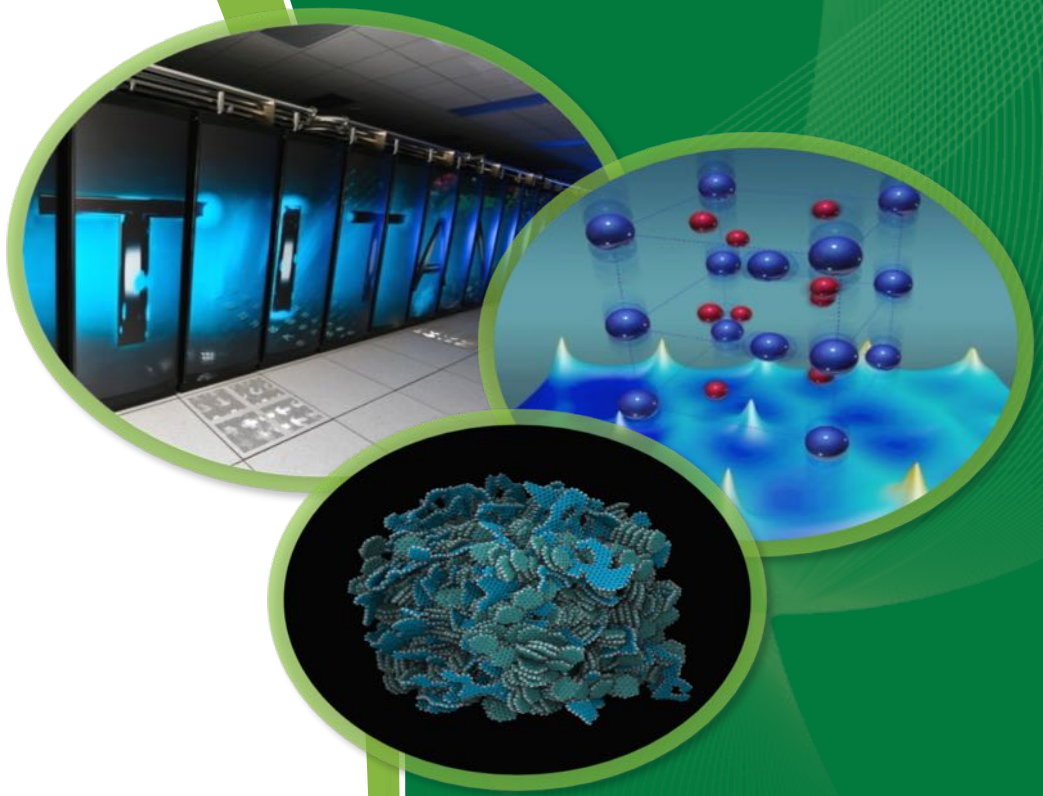
Introduction to HPC Workshop – Introduction to MPI

Brian Smith

smithbe@ornl.gov



ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Topics

- Background
- “2-sided” Point-to-point communications
- Collective Communications
- Other major MPI features
- Examples, Hands-on

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Note

- MPI is big. Even ”basic” MPI has a lot of complexities
  - This talk tries to stick with the most useful/most frequently used pieces of MPI
  - It’s still a lot of content, especially for an hour in an introductory HPC workshop
  - Try to point out “gotchas” and use cases/examples of frequently used operations

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

History/Background/Intro

- MPI - “Message Passing Interface”
- A definition for an API or library, NOT a specific implementation
- MPI 1.0 Standard – 1994
  - Many commercial and a few opensource implementations developed
- MPI 2.0 – 1997
  - Major additions: Added MPI I/O, RMA (one-sided), dynamic processes, F90 and C++ bindings
- MPI 1.3/2.1 – 2008 (after 10 year hiatus) – Mostly clarifications/errata
- MPI 3.0 – 2012
  - Major additions: Nonblocking collectives, better (usable) one-sided operations, F2008 bindings
  - Major deletions: Remove C++ bindings
- MPI 3.1 – 2015 – Mostly clarifications/errata, nonblocking IO routines
- MPI 4.0 - ? 2019 maybe?

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Learning MPI

- Lots of web tutorials exist
- Tutorials at Supercomputing conference each year
- Books
  - MPI: The Complete Reference (2 volume set) (primarily covers MPI1.x)
  - Using MPI2: Advanced Features of the Message Passing Interface
  - Using MPI: Portable Parallel Programming with the Message-Passing Interface
    - Third edition covers MPI 3.0 features
- MPI Reference
  - Standards document: www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
  - Available as a printed book from HLRS in Germany, via Amazon
  - Primarily for implementors but useful as a reference

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

What is it and Why?

- Distributed memory model
  - Provides mechanisms to move data among disjoint processes
  - Can still be used within a node, but other strategies might be better (e.g. OpenMP)
- Requires explicit code for parallelism
  - No magic from the compiler
  - No transparent large arrays spanning processes for example

- Why should I use MPI?
  - Standardized - All HPC vendors support MPI; most scientific/HPC libraries support MPI; most parallel codes use MPI
  - Portable - MPI defines an API, so as long your code is MPI compliant and your implementation is too, your **MPI parts** should be portable
  - Functionality - Well over 400 routines
  - Performance - Implementations are encouraged to optimize for performance

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Current Implementations

- 2 Major open source Implementations
  - MPICH from Argonne
  - OpenMPI – merge of FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI in the mid 2000s
  - Both will run fine on a laptop or cluster
- Several current commercial implementations
  - IBM Spectrum MPI (Summit) – OpenMPI derivative
  - IBM BlueGene MPI – MPICH derivative
  - Cray MPI (Titan) – MPICH derivative

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

High-level MPI Functionality (in rough order of frequency of usage)

- Init/Finalize, Point-to-point message passing, Process Groups/Communicators, Collective message passing

Less common:

- Parallel I/O
- Tools interface
- MPI 3.x One-sided Message Passing
- Derived Datatypes

Very uncommon:

- MPI 2.x One-sided Message Passing
- Dynamic processes

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

A word about function prototypes and Fortran

- MPI is implemented in C (…typically)
  - Fortran interfaces are wrappers into C calls (…typically)
  - All routines have Fortran interfaces available
- Examples here are in C
- Fortran prototypes can be found in the standard, in man pages, and via a google search
- #include<mpi.h> for C programs
- For Fortran - “include mpif.h” or “use mpi” or “use mpi_f08”
  - Subtle differences between them; see the standard for details.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Compiling MPI Code

- Typically, some sort of compiler wrapper that links in all the required libraries
- Cray (Titan)
  - cc – C Wrapper for automatically including MPI and tons of other parallel environment libraries.
  - CC – The C++ wrapper
  - ftn – The Fortran wrapper (77, 90, 08, etc)
  - Underlying compiler is set by whatever PrgEnv you have loaded
- Spectrum MPI (Summit) and generic MPICH and OpenMPI
  - mpicc – The C wrapper
  - mpic++, mpiCC – The C++ wrappers
  - mpifort, mpif77, mpif90 – The Fortran wrappers
  - Underlying compiler is set by whatever compiler module is loaded

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Init()/MPI_Finalize()

- Before calling any useful MPI routines, a program needs to call MPI_Init() or MPI_Init_thread().
- Int MPI_Init(int *argc, char ***argv)
- Int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
  - The ”*required*” argument is what the program desires. The value returned in *provided* is what the implementation/system/etc can actually provide
  - Overheads can be somewhat higher with MPI_THREAD_MULTIPLE
- int MPI_Finalize – Called at the end of any MPI usage.
  - No useful MPI calls can come after MPI_Finalize()

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Point to Point Data Movement

- MPI provides four variants on send with blocking and nonblocking versions of each.
- Blocking means the call will not complete until the local data is safe to modify
  - It could be moved into an MPI internal temporary buffer, it could be in a buffer on the network card, it could even be at the remote already (but NOT guaranteed)
- Nonblocking means the call returns “immediately”
  - Nonblocking data movement calls in MPI are MPI_I{command}, e.g. MPI_Irecv() or MPI_Ialltoallv() (capital “Eye”)
  - Nonblocking calls require a mechanism to tell when they are done – MPI_Wait*, MPI_Test*
  - Data may or may not actually move before a call to MPI_Wait*/MPI_Test*
  - It is not safe to reuse buffers until the Wait/Test says the operation is locally done.
  - Nonblocking calls (can) allow for compute and communication to overlap

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Point-to-Point Message Passing

- MPI_Send – Basic, blocking send. Moves data from calling process to a destination process. Program progress stops at the LOCAL side until the call completes (data is moved to network buffers for example). Data on A can be changed once the call completes.
- MPI_Isend – Basic, nonblocking send. Moves data from calling process to a destination process. Program execution continues “immediately”. Data can’t be touched until a Wait*() or Test*() call says the request is complete
- MPI_Bsend/Ibsend – Buffered send. Requires providing a buffer with MPI_Buffer_attach().
- MPI_Rsend/Irsend – Ready send. The programmer has promised the matching receive has already posted on destination.
- MPI_Ssend/Issend – Synchronous send. Waits until the receive has been posted on the receive side before completing on the send side.
- MPI Provides a blocking receive and a nonblocking receive.
  - All send() variations can be matched by a receive or nonblocking receive (no MPI_Srecv for example)
  - Can mix and match blocking/nonblocking

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Send/Isend()

- int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
- int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request)

- buf – the source buffer you want to transfer
- count – the number of elements you want to transfer
- type – the type of the elements you want to transfer (MPI_INT, MPI_DOUBLE, my_derived_mpi_type, etc)
- dest – The rank of the recipient of the data
- tag – An identifier for this particular send. Used to differentiate messages
- comm – The group of processes for which *dest* is a member (more later)
- request – An MPI_Request object which can be used to determine when the send() is complete via MPI_Wait*()/MPI_Test*() functions

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Recv/Irecv()

- int MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status)
- int MPI_Irecv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request*request)

- buf – the buffer to place data in
- count – the number of elements of type type to receive
- type – the MPI datatype (MPI_INT, etc)
- source – The originator of the data. Can be MPI_ANY_SOURCE
- tag – An identifier for a particular message. Can be MPI_ANY_TAG
- comm - The group of processes for which *source* is a member (more later)
- status – An object (struct) containing things such as source, tag, count of received elements, etc (more later)
- request - An MPI_Request object which can be used to determine when the recv() is complete via MPI_Wait*()/MPI_Test*() functions

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Avoiding Deadlock

- Blocking point-to-point calls make it possible to deadlock a program

Process 0	Process 1
MPI_Recv(from process 1)	MPI_Recv(from process 0)
MPI_Send(to process 1)	MPI_Send(to process 0)

-Use nonblocking calls

-Have odd numbered processes post Sends first

-Use MPI_Sendrecv() call

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Waiting on Requests

- All nonblocking routines (p2p and collective) return an MPI_Request object
  - To ensure completion, the calling process must call one of the MPI_Wait*() or MPI_Test*() routines on the request.

- MPI_Wait(MPI_Request *request, MPI_Status *status)
  - This routine is blocking
  - Returns when *request* is complete
  - *status* has things like tag and source for receives
  - The local operation is done. Doesn't guarantee remote is done

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Waitsome, MPI_Waitany, MPI_Waitall

- MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indeces, MPI_Status *array_of_statuses)
  - *outcount* is the number of requests completed
  - Only guaranteed that one request is completed when the call returns
- MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)
  - Returns when any one of the requests is complete. *index* is the request that was complete and *status* is an MPI_Status object for that completed request
- MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)
  - Waits until ALL *count* requests are complete

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Test*

- Int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
  - Returns *flag*=true and updates *status* if *request* is complete
    - At this point, it is as if you called MPI_Wait() on the request
  - Otherwise, *flag*=false and *status* is undefined
  - MPI_Test* return immediately
- MPI_Testany/Testall/Testsome
  - Same parameters as the Wait* equivalents, with the addition of int *flag
  - *flag* is singular, i.e. MPI_Testall *flag* is only set if ALL requests are complete

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Status

- Implementation defined “structure”, but some guaranteed fields:
  - MPI_SOURCE – The source of an incoming message (useful for using MPI_ANY_SOURCE receives)
  - MPI_TAG – The tag of an incoming message (useful with MPI_ANY_TAG)
  - MPI_ERROR – Any errors encountered in the received message
  - Indirectly contains things like the length of the message actually received
    - Requires calling MPI_Get_count(), which returns the number of entries (not bytes) received
    - Int MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count)
  - Implementations can add other fields to the structure
  - On status structures for nonblocking collectives, MPI_TAG and MPI_SOURCE are undefined
- MPI_Status can be MPI_STATUS_IGNORE

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_ANY_SOURCE - Typical use case

```
If(rank == master)
{
        while(done_count < NUM_WORKERS)
        {
                /* Wait for “I’m done” from workers; No idea who will be first */
                MPI_Recv(…. MPI_ANY_SOURCE …. &status)
                done_workers[status.MPI_SOURCE]=1;
                done_count++;
        }
        /* Everyone is done, move on */
}
else
{
        do_some_work()
        MPI_Send( “I’m done” to master)
}
```


OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI Communicators (and MPI Groups)

- An MPI group is an ordered collection of MPI processes. Groups can be manipulated separate from communicators, but only communicators can be used for direct communication
- By default, every MPI process is a member of the communicator MPI_COMM_WORLD.
- Subcommunicators can be created from MPI_COMM_WORLD
  - routines like MPI_Comm_split()
  - creating a group then using the group to create a communicator.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI Communicators and Groups

- int MPI_Comm_size(MPI_Comm comm, int *size)
  - Returns the size of the given communicator (number of ranks belonging to it).
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - Returns the rank of the calling process in the given communicator
- All processes define MPI_COMM_WORLD and a few other special communicators

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Hello World

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    int rank, size, rc;
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello from %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```


OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

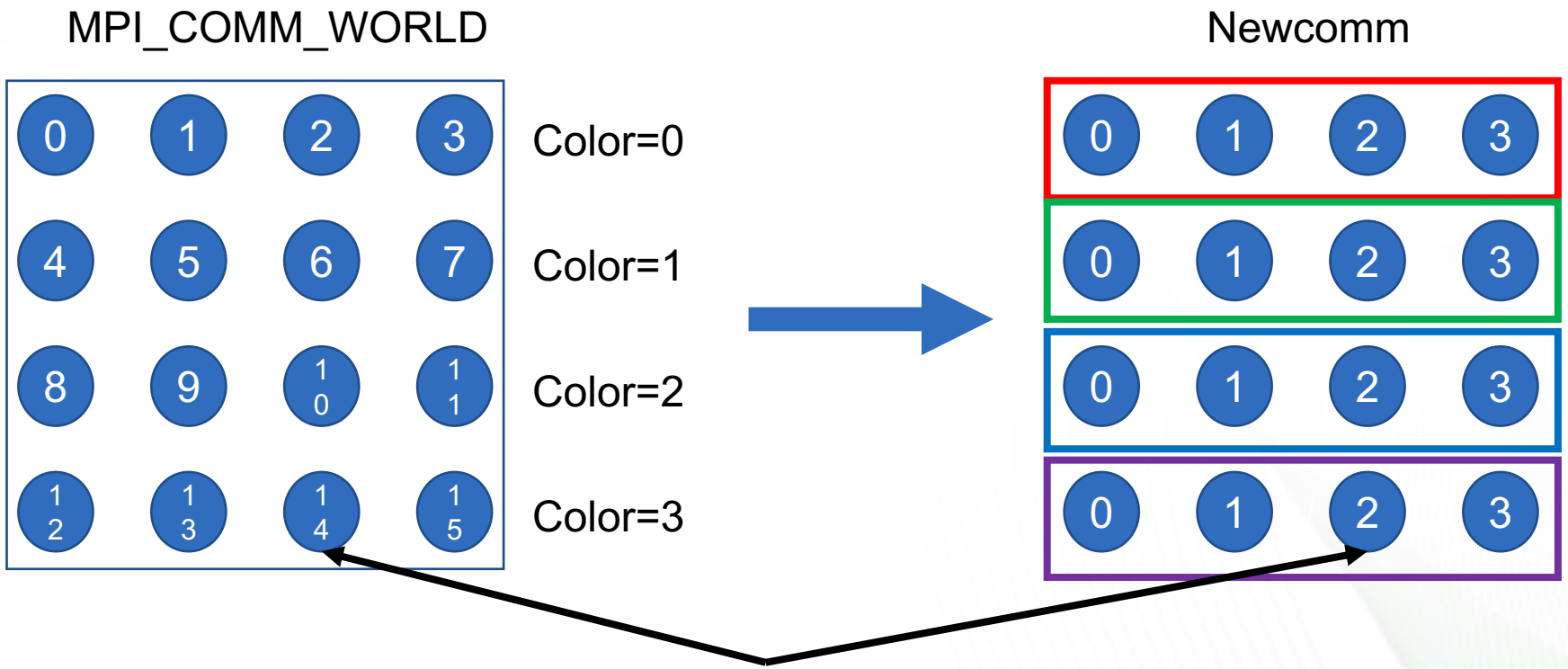
MPI_Comm_split

- Int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
- *color* is the grouping conditional, *key* controls the ranks in the new communicator
- Example

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Comm_split Example

Color=worldrank/4
Key=worldrank



MPI_Comm_rank(MPI_COMM_WORLD, &worldrank) -> 14
MPI_Comm_rank(newcomm, &newrank) -> 2
MPI_Comm_size(newcomm, &newsize) -> 4

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Collective Communications

- Nodes in a communicator all move data together
- MPI provides blocking and nonblocking versions of each collective
- New in MPI3 - neighborhood collectives
  - Enables halo-exchange with a single MPI communication call
  - Enables sparse(r) communicators and communication within them
  - Neighbors defined with MPI comm/group communicator topology creation routines

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

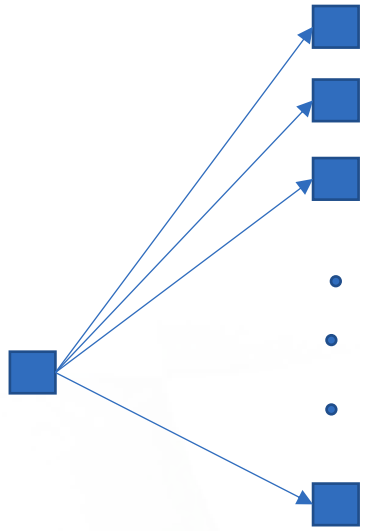
Barrier

- Simplest collective
  - int MPI_Barrier(MPI_Comm comm);
  - Provides a synchronization point where no member of *comm* can pass until all members of *comm* enter
  - Nonblocking version – call returns immediately, synchronization must occur before the MPI_Wait() call on *request* can complete.
- Example use cases
  - Synchronize after some asynchronous event (i.e. dumping data to disk)
  - Ensure all processes have a known state (i.e. all network operations are done)
  - Wait for a “time step” to complete on all nodes

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Broadcast

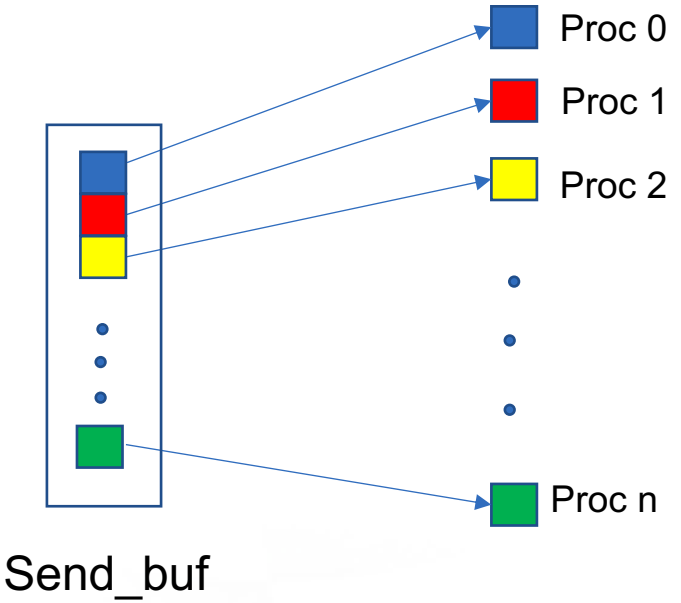
- int MPI_Bcast(void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm)
- One-to-many broadcast of a message from root to all processes in the communicator
- Example use case
  - Distribute data from an input file opened by just one node



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Scatter

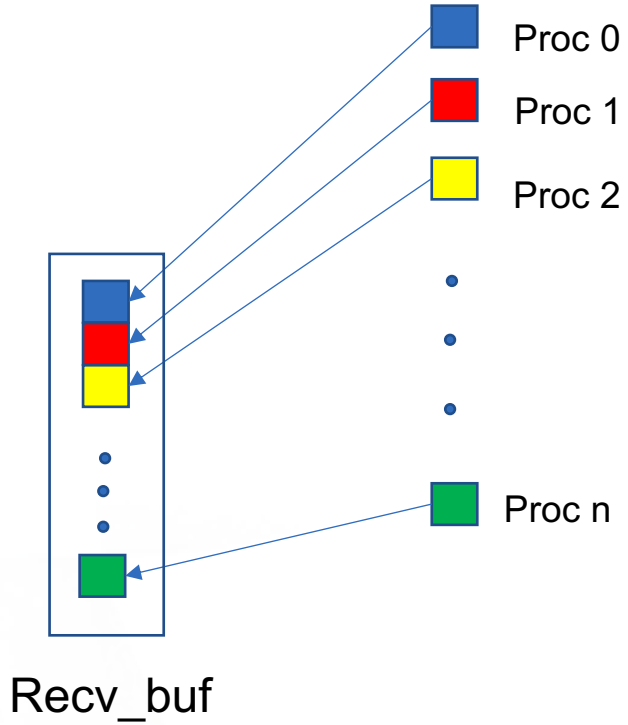
- One-to-many, different data
- int MPI_Scatter(void *sendbuf, int send_count, MPI_Datatype send_type, void *recvbuf, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm);
- Takes an array of elements from the *root* rank and in-order distributes them to the other processes in *comm* (including *root*)



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Gather

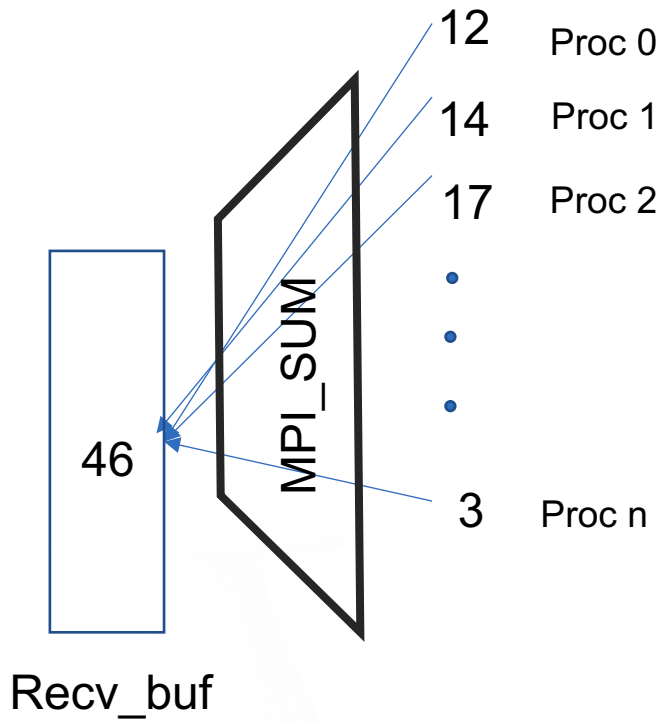
- Many-to-one
- Inverse of MPI_Scatter
- Int MPI_Gather(void *send_buf, int send_count, MPI_Datatype send_datatype, void *recv_buf, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm comm);
- Root process receives chunks of data from each process in comm, including *root* and “assembles” them back into the recv_buf



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Reduce

- Many-to-one, with an operation
- Similar to gather, but an operation is performed on the data
- int MPI_Reduce(void *send_data, void* recv_data, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm);
- Lots of pre-defined MPI_Ops – MPI_SUM, MPI_MAX, MPI_PROD, MPI_AND, MPI_MINLOC, etc
- User can define additional operations, but this is usually bad for performance (See MPI_Op_create(), MPI_Op_free())



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Reduce – Use Case

- Compute an average of some data over multiple nodes.

```
int my_value={something}, sum=0, root=0;
double average=0.0;
MPI_Reduce(&my_value, &sum, 1, MPI_INT, MPI_SUM, root,MPI_COMM_WORLD);
 if(my_rank == root)
      average = sum/(double)num_procs;
```


OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Reduce - MINLOC, MAXLOC

- Special 2-element datatypes – MPI_FLOAT_INT, MPI_2INT, MPI_LONG_INT, etc
- One {type} and one int.
- MINLOC/MAXLOC return the min/max of the {type}
- In C, use a struct for the 2 element type, then just pass MPI_MINLOC/MPI_MAXLOC as the MPI Op
- In Fortran, create an array of the {type} and promote the int to the {type}

```
DOUBLE PRECISION in(2)
    in(1) = ! Important value
    in(2) = myrank !my rank changed to a double
```


OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Example

```
struct {double val; int rank;} in, out;
in.val = /* some value */
in.rank = myrank;
MPI_Reduce(&in, &out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
If(myrank == root)
        printf("The largest value was on node %d - %lf\n", out.rank, out.val);
```

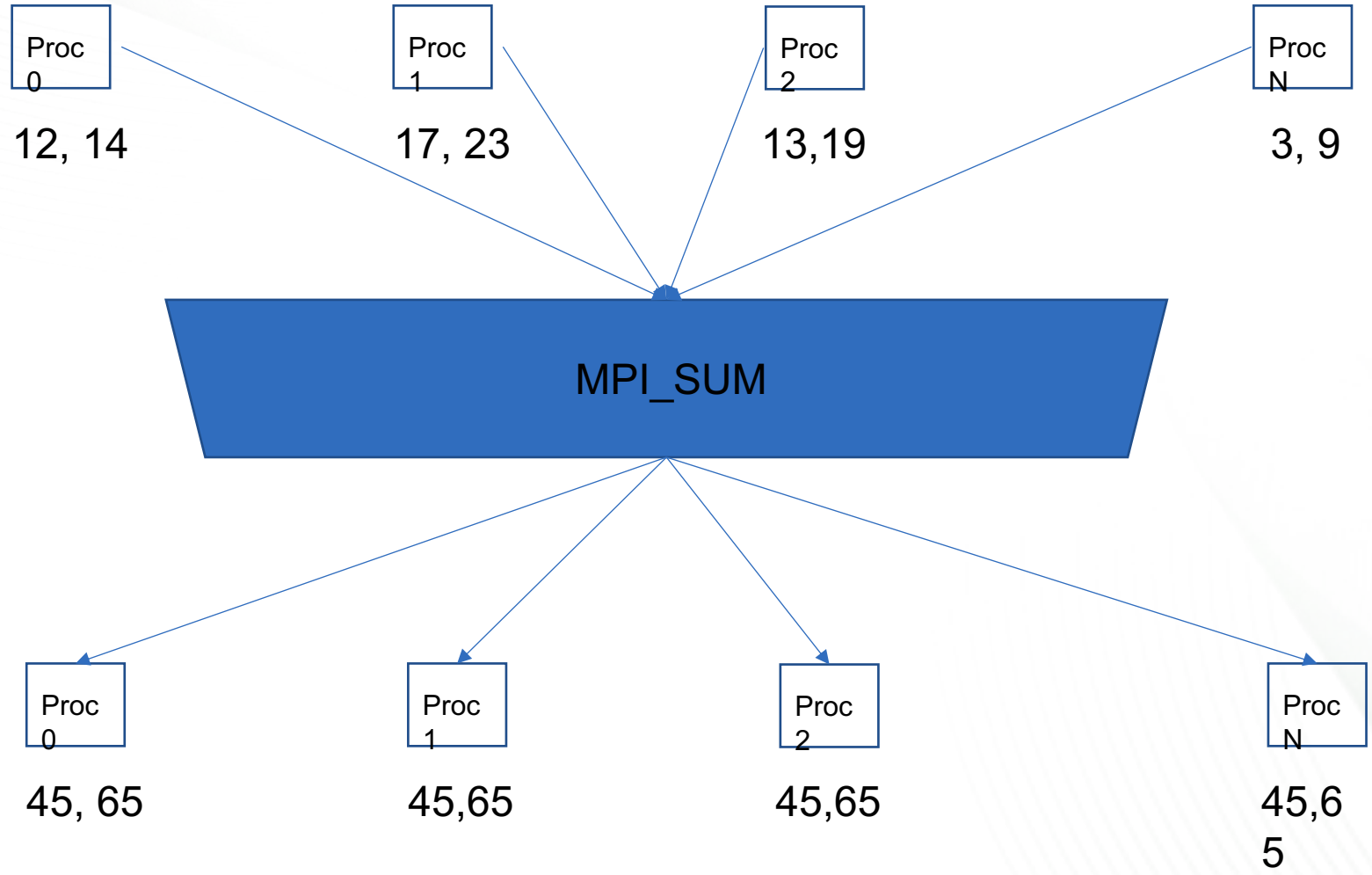

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Allreduce

- int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, MPI_Comm comm)
  - Conceptually –Reduce Operation, followed by broadcast
  - Very common in science codes

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Allreduce

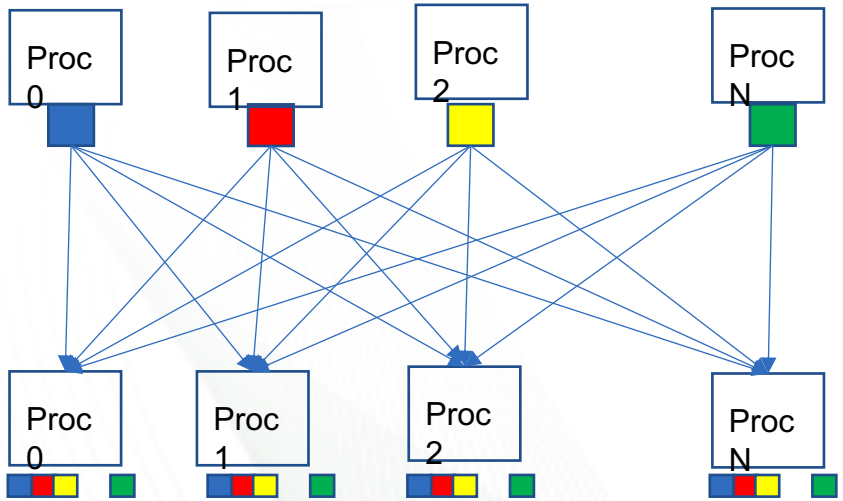


OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Allgather

int MPI_Allgather(void *sendbuf, int send_count, MPI_Datatype send_type, void *recvbuf, int recv_count, MPI_Datatype recv_type, MPI_Comm comm)

- Conceptually – MPI_Gather() followed by a broadcast
- Or, a series of N MPI_Gather()s where root=1..N



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Alltoall

- All-to-all, generalized data movement
- Essentially the same as all processes calling an MPI_Send(my_data) to all other processes and all processes calling MPI_Recv() at the same time
- Useful for shuffling data, frequently for things like FFTs and matrix transposes
- int MPI_Alltoall(void *sendbuf, int send_count, MPI_Datatype send_type,

  void *recvbuf, int recv_count, MPI_Datatype recv_type,

  MPI_Comm comm);

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Alltoall Example

- MPI_Alltoall(A, 2, MPI_INT, B, 2, MPI_INT, MPI_COMM_WORLD);

Array A	Rank	Array B
10|11|12|13|14|15|16|17	0	10|11|20|21|30|31|40|41
20|21|22|23|24|25|26|27	1	12|13|22|23|32|33|42|43
30|31|32|33|34|35|36|37	2	14|15|24|25|34|35|44|45
40|41|42|43|44|45|46|47	3	16|17|26|27|36|37|46|47

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

V (vector) variants

- Scatterv, gatherv, allgatherv, alltoallv, alltoallw
- Each process can contribute a different amount of data
- Take an array of counts and an array of displacements (distance between each element)
- These function calls can be very expensive for memory and data movement
  - Alltoallv requires 4 arrays of size(commsize) ints, plus the actual data
- Alltoallw is even more generalized and has arrays for element types
  - Requires 6 arrays of size(commsize), plus the actual data
  - Challenging to optimize at the MPI level
  - Not frequently used

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Scatterv

int MPI_Scatter(void *sendbuf, int send_count, MPI_Datatype send_type, void *recvbuf, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm);

int MPI_Scatterv(void *sendbuf, int *sendcounts, int *senddispls, MPI_Datatype send_type,

void *recvbuf, int recvcount, MPI_Datatype recv_type, int root, MPI_Comm comm)

Allows a varying count of data at varying offsets to be sent to each process from *sendbuf*. Each receiver will need to set a *recvcount* appropriately (and allocate enough memory).

Example:

int counts[4] = {2,4,6,8};

int sdispls[4] = {0, 4,16,32}

int recvcount=2*(myrank+1); /* same as sendcounts */

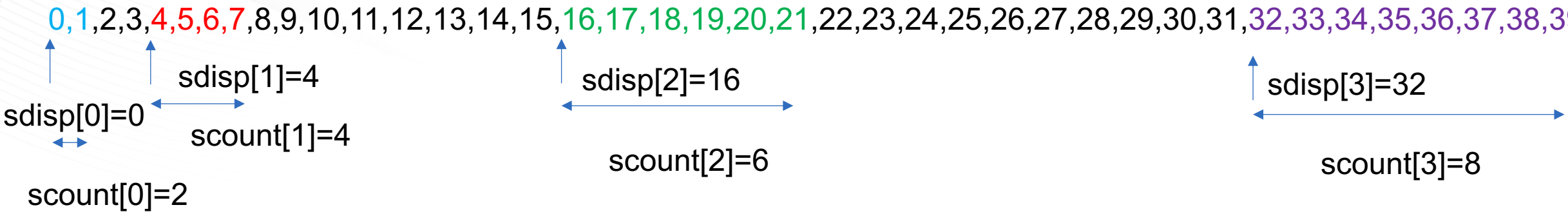
Root's sendbuf:

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Scatterv

Counts: 2,4,6,8 Disps: 0, 4, 16, 32 Recv: 2*(myrank+1)



Rank 0 (also the root) Recvbuf 0,1

Rank 1 Recvbuf 4,5,6,7

Rank 2 Recvbuf 16,17,18,19,20,21

Rank 3 Recvbuf 32,33,34,35,36,37,38,39

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Gatherv

- Int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int rdispls, MPI_Datatype recvtype, int root, MPI_Comm comm);
- Gatherv is the inverse of scatter. Data is put in the receive buffer of the root process in rank order at the places specified by the displacements

Example:

int counts[4] = {2,4,6,8};

int rdispls[4] = {0, 4,16,32}

int sendcount=2*(myrank+1);

Root's recvbuf:

0,1,x,x,4,5,6,7,x,x,x,x,x,x,x,x,16,17,18,19,20,21,x,x,x,x,x,x,x,x,x,x,32,33,34,35,36,37,38,39

X = untouched by MPI

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Allgatherv

- Int MPI_Allgatherv(void *sendbuf, int scount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
- Conceptually, a series of gatherv()s where all processes in the communicator are a root, or a gatherv() followed by a broadcast.

Example:

int counts[4] = {2,4,6,8};

int rdispls[4] = {0, 4*sizeof(int), 16*sizeof(int), 32*sizeof(int)

int sendcount=(myrank*2);

Everyone ends up with recvbuf:

0,1,x,x,4,5,6,7,x,x,x,x,x,x,x,x,16,17,18,19,20,21,x,x,x,x,x,x,x,x,x,x,32,33,34,35,36,37,38,39

X = untouched by MPI

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Alltoallv

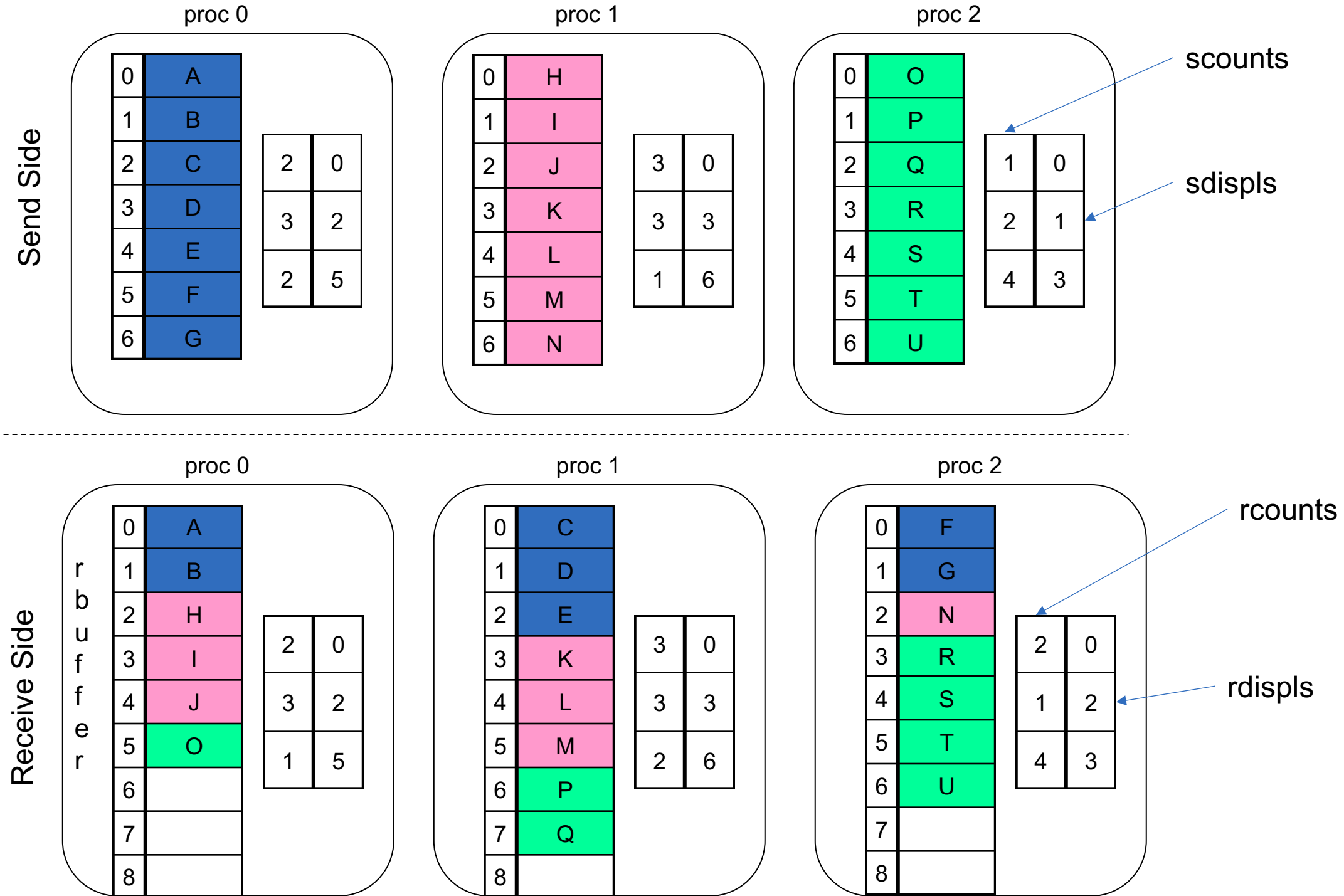
- Basically, MPI_Alltoall, but with counts and displacements for send and receive buffers
- Int MPI_Alltoallv(void *sendbuf, int *scounts, int *sdispls, MPI_Datatype stype, void *recvbuf, int *rcounts, int *rdispls, MPI_Datatype rtype, MPI_Comm comm);
- Requires a substantial amount of memory that scales as 4x communicator size

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Alltoallv Example

- Process 0:
- Scounts[]={2,3,2} Sdispls[] = {0,2,5} Rcounts[] = {2,3,1} Rdispls[] = {0,2,5}
- Process 1:
- Scounts[]={3,3,1} Sdispls[] = {0,3,6} Rcounts[] = {3,3,2} Rdispls[] = {0,3,6}
- Process 2:
- Scounts[]={1,2,4} Sdispls[] = {0,1,3} Rcounts[] = {2,1,4} Rdispls[] = {0,2,3}

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY


Send Side
proc 0
proc 1
proc 2
0 A
1 B
2 C
3 D
4 E
5 F
6 G
2 0
3 2
2 5
0 H
1 I
2 J
3 K
4 L
5 M
6 N
3 0
3 3
1 6
0 O
1 P
2 Q
3 R
4 S
5 T
6 U
1 0
2 1
4 3
scounts
sdispls
Receive Side
rbuffer
proc 0
proc 1
proc 2
0 A
1 B
2 H
3 I
4 J
5 O
6
7
8
2 0
3 2
1 5
0 C
1 D
2 E
3 K
4 L
5 M
6 P
7 Q
8
3 0
3 3
2 6
0 F
1 G
2 N
3 R
4 S
5 T
6 U
7
8
2 0
1 2
4 3
rcounts
rdispls

Alltoallw

- Even more generalized than alltoallv. Takes an array of datatypes as well as counts and displacements
- Int MPI_Alltoallw(void *sendbuf, int *scounts, int *sdispls, MPI_Datatype *stypes, void *recvbuf, int *rcounts, int *rdispls, MPI_Datatype *rtypes, MPI_Comm comm)
- Displacements are in bytes not elements
- The amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. So send type and recv type can be different as long as the counts and sizes of the types make up for it
  - E.g. send type could be double, count=1 and recv type could be float, count=2
- Can be used to generalize other MPI functions as well. For example if all but one sendcounts[i] = 0 it behaves like the equivalent of an “MPI_Scatterw()”
- 6x size of communicator memory overhead
- Challenging to optimize, rarely used

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Scan/MPI_Exscan

- MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, MPI_Comm comm)
- Same types/ops as MPI_Reduce
- Computes the prefix reduction
  - Sendbufs: 0, 1, 2, 3 on 4 nodes. Recvbufs: 0, 1, 3, 6
- Essentially, cumulative operation from ranks 0..N
- Rarely used.
- MPI_Exscan – same as MPI_Scan except calling process's data is not used

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Reduce_scatter

- Int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int *recvcounts, MPI_Datatype type, MPI_Op op, MPI_Comm comm)
- Essentially

  MPI_Reduce(sendbuf, tmpbuf, count, type, op, root, comm) followed by

  MPI_Scatterv(tmpbuf, recvcount, displs, type, recvbuff, recvcount[myrank], type, root, comm);
- (*displs*[*k*] is the sum of the *recvcounts* up to processor *k*-1)
- Primarily used for matrix-vector multiplication
- Rarely used, and usually not well optimized

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

MPI_Reduce_scatter_block (MPI3)

- int MPI_Reduce_scatter_block(void *sendbuf, void *recvbuf, int recvcount, MPI_Datatype type, MPI_Op op, MPI_Comm comm);
- Essentially an MPI_Reduce with *count=recvcount*num procs in *comm*, followed by MPI_Scatter() with the *sendcount* argument to MPI_Scatter equal to *recvcount* passed to MPI_Reduce_scatter_block.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Nonblocking Collectives

- All collectives have a nonblocking version
- Preface collective name with “I”
- Requires an extra MPI_Request *request parameter

e.g. MPI_Ibcast(void *buf, int count, MPI_Datatype type, int root, MPI_Comm comm, MPI_Request *request)

- Returns immediately.
  - Local call completion occurs at MPI_Wait*()/MPI_Test*().
  - Global completion is not guaranteed until a synchronization point (except with implicitly synchronizing collectives)

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Other Significant MPI Features

- MPI I/O
  - Collective (across a communicator) operations for file access and creation
  - Routines to access files with aggregation
  - Single file, parallel access (vs one file/process)
  - Nonblocking as well as two-stage operations
  - Noncontiguous IO with file “views”

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Other Significant MPI Features

- MPI Tools Interface
  - Most implementations provide profiling interfaces
    - Wrapper intercepts calls to MPI_Foo, does some work such as timing, internally calls MPI_Pfoo
  - Interfaces for debuggers
  - Interfaces for internal (implementation) profiling
  - Tools interface document is separate from the standard but available at the forum website and approved by the forum

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Other Significant MPI Features

- Derived Datatypes
  - MPI provides routines to construct datatypes
  - Simple vectors
  - Contiguous vectors
  - Multi-dimensional vectors-of-vectors
  - Generic C-like “structures”
  - Typically not optimized by implementors, especially for things like (All)reduce operations. i.e. performance may be bad

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Other Significant MPI Features

- One-sided/RMA communications
  - Major revamp in MPI3.x
  - Big improvement from MPI2.x ”one-sided”
  - “Put” – Copy data from a source to a target, without the target having to post a receive and (hopefully) without the target CPU being involved
  - “Get” – Pull data from a target without the target posting a send, and (hopefully) without the target CPU being involved
  - “Accumulate” – atomic operations such as fetch-and-add, compare-and-swap, etc. Useful for lock-free algorithms
  - New synchronization methods available as well.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Summary

- MPI provides a substantial number of functions and features
  - Current standard is well over 800 pages. 400+ functions
- You will probably need 10-20 calls to be productive
  - MPI_(I)Send, MPI_(I)Recv, MPI_Wait*, Allreduce, barrier, bcast, alltoall(v) and MPI_Comm_split are probably >85% of the MPI in common usage.
  - This tends to be where implementors focus major optimization efforts too
- Focus on how to actually divide up the work and decide what operations will be required to move data around

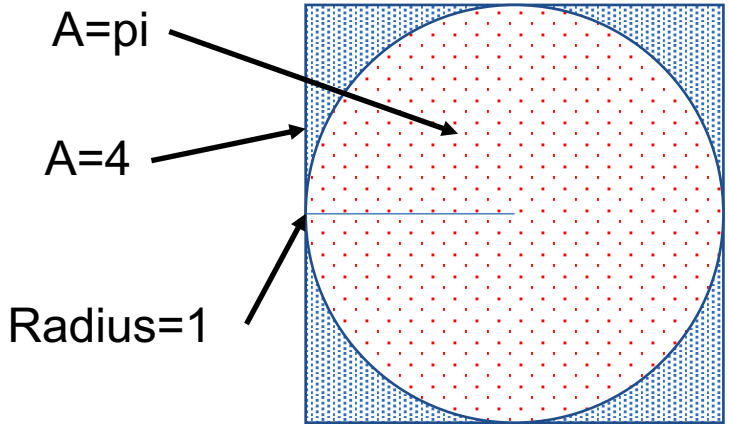
OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

- Questions before Hands On portion

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Monte Carlo Pi Calculation

- Inscribe a circle of radius r=1 unit in a square.
- The square will be 2 units wide/tall. The square will have area 2*2=4.
- The circle will have area pi*r^2=pi
- The ratio of the area of the circle to the area of the square is pi/4
- Pick random points inside the square. Some points will be inside the circle and some will not. The ratio of (inside circle) to (total) will be ~pi/4



Length/width = 2

Total points – n
Points inside circle = c
Points outside circle = s
Percentage of points inside circle: c/(c+s) or c

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Serial Version

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(int argc, char* argv[])
{
    double x,y,z, pi;
    int i, count=0, niter=1000000;
    srand(time(NULL));
    //main loop
    for (i=0; i<niter; i++)
    {
        //get random points
        x = (double)random()/RAND_MAX;
        y = (double)random()/RAND_MAX;
        z = sqrt((x*x)+(y*y));
        //check to see if point is in unit circle
        if (z<=1)
            count++;
    }
    pi = ((double)count/(double)niter)*4.0; //p = 4(m/n)
    printf("Pi: %f\n", pi);
}
```


OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

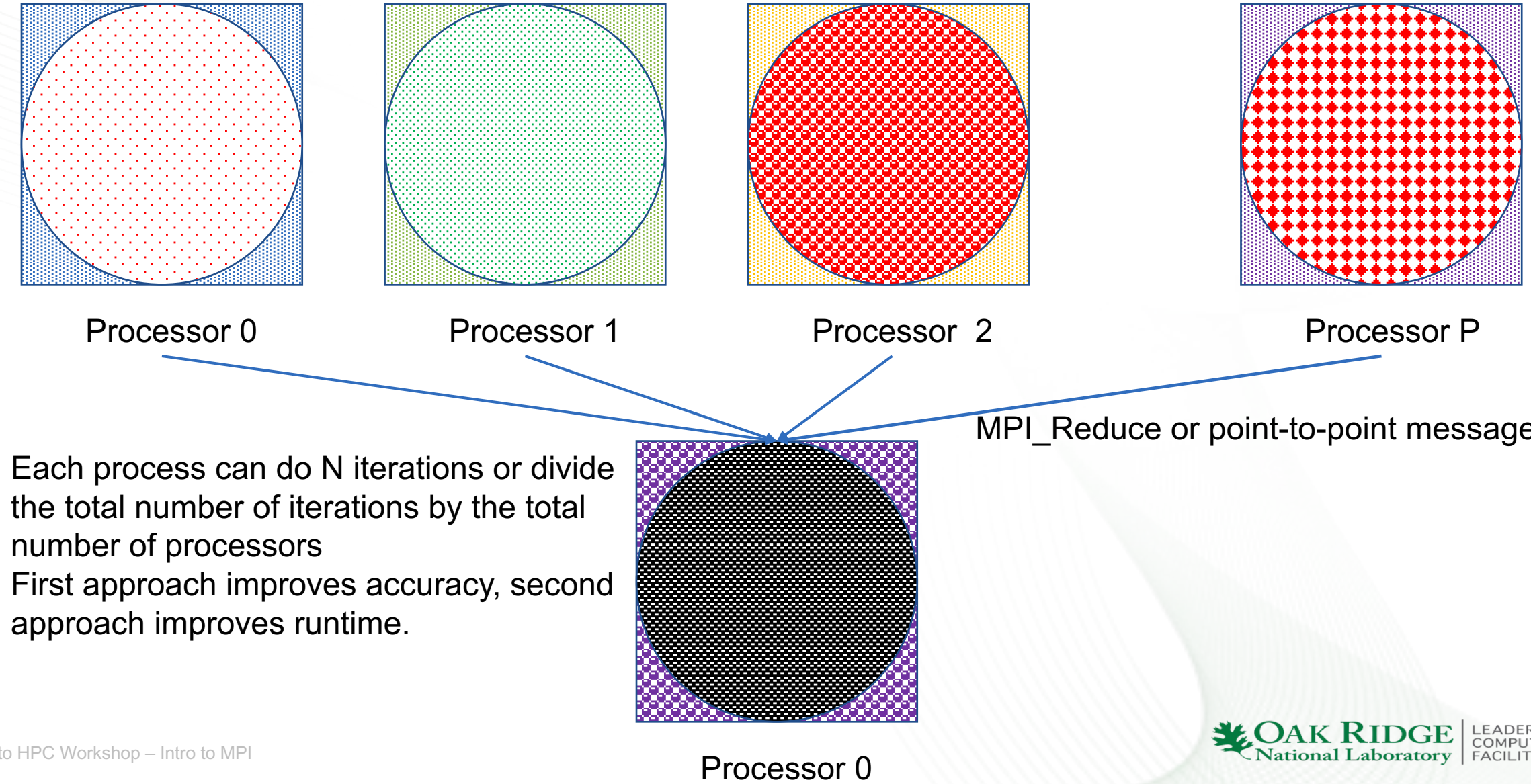
Parallel Ideas

This is a trivially parallelizable problem. There are multiple ways you could divide the problem up or parallelize the algorithm.

1. Each process computes N iterations, sends the *c* and *s* counts back to master ("weak scaling")
2. Each process computes N/*np* iterations, sends the *c* and *s* counts back to master ("strong scaling")
3. Divide the problem geometry into *np* chunks. Compute N/*np* iterations for each chunk of geometry, send *c* and *s* values back. Master determines totals
4. Hybrid – Divide the problem geometry into *np* chunks. Have multiple nodes compute N iterations per chunk. Reduce results per geometry chunk, then reduce globally.
5. Others?

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

np Processors Contribute Results back to Main Processor

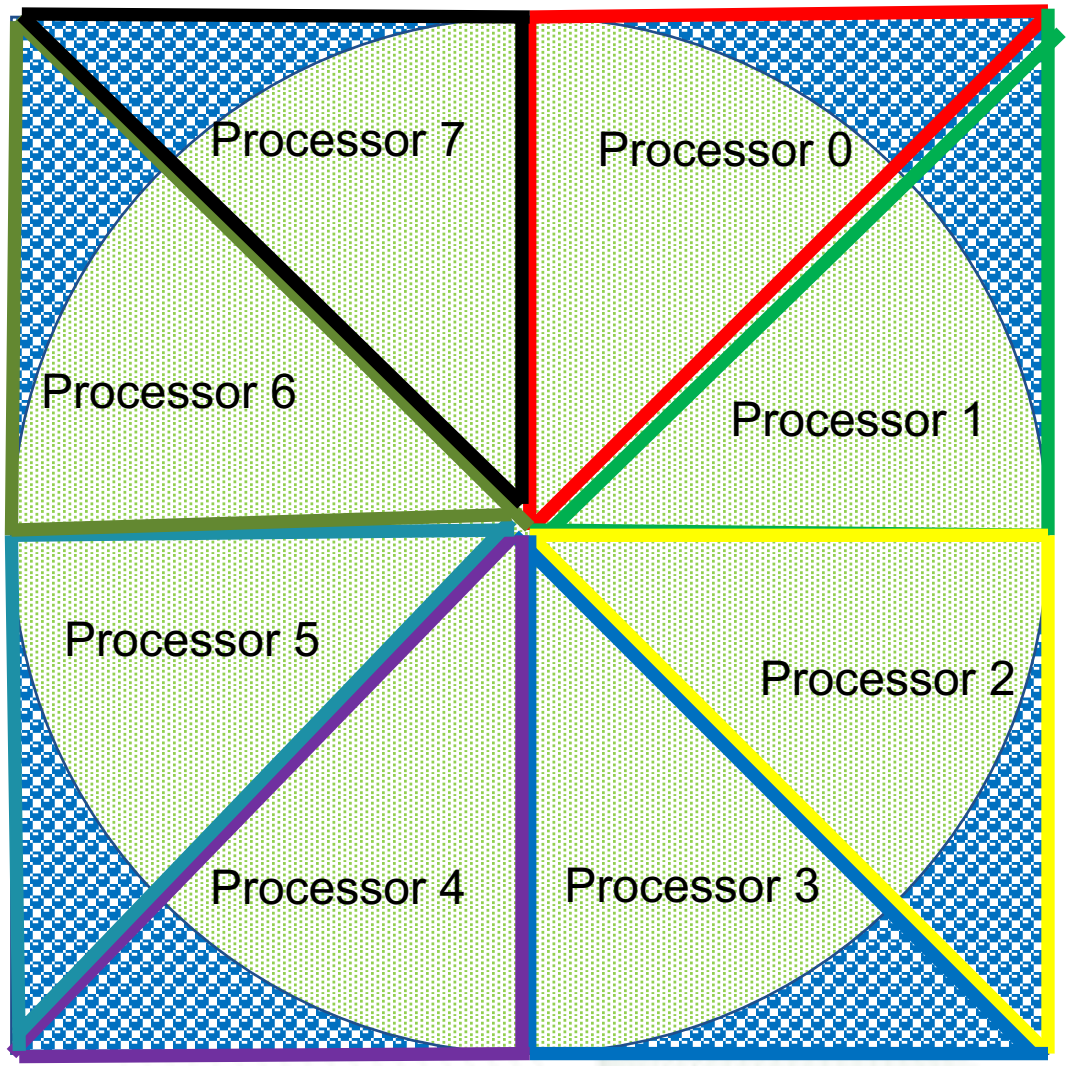


Each process can do N iterations or divide the total number of iterations by the total number of processors
First approach improves accuracy, second approach improves runtime.

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

np Processors divide the geometry

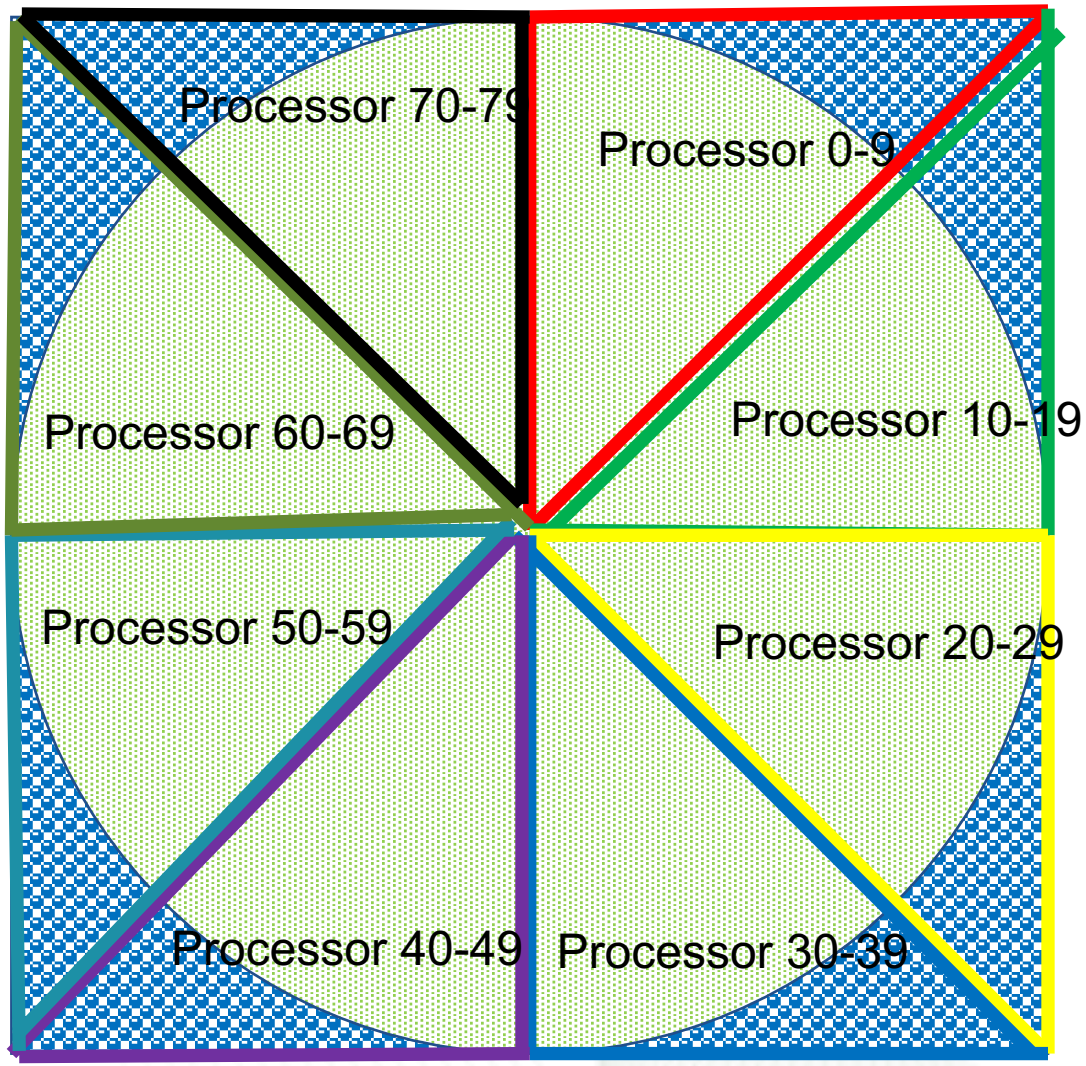
Can be more complicated algorithm, but some problems will divide this way more naturally



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Hybrid

Divide the geometry and have multiple nodes work on iterations
-Could utilize OpenMP on node for parallelizing among cores



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Monte Carlo Pi Code

- git clone https://github.com/olcf/Serial-to-Parallel--Monte-Carlo-Pi.git
- 5 examples, 2 exercises for MPI, additional OpenMP and OpenMP+MPI examples
- Makefile included
- “make examples” will build the 5 examples
- “make exercises” will make the 2 exercises, once you’ve added the bits of code (They do not compile as-is)
- “cc mpireduce-noverp.c –o mpireduce-noverp.out” for individual exercise

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Monte Carlo Pi Code

- 5 Examples
  - Serial code (serialpi.c)
  - Parallel code with blocking send-receive to a master process (mpiSRpi.c)
  - Parallel code with blocking MPI_Reduce (mpireducepi.c)
  - Parallel code with nonblocking send-receive to a master process (mpiSRnbpi.c)
  - Parallel code with nonblocking MPI_Reduce (mpiNBreducepi.c)
- 2 Exercises
  - Convert one of the MPI_Reduce examples to divide the total iterations by number of processors. Add a call to MPI_Reduce to determine how many iterations each node did. There's a stub for the blocking version. Start with mpireduce-noverp.c
  - Convert one of the send-receive examples to divide the total iterations by number of processors. Add another set of send-receives to determine how many iterations each node did. There's a stub for the blocking version. Start with mpiSRpi-noverp.c

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Using Titan

- Jobs can only run from /lustre filesystem
- Everyone should have individual $MEMBERWORK/trn001 space
- You can clone from there and build the examples and exercises there
- git clone https://github.com/olcf/Serial-to-Parallel--Monte-Carlo-Pi.git
- ”make examples”, then edit mpireduce-noverp.c or mpiSRpi-noverp.c and compile them
- qsub –I –A trn001 –lnodes=2, walltime=60:00
  - -I – interactive. (capital “Eye”) Recommended for debugging
  - -A – the project name
  - -l – what resources your job needs. In this case, 2 nodes and one hour of time (lowercase “ell”)
- Once the allocation starts you can change to $MEMBERWORK/trn001
  - aprun –n 1 ./serialpi.out
  - aprun –n 2 ./mpiSR-pi.out

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY