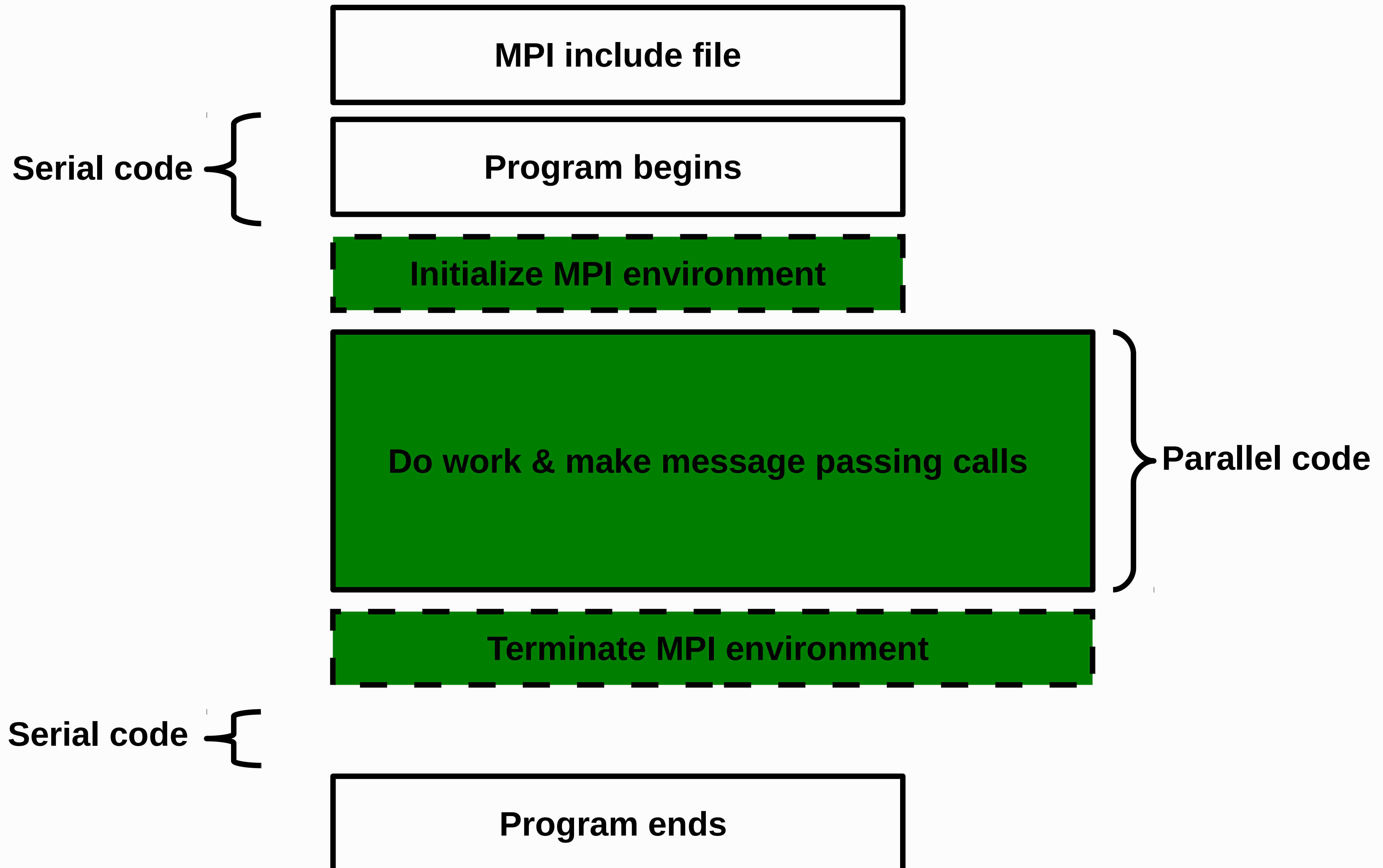
The MPI Programming Model



The MPI Programming Model (cont.)

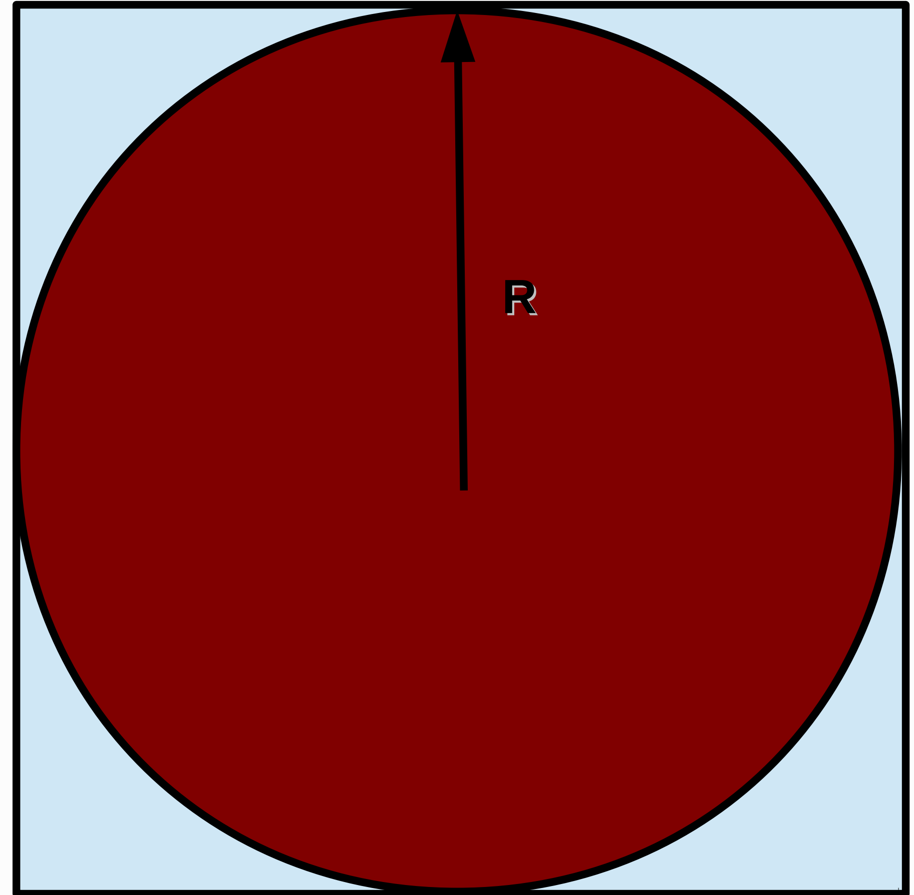


Structure of a MPI Program



Computing Pi

- If we randomly throw a dart into the square region, what is the probability of the dart landing in the red circle?
- Probability $= \frac{a_{\text{circle}}}{a_{\text{square}}} = \frac{\pi}{4}$



Computing Pi Algorithm

- Random generate a set of points, (x,y), in the blue square region

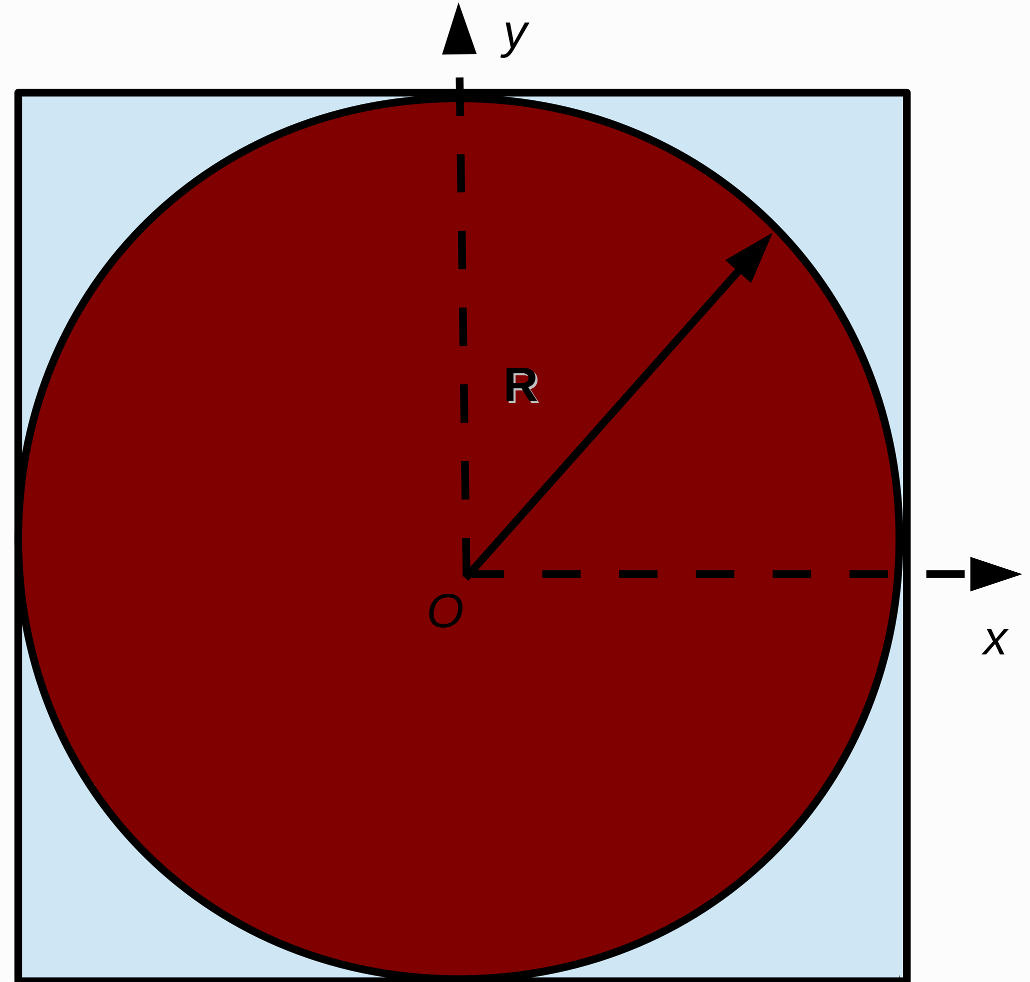
$$-R \leq x \leq R$$

$$-R \leq y \leq R$$

- Count the number of tries and the number of hits that land in the circle

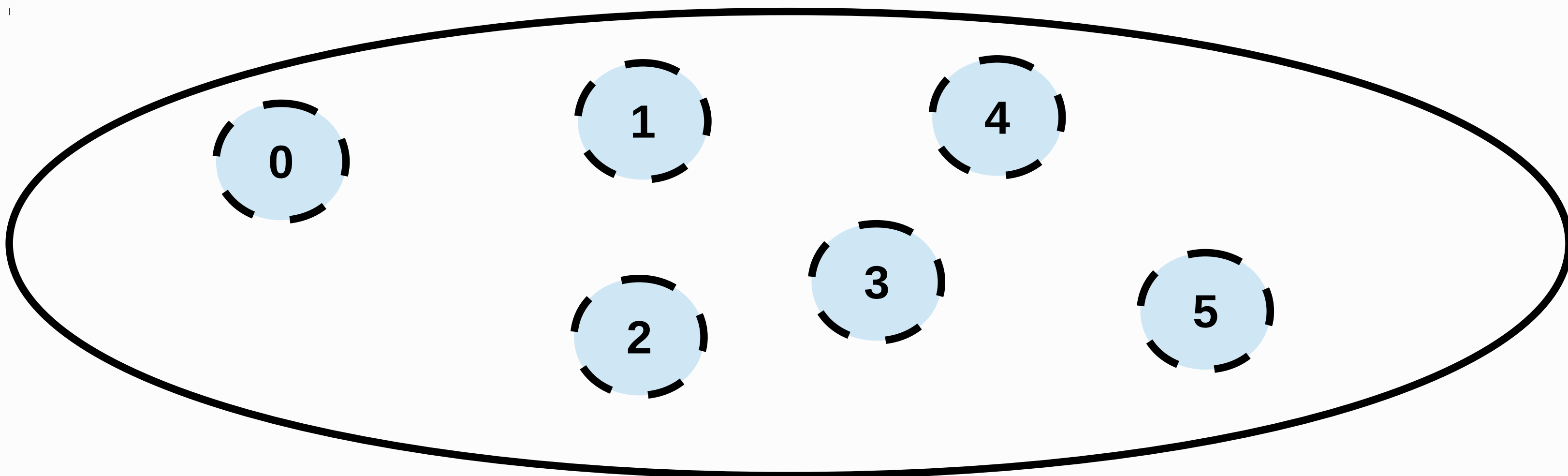
- A hit if $x^2 + y^2 \leq R^2$

- $\pi = 4 \frac{N_{\text{hits}}}{N_{\text{tries}}}$



MPI_COMM_WORLD

- The initial universe intra-communicator for all processes
  - Defined when MPI_Init(...) is called
  - Within each communicator each MPI task has its own unique id called the rank.
  - The ranks are contiguous starting from 0.



MPI Environmental Functions

- **MPI_Init**
  - Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.
  - C bindings: **MPI_Init (&argc,&argv)**
  - Fortran bindings: **MPI_INIT (ierr)**

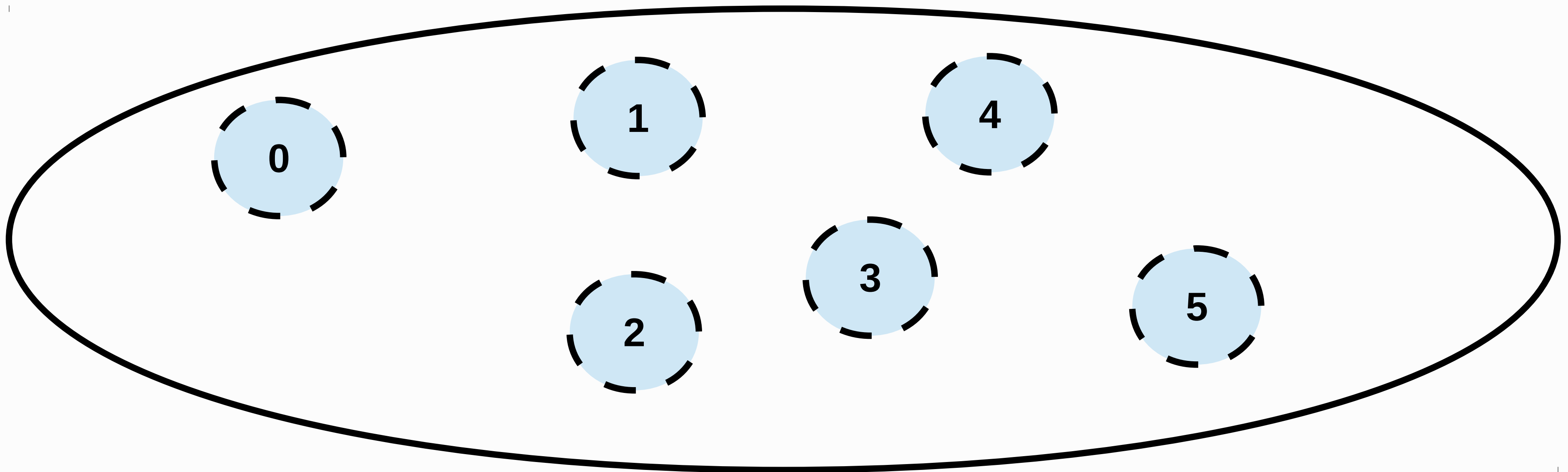
- **MPI_Finalized**
  - Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.
  - C bindings: **MPI_Finalized( )**
  - Fortran bindings: **MPI_Finalized(ierr)**

- **MPI_Comm_size**
  - Determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application.
  - C bindings: **MPI_Comm_size (comm,&size)**
  - Fortran bindings: **MPI_COMM_SIZE (comm,size,ierr)**

- **MPI_Comm_rank**
  - Determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.
  - C bindings: **MPI_Comm_size (comm,&rank)**
  - Fortran bindings: **MPI_COMM_SIZE (comm,rank,ierr)**

- **MPI_Abort**
  - Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.
  - C bindinds: **MPI_Abort (comm,errorcode)**
  - Fortran bindings: **MPI_ABORT (comm,errorcode,ierr)**

Point to Point Communications



How do we send a message from process 0 to process 1?

Point To Point Communcations

- Sender
  - Buffer
  - Data count
  - Data type
  - Destination
  - Tag
  - Communicator

MPI_Send (&buf,count,datatype,dest,tag,comm)

- Receiver
  - Buffer
  - Data count
  - Data Type
  - Source
  - Tag
  - Communicator

MPI_Recv (&buf,count,datatype,source,tag,comm,&status)

Order and Fairness

- **Order:**
  - **MPI guarantees that messages will not overtake each other.**
  - **If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.**
  - **If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.**
  - **Order rules do not apply if there are multiple threads participating in the communication operations.**

- **Fairness:**
  - **MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".**
  -
  - **Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.**

- **MPI_Send**
  - Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send (discussed below) to implement the basic blocking send.
  -
  - C bindings: **MPI_Send** (&buf,count,datatype,dest,tag,comm)
  - Fortran bindings: **MPI_SEND**(buf,count,datatype,dest,tag,comm,ierr)

- **MPI_Recv**
  - Receive a message and block until the requested data is available in the application buffer in the receiving task.
  - C bindings: **MPI_Recv** (&buf,count,datatype,source,tag,comm,&status)
  - Fortran bindings: **MPI_RECV** (buf,count,datatype,source,tag,comm,status,ierr)

- **MPI_Sendrecv**
  - **Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.**
  - **MPI_Sendrecv(&sendbuf,sendcount,sendtype,dest, sendtag,&recvbuf,recvcount,recvtype,source, recvtag,comm,&status)**
  - **MPI_SENDRECV(sendbuf,sendcount,sendtype, dest,sendtag,recvbuf, recvcount,recvtype,source,recvtag,comm,status,ierr)**
  -
  -

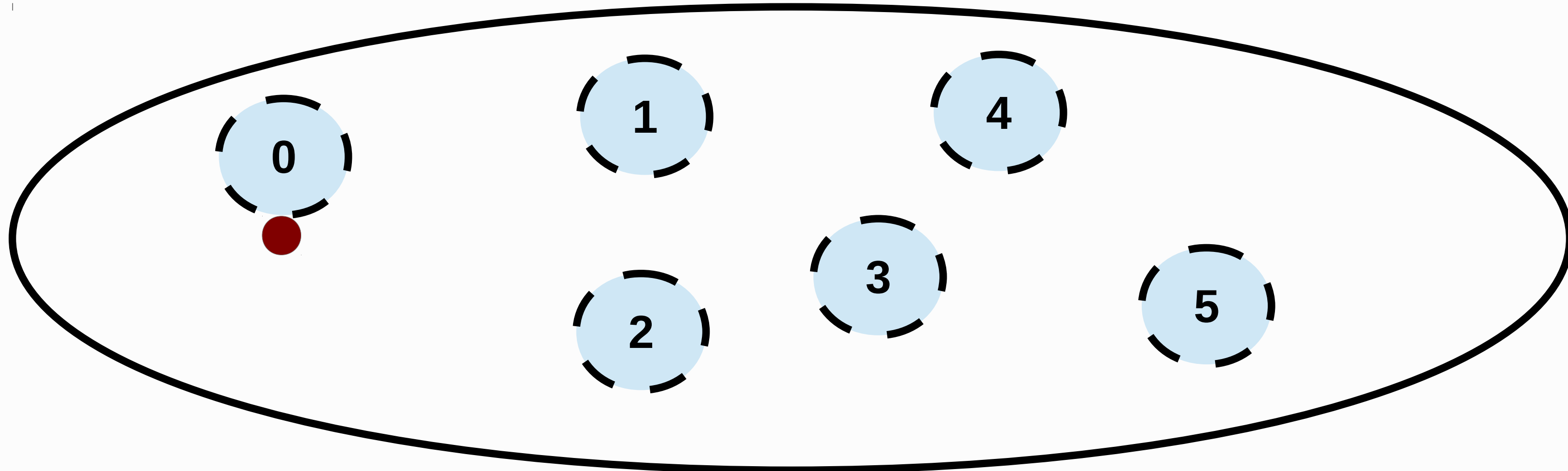
Collective Communications

- **MPI_Barrier**
  - Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call.
  - C: **MPI_Barrier** (comm)
  - Fortran: **MPI_BARRIER** (comm,ierr)

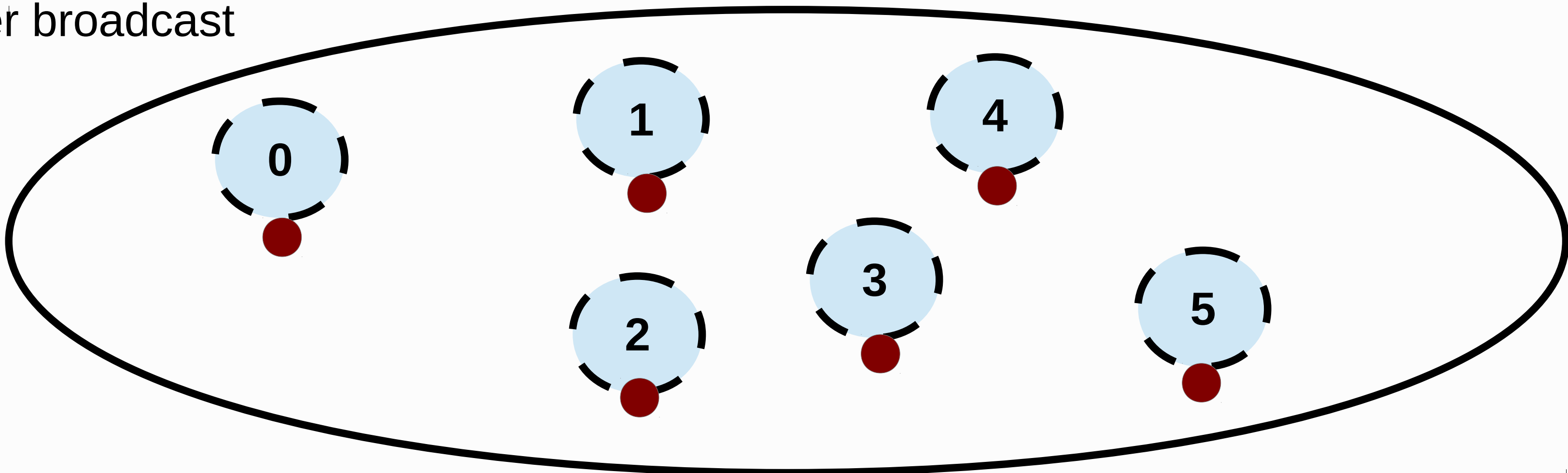
- **MPI_Bcast**
  - Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.
  - C: **MPI_Bcast** (&buffer,count,datatype,root,comm)
  - Fortran: **MPI_BCAST**(buffer, count,datatype,root,comm,ierr)

MPI_Bcast

Before broadcast



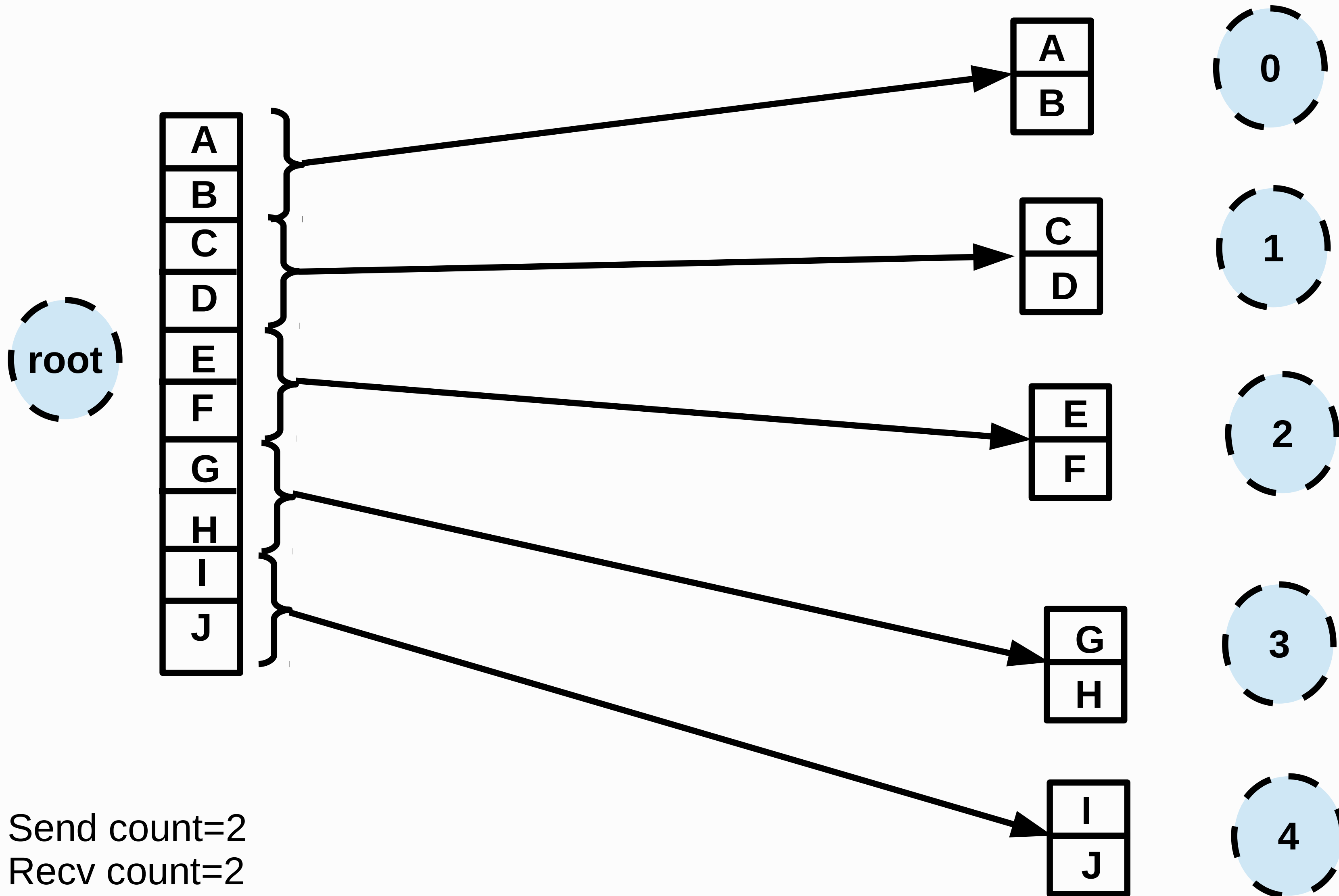
After broadcast



- **MPI_Scatter**
  - Distributes distinct messages from a single source task to each task in the group.

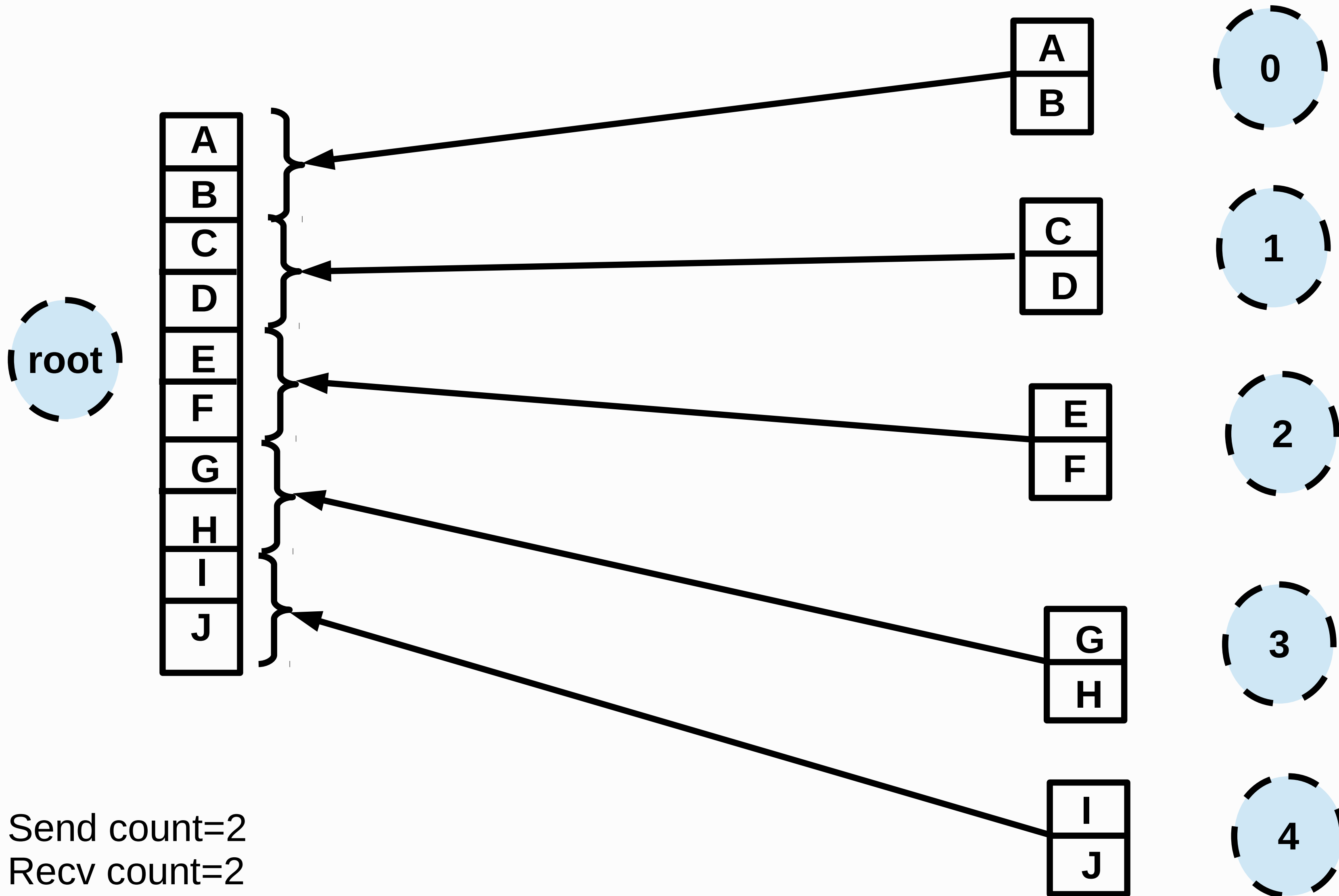
  - **MPI_Scatter** (&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,root,comm)
  - **MPI_SCATTER** (sendbuf,sendcnt,sendtype,recvbuf, recvcnt,recvtype,root,comm,ierr)

MPI_Scatter



- **MPI_Gather**
  - Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.
  - **MPI_Gather** (&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,root,comm)
  - **MPI_GATHER** (sendbuf,sendcnt,sendtype,recvbuf, recvcnt,recvtype,root,comm,ierr)

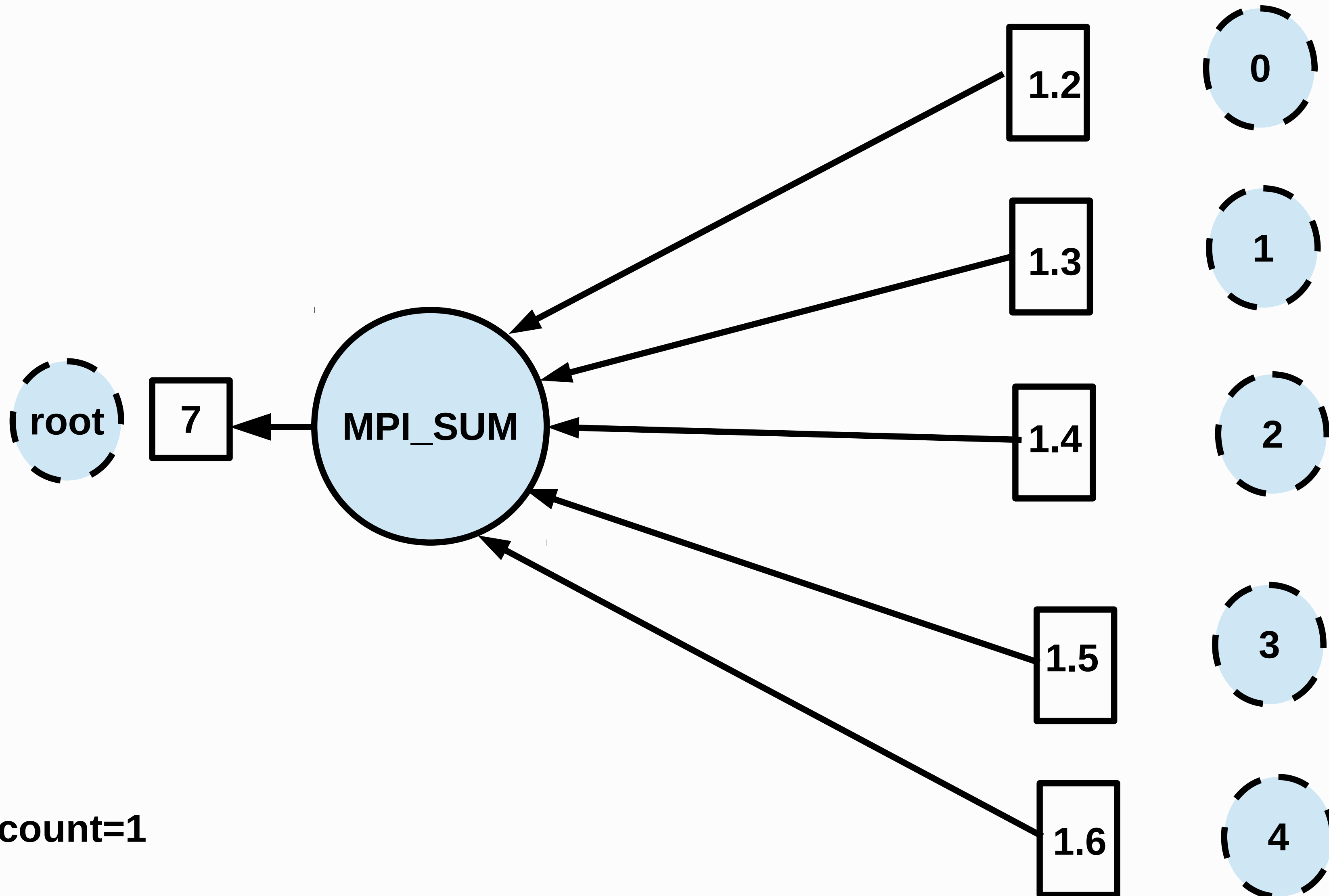
MPI_Gather



- **MPI_Reduce**
  - Applies a reduction operation on all tasks in the group and places the result in one task
  - **C: MPI_Reduce**(&sendbuf,&recvbuf,count, datatype, op, root,comm)
  - **MPI_REDUCE**(sendbuf,recvbuf,count,datatype, op, root,comm,ierr)

MPI Reduction Operation		C Data Type	Fortran Data Type
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex

MPI_Reduce



References

- Blaise Barney, ***Message Passing Interface (MPI)*** <https://computing.llnl.gov/tutorials/mpi>, February 14, 2012
- Gropp, W., Lusk E., and Skjellum, A. (1999) ***Using MPI***. Cambridge, Massachusetts: The MIT Press
- ***The Message Passing Interface (MPI) Standard*** <http://www.mcs.anl.gov/research/projects/mpi>