HELP US ALL STAY HEALTHY

SIX SIMPLE TIPS

Maintain 1.5 metres distance between yourself and others where possible

Cough and sneeze into your elbow or a tissue (not your hands)

Avoid shaking hands

Put used tissues in the bin

Wash hands with soap and warm water or use an alcohol-based hand sanitiser after you cough or sneeze

Do not touch your face

IF YOU ARE UNWELL AND WORRIED ABOUT COVID-19:

- Call the National Coronavirus Helpline: 1800 020 080
- Call your usual GP for advice
- Call the UWA Medical Centre for advice: 6488 2118

UWA FAQs:
uwa.edu.au/coronavirus

Report COVID-19 hazards and suspected/confirmed cases via RiskWare:
uwa.edu.au/riskware

THE UNIVERSITY OF WESTERN AUSTRALIA

DCS:507757521

High-Performance Computing

Lecture 7 MPI Collective Communication

CITS5507

Zeyi Wen

Computer Science and Software Engineering

School of Maths, Physics and Computing

Acknowledgement: The lecture slides are adapted from many online sources.

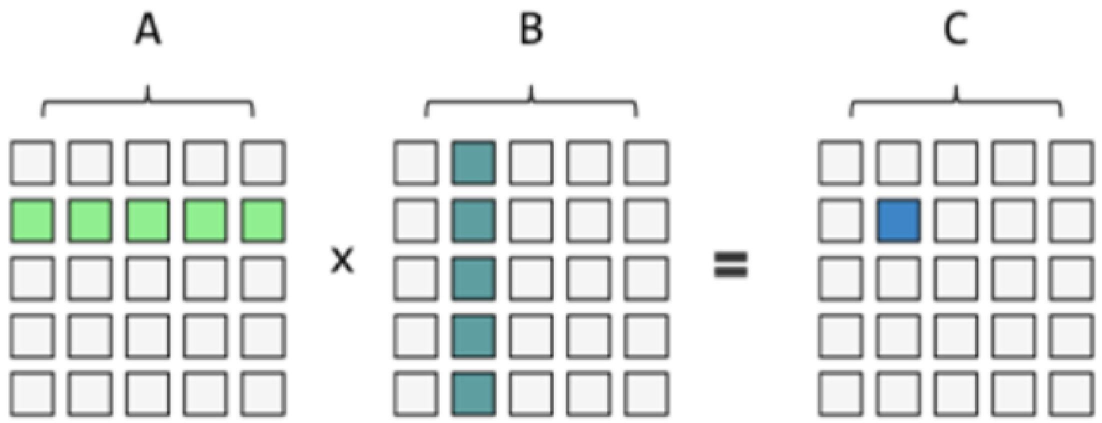
Outline

- **Point-to-Point (continued)**
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

Extended Example - Matrix Multiplication

- We introduce one of the most common structures for a parallel program
  - Self-scheduling
  - Master-worker
  - In the code, the master process distributes a matrix multiply operation to (numtasks-1) worker processes

Matrix Multiplication - Definition



```
for(int i=0;i<ROW;i++)
{
	for(int j=0;j<ROW;j++)
	{
		for(int z=0;z<COL;z++)
		{
			C[i][j] += A[i][z]*B[z][j];
		}
	}
}
```

Matrix Multiplication - Initialise

THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define NRA 62 /* number of rows in matrix A */
#define NCA 15 /* number of columns in matrix A */
#define NCB 7 /* number of columns in matrix B */
#define MASTER 0 /* taskid of first task */
#define FROM_MASTER 1 /* setting a message type */
#define FROM_WORKER 2 /* setting a message type */
int main (int argc, char *argv[]) {
 int numtasks, /* number of tasks in partition */
 taskid, /* a task identifier */ numworkers, /* number of worker tasks */
 source, /* task id of message source */ dest, /* task id of message destination */
 mtype, /* message type */ rows, /* rows of matrix A sent to each worker */
 averow, extra, offset, /* used to determine rows sent to each worker */
 i, j, k, rc; /* misc */
 double a[NRA][NCA], /* matrix A */ b[NCA][NCB], /* matrix B */
 c[NRA][NCB]; /* result matrix C */ MPI_Status status;
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
 MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
 if (numtasks < 2 ) {
   printf("Need at least two MPI tasks. Quitting...\n");
   MPI_Abort(MPI_COMM_WORLD, rc); exit(1);
 }
 numworkers = numtasks-1;
```

Terminates all MPI processes associated with the communicator.

Matrix Multiplication- Master Task

```
if (taskid == MASTER) {
  printf("mpi_mm has started with %d tasks.\n",numtasks);
  printf("Initializing arrays...\n");
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
        a[i][j]= i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
        b[i][j]= i*j;
/* Send matrix data to the worker tasks */
  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  mtype = FROM_MASTER;
  for (dest=1; dest<=numworkers; dest++) {
     rows = (dest <= extra) ? averow+1 : averow;
     printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
     MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
     MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
     MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
     MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
     offset = offset + rows;
  }
```

Matrix Multiplication- Master Task

```
/* Receive results from worker tasks */
mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++) {
  source = i;
  MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,  &status);
  MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
  printf("Received results from task %d\n",source); } /* Print results */
  printf("******************************************************\n");
  printf("Result Matrix:\n");
  for (i=0; i<NRA; i++) {
    printf("\n");
    for (j=0; j<NCB; j++)
      printf("%6.2f ", c[i][j]);
  }
  printf("\n******************************************************\n");
  printf ("Done.\n");
}
```

Matrix Multiplication- Worker Task

```
if (taskid > MASTER) {
  mtype = FROM_MASTER;
  MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
  for (k=0; k<NCB; k++)
    for (i=0; i<rows; i++) {
      c[i][k] = 0.0;
      for (j=0; j<NCA; j++)
         c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }
  mtype = FROM_WORKER;
  MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
}

MPI_Finalize();
}
```

Matrix Multiplication - Summary

Summary

- Each worker process is assigned to the partial rows of matrix A and the whole matrix B by master.
- Each worker process calculates the product of the partial rows of matrix A and matrix B to get the partial rows of matrix C.
- After all processes finish the calculation, the result is passed to the master process for summary and the final result is obtained.

Outline

- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

Parallel Program Performance

- Timing program is one way to test parallel performance, but we can do more than that

- In this case, goal is to estimate theoretically
  - ✓ **Computation**
  - ✓ **Communication**
  - ✓ **Scaling w.r.t. (with regard to) problem size**

Performance Analysis: Example 1

- Consider **Matrix-vector multiplication**
  - ✓ Square, dense matrix $n \times n$
  - ✓ Each element of c requires n multiplications and $n - 1$ additions
  - ✓ There are n elements in c so our FLOP requirements are

$$n(n + (n - 1)) = 2n^2 - n$$

Performance Analysis: Example 1 (Cont'd)

- We also consider communication **costs**
- We assume all processes have the original vector already
- Need to send n + 1 values (sending to and back)
- n times (for each row)

$$n\,(n + 1) = n^2 + n$$

- A ratio of communication to computation is

$$(n^2 + n) / (2n^2 - n) \times (T_{comm} / T_{calc})$$

- Computation is usually cheaper than communication
  - ✓ Since we try to **minimise** this ratio
- Often making the problem larger makes communication overhead insignificant
- Here, this is not the case
  - ✓ For large n the ratio gets closer to 1

Performance Analysis: Example 2

- We could easily adapt our approach for **matrix-matrix multiplication**
- Instead of a vector b we have another square **matrix B**
- Each round sees a **vector** sent back instead of a single value

Performance Analysis: Example 2 (Cont'd)

- Computation requirements
  - ✓ The operations for each element of C is n multiplications and n-1 adds
  - ✓ Now n^2 elements to compute

$$n^2\,(n + n - 1) = 2n^3 - n^3$$

- Communication requirements
  - ✓ n (to send each row) + n (to send a row back) and there are n rows in total, so

$$n \times 2n = 2n^2$$

- Communication/Calculation ratio

$$(2n^2 \;/\; (2n^3 - n^2)) \times (T_{comm} \;/\; T_{calc})$$

  - ✓ Which scales to 1 / n for large n

Outline

- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- **Collective Communication**
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

Collective Communication Introduction

- Collective communications transmit data among all processes in a communicator.
  - ✓ **Barriers synchronise** processes without passing extra data.
  - ✓ **Global communication functions** with a variety of patterns.
  - ✓ **Global reduction** (max, min, sum etc.) across all processes.

The communication function and communicator itself work together to achieve tremendous performance

- ✓ Collective communication functions can leverage special optimisations over many point-to-point calls.

Some semantics

- Some collective communication involves a single process sending information to all others
  - ✓ This process is the **root** (*typically, rank == 0*)

- All collective communication functions come in two flavours
  - ✓ **Simple** → Data is stored contiguously
  - ✓ **Vectored** → Can 'pick and choose' from an array

Types of collective communication

Collective communication operations are made of the following types:

Barrier Synchronisation – Blocks until all processes have reached a synchronisation point

Data Movement (or Global Communication) – Broadcast, Scatters, Gather, All to All transmission of data across the communicator.

Collective Operations (or Global Reduction) – One process from the communicator collects data from each process and performs an operation on that data to compute a result.

Outline

- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

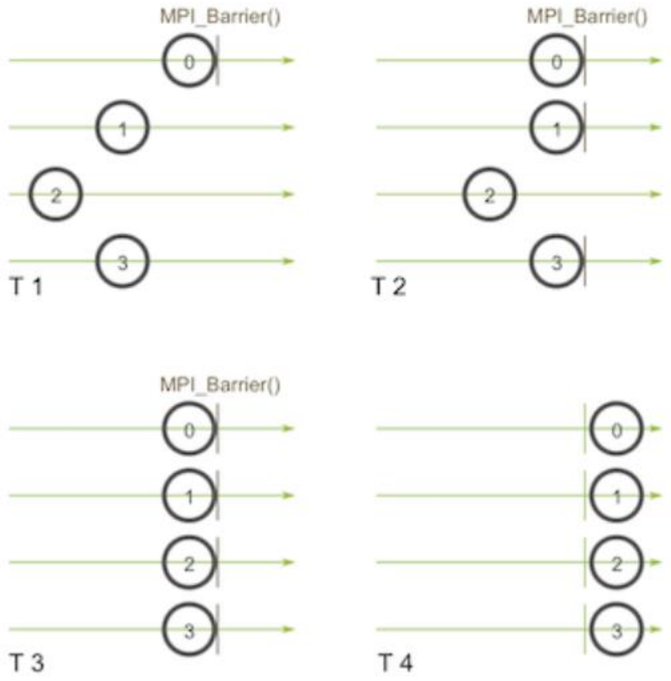
Barrier Synchronisation

MPI_Barrier blocks until all process have reached this routine,

```
MPI_Barrier( MPI_Comm comm );
```

comm
communicator (handle)

That is, the call returns at any process only after all members of the communicator have entered the call.



Barrier Synchronisation - Examples

```
#include "stdio.h"
#include "string.h"
#include "mpi.h"

int main(int agc,char *agv[])
{
     int comm_size;
     int my_rank;

     MPI_Init(&agc,&agv);
     MPI_Comm_size(MPI_COMM_WORLD,&comm_size);
     MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
     for(int i=0;i<5;i++)
          printf("process %d: %d\n",my_rank,i);
     printf("waiting.....\n");
     MPI_Barrier(MPI_COMM_WORLD);
     for(int i=5;i<10;i++)
          printf("process %d: %d\n",my_rank,i);
     MPI_Finalize();

     return 0;
}
```

When the process has finished printing 0-4, it waits for other processes to finish printing 0-4 before continuing to print 5-9.

Outline

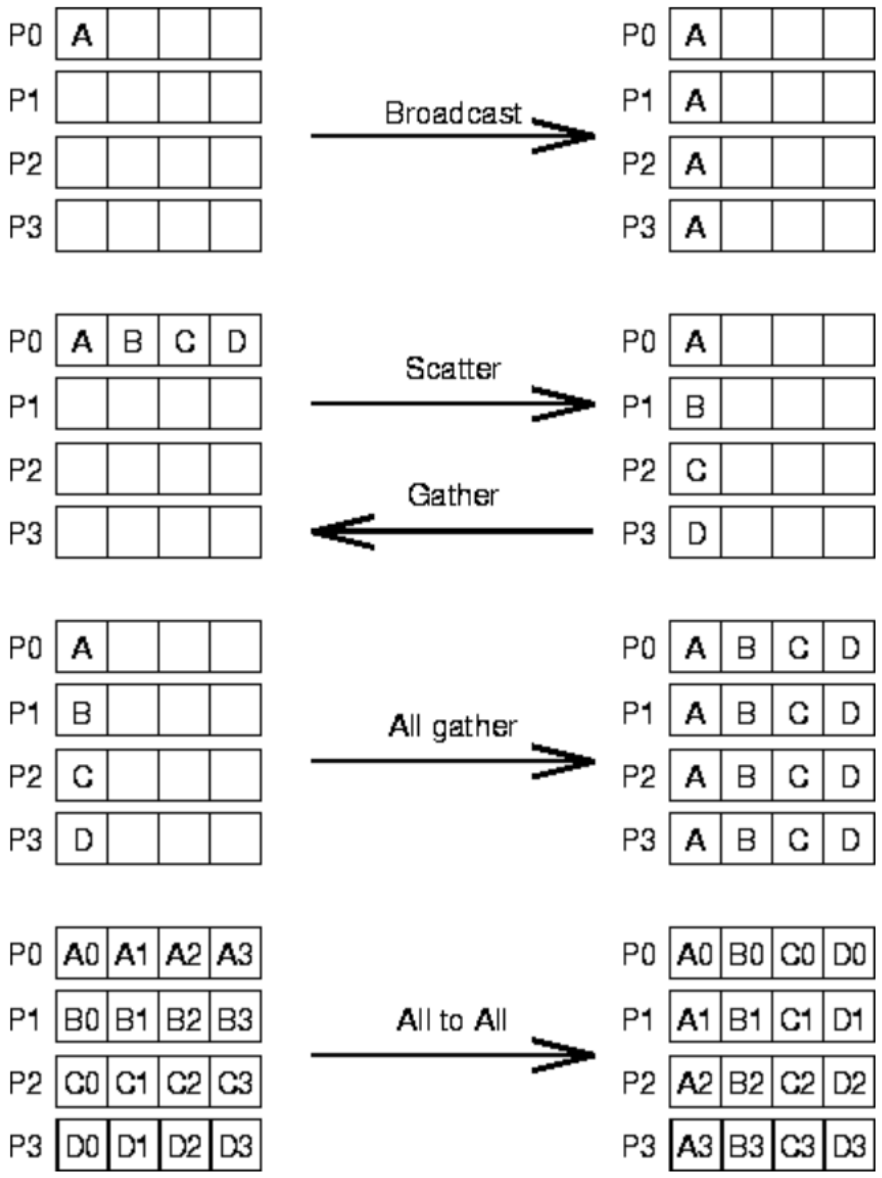
- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

Global Communication - Basic Patterns

Three flavours:

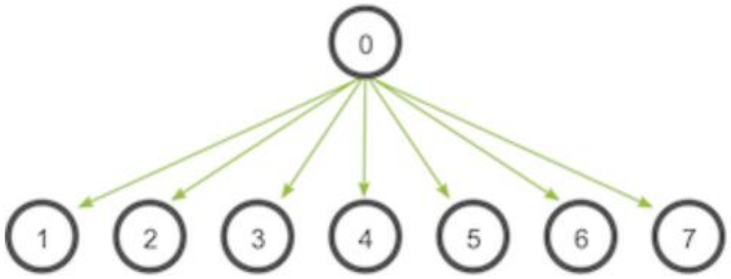
- Root sends to all processes (itself included)
  - ✓ Broadcast, Scatter
- Root receives data from all processes (itself included)
  - ✓ Gather
- Each process communications with each process (itself included)
  - ✓ Allgather and Alltoall

Global Communication – Overview



Global Communication- Broadcast

- A *broadcast* is one of the standard collective communication techniques.
- During a broadcast, one process sends the same data to all processes in a communicator.
- One of the main uses of broadcasting is to send out user input to a parallel program or send out configuration parameters to all processes.
- The communication pattern of a broadcast looks like this:



Process zero is the *root* process, and it has the initial copy of data. All of the other processes receive the copy of data.

Global Communication- Broadcast API

```
MPI_BCAST(buffer, count, datatype, root, comm)
```

• buffer	(INOUT)	starting address of buffer
• count	(IN)	number of elements in buffer
• datatype	(IN)	datatype of the buffer
• root	(IN)	the rank of the root in the comm
• comm	(IN)	the communicator

Global Communication- Broadcast

```
#include "mpi.h"

int main( int argc, char* argv[] )
{
    int rank;
    int ibuf;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if(rank == 0)
        ibuf = 12345;
    else // set ibuf Zero for non-root processes
        ibuf = 0;

    MPI_Bcast(&ibuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank !=0 )
        printf("my rank = %d ibuf = %d\n", rank,ibuf);

    MPI_Finalize();

}
```

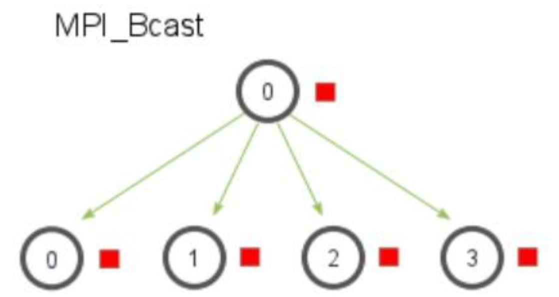
The root process broadcasts 12345 to other processes

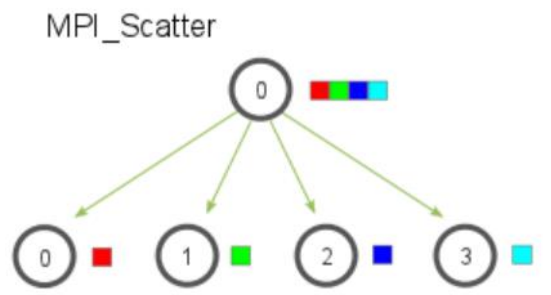
Global Communication- Scatter and Gather

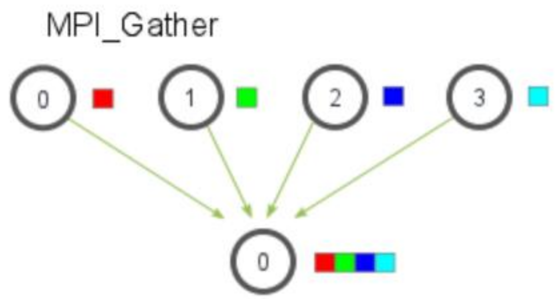
- **MPI_Bcast** takes a single data element at the root process (the red box) and copies it to all other processes.

- **MPI_Scatter** takes an array of elements and distributes the elements in the order of process rank.

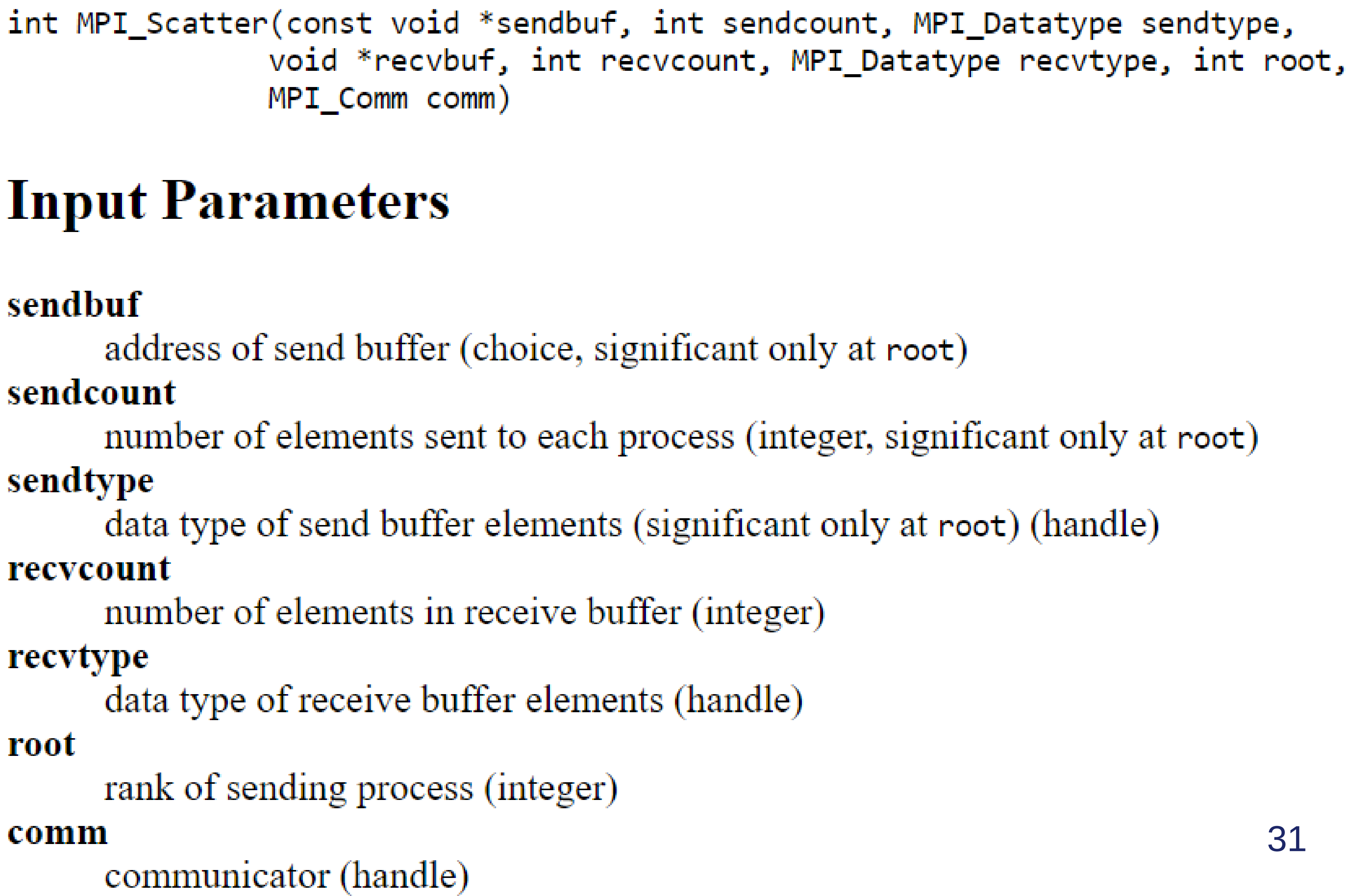
- **MPI_Gather** is the inverse of MPI_Scatter







Global Communication- Scatter and Gather



```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm)
```

Input Parameters

sendbuf
: address of send buffer (choice, significant only at `root`)

sendcount
: number of elements sent to each process (integer, significant only at `root`)

sendtype
: data type of send buffer elements (significant only at `root`) (handle)

recvcount
: number of elements in receive buffer (integer)

recvtype
: data type of receive buffer elements (handle)

root
: rank of sending process (integer)

comm
: communicator (handle)

Global Communication- Scatter and Gather

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Input Parameters

sendbuf
: starting address of send buffer (choice)

sendcount
: number of elements in send buffer (integer)

sendtype
: data type of send buffer elements (handle)

recvcount
: number of elements for any single receive (integer, significant only at root)

recvtype
: data type of recv buffer elements (significant only at root) (handle)

root
: rank of receiving process (integer)

comm
: communicator (handle)

Global Communication- Scatter and Gather

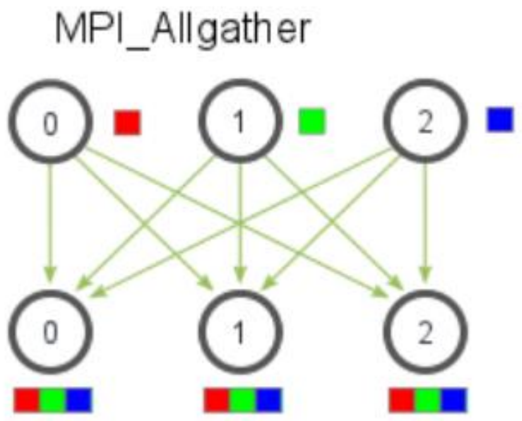
THE UNIVERSITY OF WESTERN AUSTRALIA

Computing the average of an array (original source code)

```
float *rand_nums = NULL;
if (world_rank == 0)
         rand_nums = create_rand_nums(elements_per_proc * world_size);

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
         elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;

if (world_rank == 0) sub_avgs = malloc(sizeof(float) * world_size);
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) float avg = compute_avg(sub_avgs, world_size);
```

Global Communication- Allgather

- Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes.



Global Communication- Allgather

Average computation by using MPI_Allgather (original code)

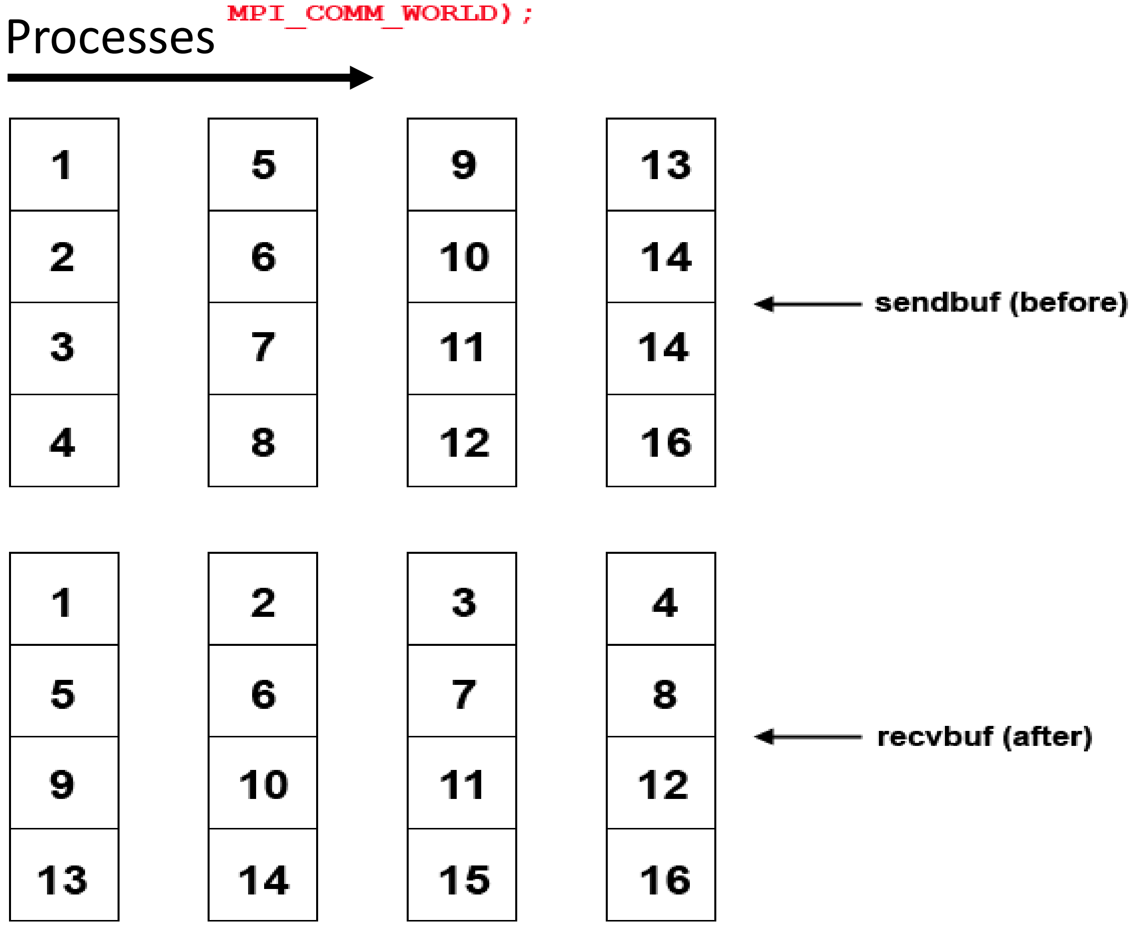
```
// Gather all partial averages down to all the processes
float *sub_avgs = (float *)malloc(sizeof(float) * world_size);
MPI_Allgather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, MPI_COMM_WORLD);

// Compute the total average of all numbers.
float avg = compute_avg(sub_avgs, world_size)
```

```
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

Global Communication- Alltoall

Scatter data from all tasks to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT
             recvbuf, recvcnt, MPI_INT
             MPI_COMM_WORLD);
```



Outline

- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

Global Reductions

- **Global reductions** perform some numerical operation in a distributed manner and is extremely useful in many cases
  - ✓ Analogous to reduction operators in OpenMP

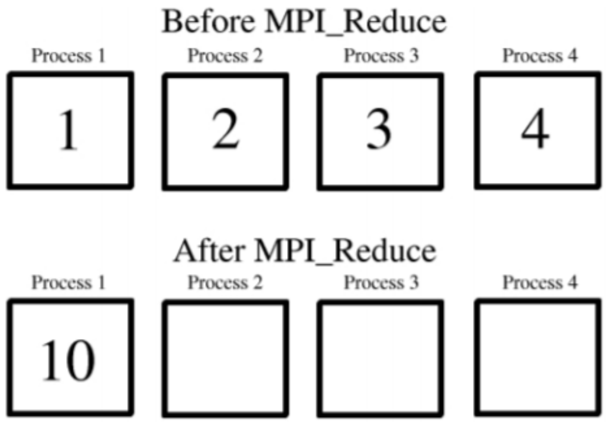
- Many numerical algorithms can replace senc/recv with broadcast/reduce with a correct topology

- Some operations which can be performed include:
  - ✓ Max
  - ✓ Min
  - ✓ Sum
  - ✓ Product, etc. (there are others)

Global Reductions - Reduce

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

sendbuf	(IN)	Address of send buffer
recvbuf	(OUT)	Address of receive buffer
count	(IN)	The number of elements in the send buffer
datatype	(IN)	The datatype of elements in the buffer
op	(IN)	*NEW* The reduce operation
root	(IN)	Rank of root process
comm	(IN)	Communicator



Global Reductions - Operations

The reduction operations defined by MPI include:

MPI_MAX	- Returns the maximum element.
MPI_MIN	- Returns the minimum element.
MPI_SUM	- Sums the elements.
MPI_PROD	- Multiplies all elements.
MPI_LAND	- Performs a logical ***and*** across the elements.
MPI_LOR	- Performs a logical ***or*** across the elements.
MPI_BAND	- Performs a bitwise ***and*** across the bits of the elements.
MPI_BOR	- Performs a bitwise ***or*** across the bits of the elements.
MPI_MAXLOC	- Returns the maximum value and the rank of the process that owns it.
MPI_MINLOC	- Returns the minimum value and the rank of the process that owns it.

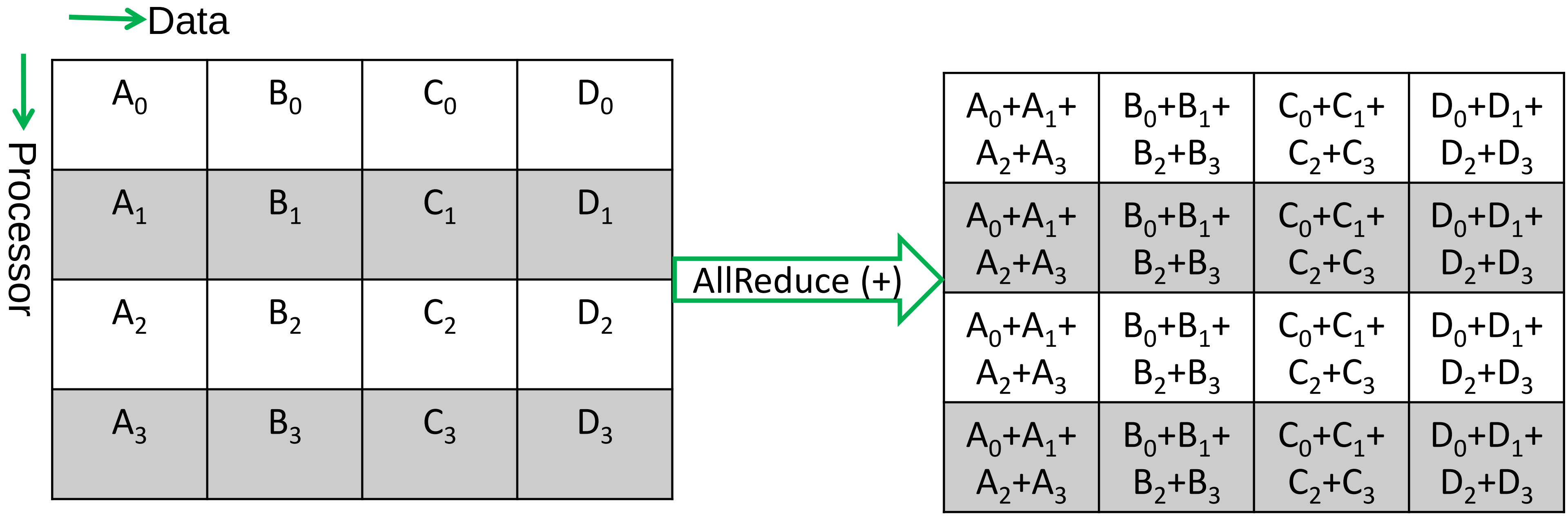
Global Reductions - Reduce

Computing average of numbers with MPI_Reduce (original source code)

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) { local_sum += rand_nums[i]; }

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n", world_rank, local_sum,
         local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,    MPI_COMM_WORLD);

// Print the result
if (world_rank == 0)
         printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
```

Global Reductions – AllReduce

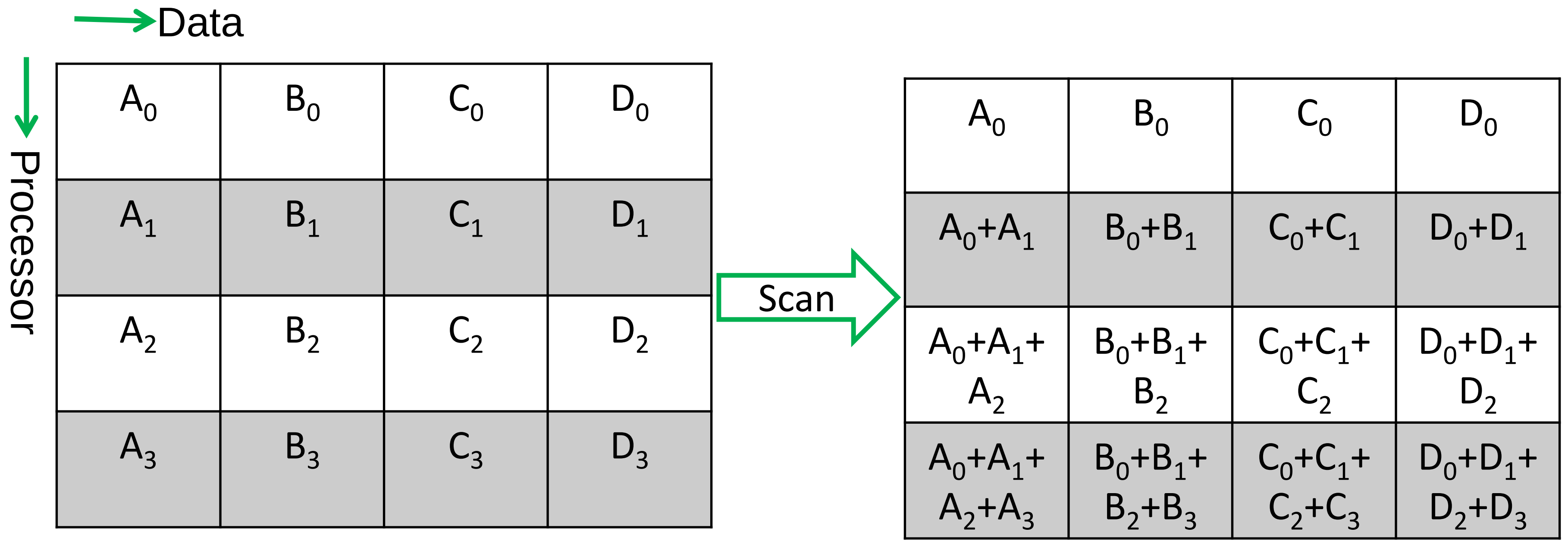
```
MPI_Allreduce( void* send_data, void* recv_data, int count,
          MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator)
```



Combines the elements in all the sendbufs of each process (using an operation) and returns that value to all processes.

Global Reductions - Scan

```
MPI_Scan( void *sendbuf, void *recvbuf, int count,
          MPI_Datatype datatype, MPI_Op op, MPI_Comm comm );
```

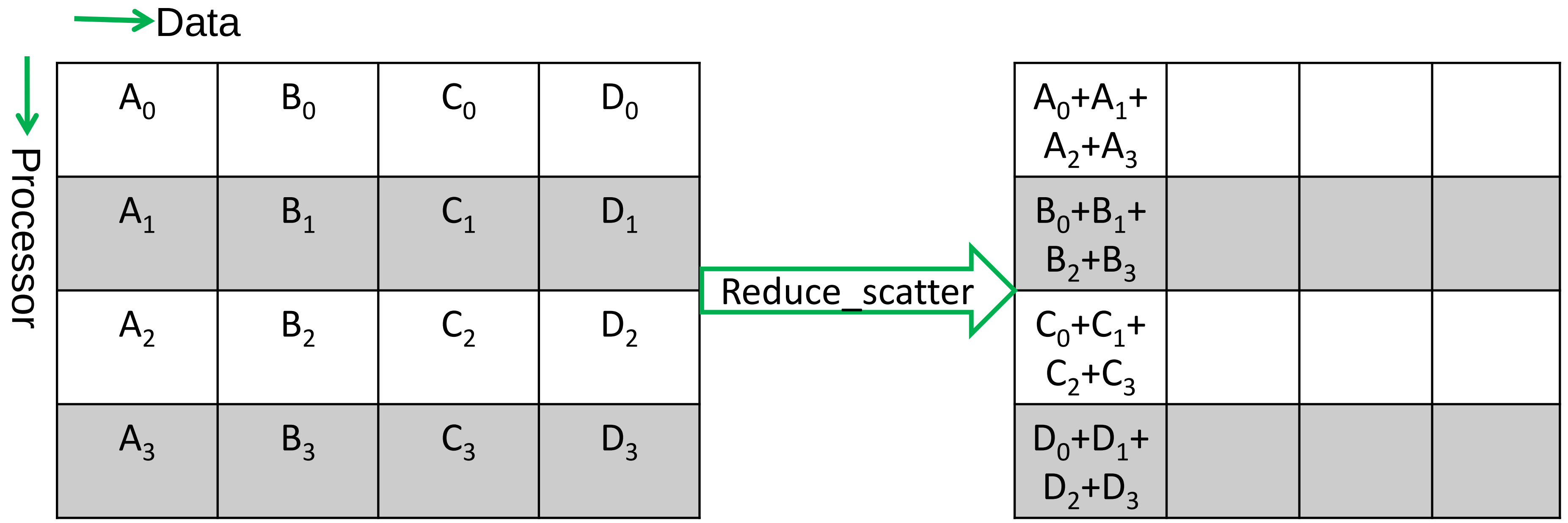


Data
Processor
A_0 B_0 C_0 D_0
A_1 B_1 C_1 D_1
A_2 B_2 C_2 D_2
A_3 B_3 C_3 D_3
Scan
A_0 B_0 C_0 D_0
A_0+A_1 B_0+B_1 C_0+C_1 D_0+D_1
$A_0+A_1+A_2$ $B_0+B_1+B_2$ $C_0+C_1+C_2$ $D_0+D_1+D_2$
$A_0+A_1+A_2+A_3$ $B_0+B_1+B_2+B_3$ $C_0+C_1+C_2+C_3$ $D_0+D_1+D_2+D_3$

Combines the elements in all the sendbufs of each process and the ‘prior’ result. i.e. Performs a prefix reduction.

Global Reductions - Reduce-Scatter

```
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int
        recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```



Combines the elements in all the sendbufs in chunks of size n of each process (using an operation) then distributes the resulting array over n processes

Global Reductions - Custom Reductions

- It is possible to define your own reduction operation, as long as it is associative
  - ✓ ‘Gives the same result regardless of the grouping of input’
  - ✓ E.g. Max, Min, Avg, etc.
  - ✓ E.g. averaging on the even numbers in an array, finding the absolute maximum, absolute average, etc.

- The operation can be commutative if specified
  - ✓ The order of operations doesn’t matter (e.g. Max, Min, Sum, etc.)

- The function must fit a specific definition and is then bound to an OP_HANDLE
- No MPI communication function can be inside your custom reduction

Global Reductions - Custom Reductions

```
typdef void MPI_User_function(void *invec, void *inoutvec, int *len,
          MPI_Datatype *datatype);
```

```
MPI_OP_CREATE(MPI_User_function *function, int commute, MPI_Op op)
```

• Function	(IN)	The user defined function
• Commute	(IN)	True if commutative, false otherwise
• Op	(OUT)	The operation

*[*More Information]*

Outline

- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

What is Communicator?

- Put simply, a communicator is a **group of processes**.

- But first, a quick reminder of why MPI exists – To make point to point and collective communication **portable** between machines.

- At the time, a few key problems existed in the field. Understanding these problems makes understanding MPI easier

Communicator: Division of Processes

- In some applications, we would like different groups of processes to do different independent tasks at a very coarse level
  - ✓ **E.g**. use 2/3 of our machine to predict weather patterns, use 1/3 to process new data

- Sometimes we divide a task based on data. It makes sense the operations acting on parts of our data is addressed to those processes
  - ✓ **E.g**. Performing operations on a diagonal of a matrix → It would be nice to reference the diagonal by name (no matter how many processes we have)

Communicator: Avoiding Message Conflicts

- Library routines have had difficulty in isolating their messages from other libraries
  - ✓ E.g. **MPI_ANY_TAG** being consumed by the wrong library

- MPI is designed to avoid this, communicators allow a library to **segment** traffic for itself
  - ✓ We don’t always know which modules before hand will be run, so we need to define these communicators at run time

Communicator: Extensibility to Users

- Often, computing efficient communication patterns (for an arbitrary machine) given a particular routine is **expensive**
- But can be reused
- After the pre-computation, if we build a communicator for an efficient way of communication, we only need to perform that operation **once**
- Also allows for logical naming of groups

Communicator: Safety

- By requiring routines to be managed by communicators, MPI implementers can guarantee **safe** (and hopefully efficient) execution

Groups

- A group is an **ordered set** of process identifiers (called processes)
  - ✓ Each process has an integer rank
  - ✓ Ranks are contiguous and start at 0

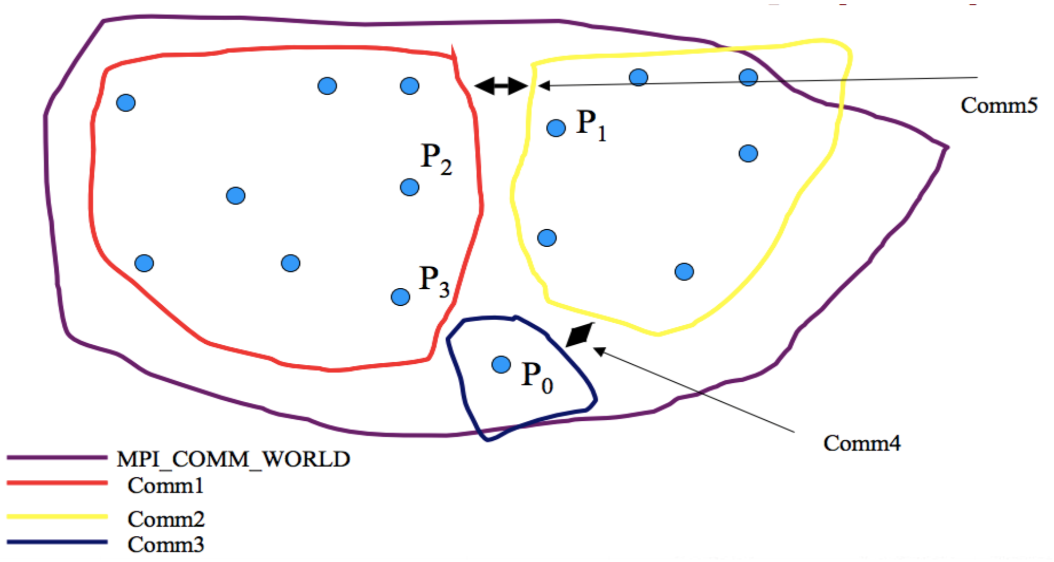
- Groups allow collective operations to work on a subset of processes

- Some special groups
  - ✓ **MPI_GROUP_EMPTY** – The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
  - ✓ **MPI_GROUP_NULL** – Returned when a group is freed

Communicators

- A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes
- An **intra-communicator** is used to communicate within a group and has two main attributes
  - ✓ The process group
  - ✓ The **topology** (logical layout of processes) (we'll cover topologies later)
- An **inter-communicator** is used to communicate between **disjoint groups** of processes and has two attributes
  - ✓ A pair of process groups
  - ✓ No **topology**
- Communicators can also have user-defined attributes

Communicators and Groups



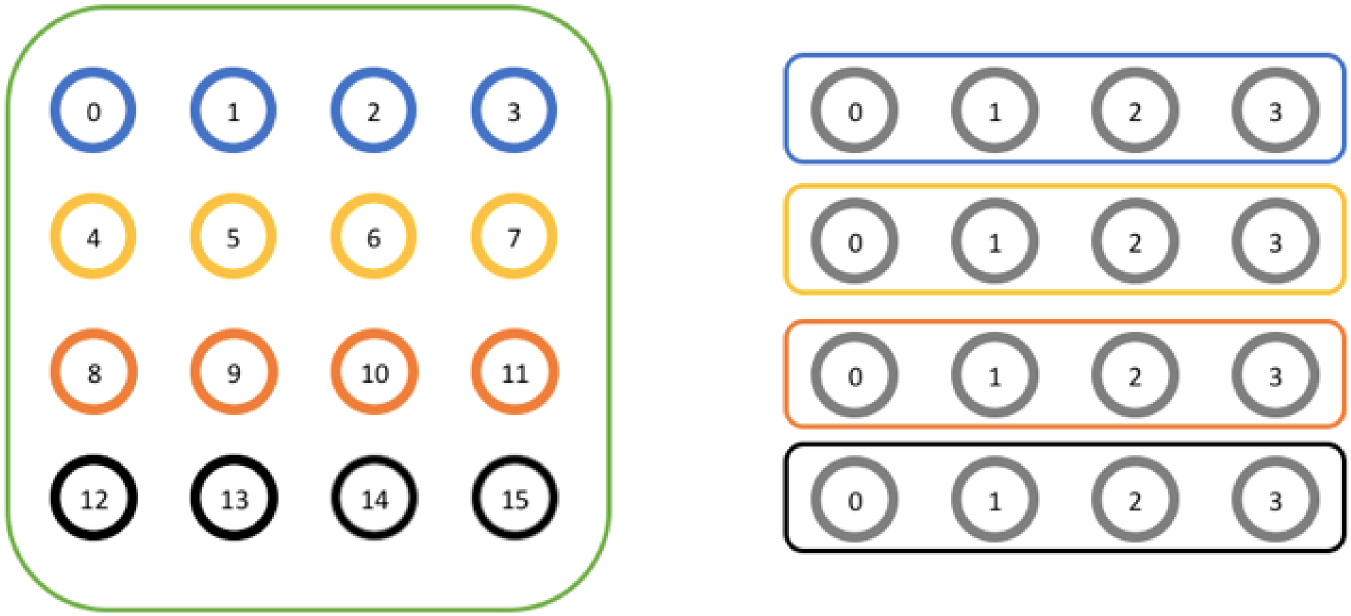
- There are 4 distinct groups. These are associated with intracommunicators: MPI_COMM_WORLD, comm1, and comm2, and comm3.
- P3 is a member of 2 groups and may have different ranks in each group (say 3 & 4).
- If P2 wants to send a message to P1 it can use MPI_COMM_WORLD (intracommunicator).
- If P2 wants to send a message to P3, it can use MPI_COMM_WORLD (send to rank 3) or comm1 (send to rank 4).
- P0 can broadcast a message to all processes associated with comm2 by using intercommunicator comm5.

Communicators

Functionality	Intra-communicator	Inter-communicator
Number of groups	1	2
Communication safety	Yes	Yes
Collective operations	Yes	No
Topologies	Yes	No
Caching (user-defined data)	Yes	Yes

Example of Communicator

- Split a single global communicator into a set of smaller communicators.
- In the image below, you can see how each group of processes with the same color on the left ends up in its own communicator on the right.



Communication Domains

Rationale

- Any point-to-point or collective communication occurs in MPI within a **communication domain**.
- Such a communication domain is represented by a set of communicators with **consistent** values, one at each of the participating processes; each communicator is the local representation of the global **communication domain**.
- If this domain is for **intra-group** communication then all the communicators are **intra-communicators**, and all have the same group attribute.
- Each communicator identifies all the other corresponding communicators.

Communication Domains

- Given by a set of **communicators** (one at each process) each with the same number of processes (representing the group)

- Allows the address for the ‘**1**’ process in a group to be **logically equivalent** for all processes but **physically different**
  - ✓ And importantly, hidden from the **user**

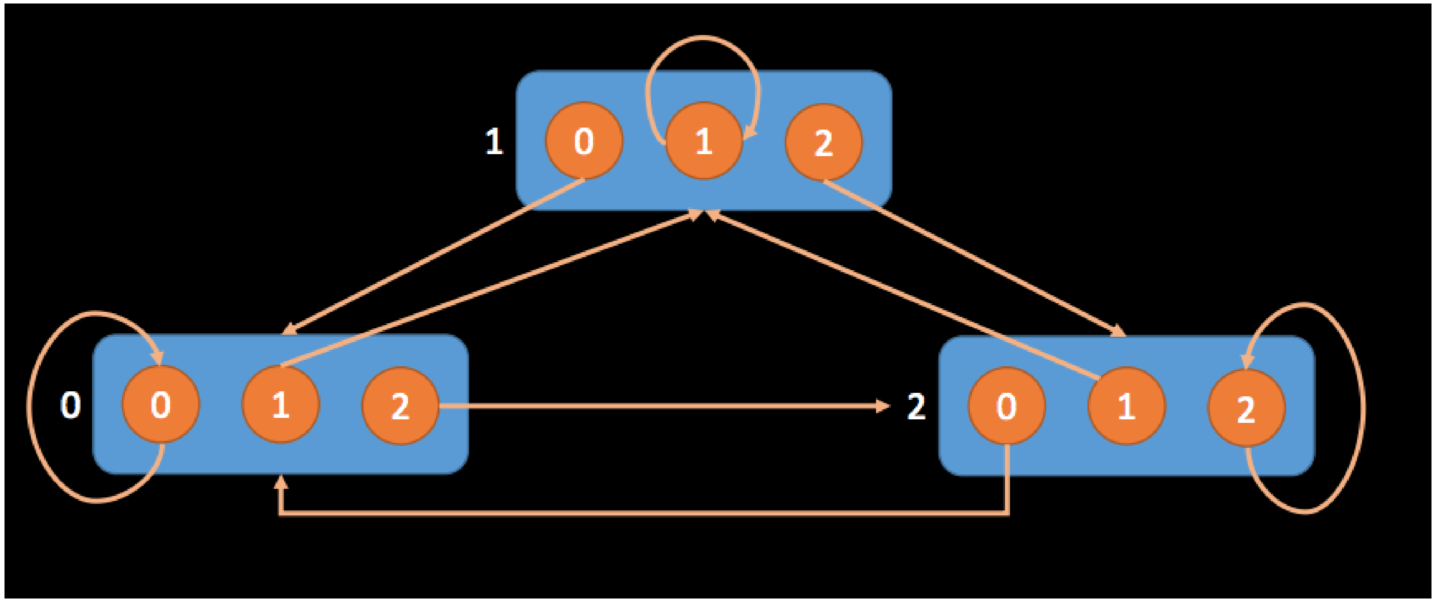
- If we take all communication domains together we get a complete communication graph

Communication Domains

THE UNIVERSITY OF WESTERN AUSTRALIA

Communication Domain – Example

MPI_COMM_WORLD for three nodes



Outline

- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

Group Management

- As will be clear, groups are initially not associated with communicators
- Groups can only be used for message passing within a communicator.
- We can **access** groups, **construct** groups, and **destroy** groups

Group Accessors

- MPI_GROUP_SIZE(group, size)
  - ✓ This routine returns the **number of processes** in the group

- MPI_GROUP_RANK(group, &rank)
  - ✓ This routine returns the **rank** of the calling process

- MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)
  - ✓ This routine takes an array of n ranks (ranks1) which are **ranks** of processes in group1. It returns in ranks2 the corresponding ranks of the processes as they are in group2

Group Accessors

- MPI_GROUP_COMPARE(group1, group2, result)
  - ✓ This routine returns the **relationship** between group1 and group2
  - ✓ If group1 and group2 contain the same processes, ranked the same way, this routine returns **MPI_IDENT**
  - ✓ If group1 and group2 contain the same processes, but ranked differently, this routine returns **MPI_SIMILAR**
  - ✓ Otherwise this routine returns **MPI_UNEQUAL**

Group Constructors

- Group constructors are used to create new groups from existing groups
- Base group is the group associated with **MPI_COMM_WORLD** (use **MPI_Comm_Group** to get this)
- Group creation is a local operation
  - ✓ **No communication** needed
- Following group creation, no communicator is associated with the group
  - ✓ No communication possible with new group
- Each process in a new group **MUST** create the group so it is identical!
- Groups are created through some communicator creation routines(covered later)

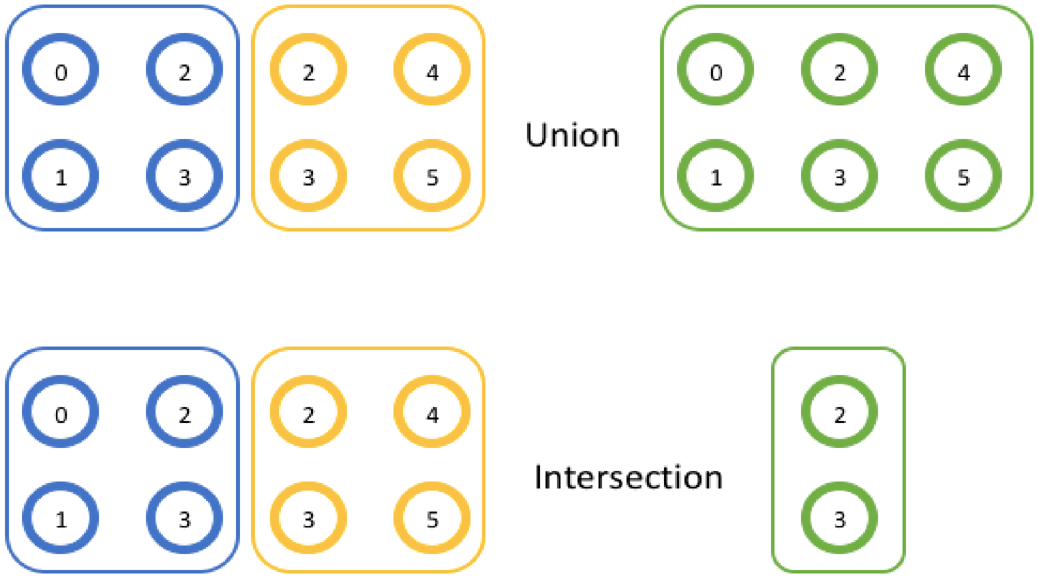
Group Constructors

- MPI_COMM_GROUP(comm, group)
  - ✓ Returns the **group** corresponding to the communicator
- MPI_GROUP_UNION(group1, group2, newgroup)
  - ✓ Newgroup will contain a **group of all processes** in group1 and group2
- MPI_GROUP_INTERSECTION(group1, group2, newgroup)
  - ✓ Newgroup will contain the processes in both groups 1 and 2
- MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
  - ✓ Newgroup will contain the **set** difference between groups 1 and 2

Group Constructors

Union/Intersection Example

- In the first example, the union of the two groups **{0, 1, 2, 3}** and **{2, 3, 4, 5}** is **{0, 1, 2, 3, 4, 5}** because each of those items appears in each group.
- In the second example, the intersection of the two groups **{0, 1, 2, 3}**, and **{2, 3, 4, 5}** is **{2, 3}** because only those items appear in each group.



Group Destruction

- MPI_GROUP_FREE(group)
  - ✓ Returns **MPI_GROUP_NULL**

Outline

- Point-to-Point (continued)
  - ✓ Extended Example
  - ✓ Performance Analysis
- Collective Communication
  - ✓ Introduction
  - ✓ Barrier Synchronisation
  - ✓ Global Communication
  - ✓ Global Reductions
- Communicator
  - ✓ Introduction
  - ✓ Group Routines
  - ✓ Communicator Routines
  - ✓ Summary

Communicator Management

- Communicator access operations are local, thus requiring **no inter-process** communication
- Communicator constructors are **collective** and may require **inter-process** communication
- All the routines in this section are for **intra-communicators**, inter-communicators will be covered **separately**

Communicator Accessors

- MPI_COMM_SIZE(comm, size)
  - ✓ Returns the **number** of processes in the rank
- MPI_COMM_RANK(comm, rank)
  - ✓ Returns the **rank** of the calling process in that communicator
- MPI_COMM_COMPARE(comm1, comm2, result) returns
  - ✓ **MPI_IDENT** if comm1 and comm2 are handles for the same object
  - ✓ **MPI_CONGRUENT** if comm1 and comm2 have the same group attribute
  - ✓ **MPI_SIMILAR** if the groups associated with comm1 and comm2 have the same members but in different rank order
  - ✓ **MPI_UNEQUAL** otherwise

Communicator Constructors

- MPI_COMM_DUP(comm, newcomm)
  - ✓ **Duplicates** the provided communicator (useful to copy and then manipulate)
- MPI_COMM_CREATE(comm, group, newcomm)
  - ✓ Creates a new **intra-communicator** using a **subset** of comm
- MPI_COMM_SPLIT(comm, color, key, newcomm)
  - ✓ Creates **separate** communicators where processes passing the same ‘**color**’ are grouped together
  - ✓ This is a rather exotic one and is worth thinking about carefully
  - ✓ Useful to segment processes into distinct subtasks

Communicator Example

- Split processes with odd and even ranks into 2 communicators

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
   int myid, numprocs;
   int color, broad_val, new_id, new_nodes;
   MPI_Comm New_Comm;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   color = myid%2;
   MPI_Comm_split(MPI_COMM_WORLD, color, myid, &New_Comm);
   MPI_Comm_rank(New_Comm, &new_id);
   MPI_Comm_size( New_Comm, &new_nodes);

   if(new_id == 0) broad_val = color;
   MPI_Bcast(&broad_val, 1, MPI_INT, 0, New_Comm);
   printf("Old_proc%d has new rank %d recevied value %d", myid, new_id, broad_val);
   MPI_Finalize();
}
```

Summary

- Collective communication can simplify many common patterns
  - ✓ **Broadcast/Reduce, Scatter/Gather**

- Collective communication is also dependent on the communicator supplied

- Communicators can be used to **separate** processes into **separate** jobs

- Communicators are created from **groups**

References

- Readings
  - Measuring Elapsed Time for OpenMP Programs
  - Introduction to Groups and Communicators

Copyright Notice