MPI and Collective Communication Patterns

Chris Kauffman

Last Updated:
Tue Feb 17 01:57:20 PM EST 2026

Logistics

Reading: Grama Ch 6 + 4

- Ch 6: MPI basics
- Ch 4: Communication patterns

Assignments

- A1 grading has commenced
- A2 will go up soon, feature MPI Coding

Today

- More MPI programming
- Discuss Comm. Patterns

Thursday

- 2:00-2:30: Mini-Exam 1
- 2:30-3:15: Lecture

Mini-Exam Logistics

- 30 minutes long, 1 page front/back 3-4 problems
- Open Resource: Review rules posted on schedule
- Valid Topics are
  - A1 Material
  - Task Dependency Graphs and Computed Stats from them
  - Basic Distributed Memory Architecture
  - Basic Parallel Algorithm Design via Partitioning Input and Output
  - MPI Send / Receive Semantics and Code
- Study A1 and in-class exercises as review

Exercise: MPI Basics Review

1. What are the two basic operations required for distributed memory parallel programming?
2. Describe some variants for these operations.
3. What does "MPI" stand for and what is it?
4. How do the two basic operations look in MPI?
5. How does one compile/run programs with MPI?

Answers: MPI Basics Review

1. `send(data,count,dest)` and `receive(data,count,source)` are the two essential ops for distributed parallel programming, transfer data between processors
2. `send/receive` can be
   - *blocking or non-blocking*: whether call waits for the partner to complete the transaction or instead immediately moves ahead
   - *buffered or unbuffered*: whether a special area of memory is used to facilitate the sends more efficiently
3. MPI: The Message Passing Interface, common distributed memory programming library
4. Send and Receive in MPI

```
MPI_Send(buf, len, MPI_INT, dest,   MPI_COMM_WORLD);
MPI_Recv(buf, len, MPI_INT, source, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

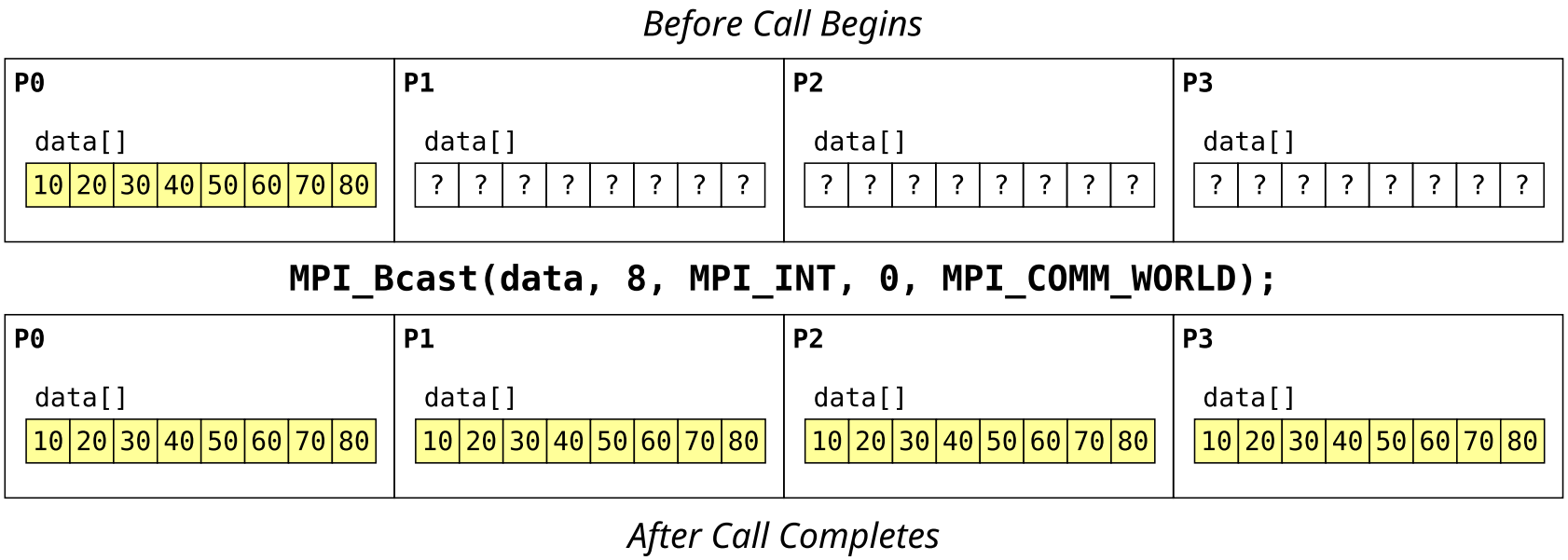
5. Compile/Run

```
mpicc -o prog parallel_program.c
mpirun -np 8 prog
```

Patterns of Communication

- Common patterns exist in many algorithms
- Reasoning about algorithms easier if these are "primitives"
  - "I'll use a loop here to send this data to every processor and a loop here for every processor to send its data to proc 0 which needs all of it."
    vs
    "I'll **Broadcast** to all procs here and **Gather** results here"
- MPI provides a variety of collective communication operations which make these single function calls
- Vendors of super-computers usually implement those functions to run as quickly as possible on the network provided - repeated halving/double if the network matches
- By making the function call, you get all the benefit the network can provide in terms of speed AND have simpler, portable code
- Learn Collective Comm Patterns to elevate parallel algorithms

Broadcasting One to All



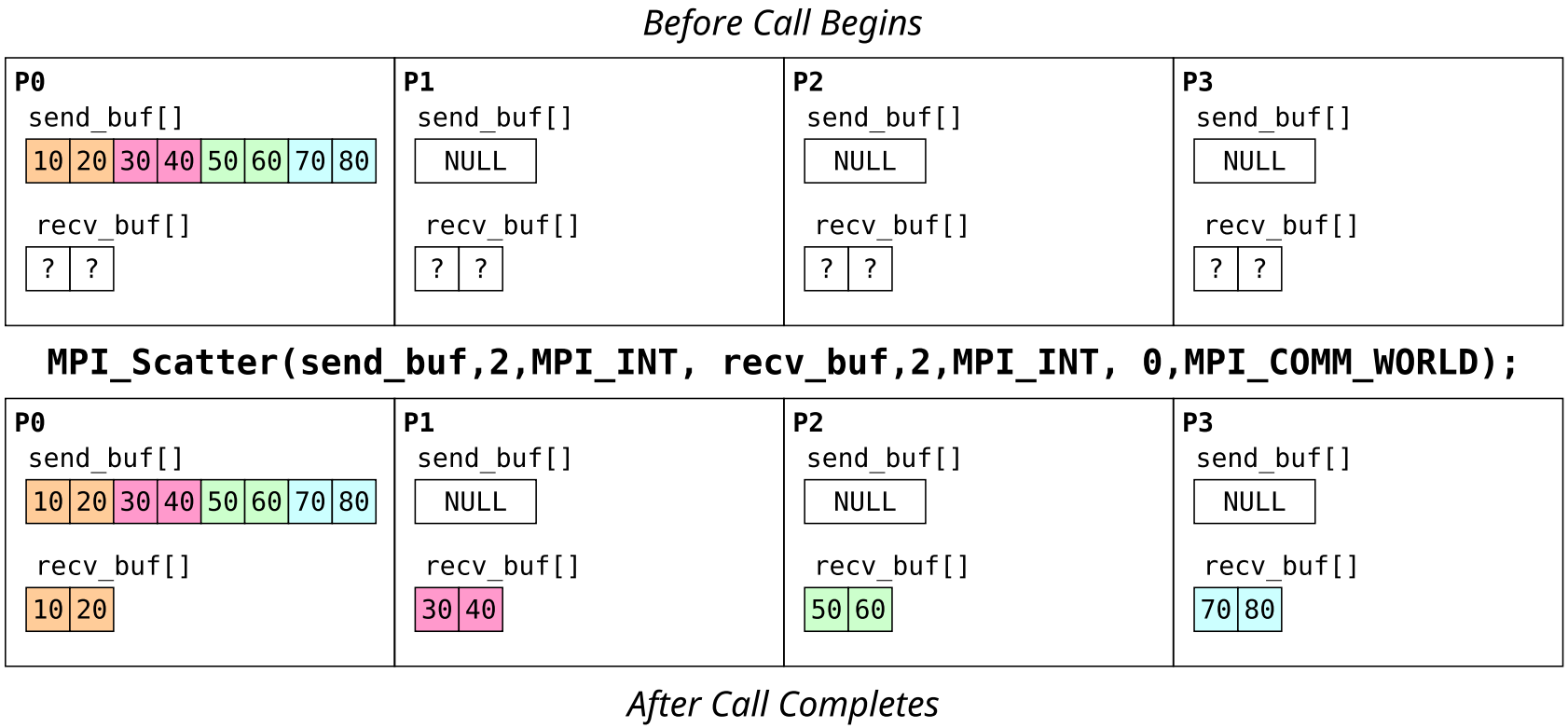
- Root processor wants to transmit `data[]` buffer to all processors
- Broadcast distributes to all procs
- Each proc gets same stuff in `data[]` buffer
- `data[]` may be as small as a single element (e.g. broadcast single variable value)

Broadcast Example Code

```
// collective_broadcast_demo.c:
// Everyone allocates
int *data = malloc(sizeof(int) * num_elements);

// Root fills data by reading from file/computation
if(procid == root_proc){
  for(i=0; i<num_elements; i++){
    data[i] = i*i;
  }
}

// Everyone calls broadcast, root proc sends, others receive
MPI_Bcast(data, num_elements, MPI_INT, root_proc,
          MPI_COMM_WORLD);
// data[] now filled with same portion of root_data[] on each proc
```

NOTE: These are Broadcast mechanics; in a real setting, if processors can compute the `data[]` array themselves, they should to avoid the need to communicate.

Scatter from One to All



- Root processor has slice of data for each proc
- Scatter distributes to each proc
- Each proc gets an individualized message

Scatter Example

```
// collective_scatter_demo.c:
// Root allocates/fills root_data by reading from file/computation
if(procid == root_proc){
  send_buf = malloc(sizeof(int) * total_elements);
  for(i=0; i<total_elements; i++){
    send_buf[i] = i*i;
  }
 }

// Everyone allocates for their share of data including root
recv_buf = malloc(sizeof(int) * elements_per_proc);

// Everyone calls scatter, root proc sends, others receive
MPI_Scatter(send_buf, elements_per_proc, MPI_INT,
            recv_buf, elements_per_proc, MPI_INT,
            root_proc, MPI_COMM_WORLD);
// recv_buf[] now filled on each proc with unique portion from send_buf
```

Exercise: Scatter a Matrix

Often have Matrix and Vector data in HPC / Parallel Computing

```
// mat vec multiply
double **mat = ...;
...;
mat[i][j] = ...;
double *vec = ...;
double *out = ...;
for(int i=0; i<rows; i++){
  for(int j=0; j<cols; j++){
    out[i] += mat[i][j]*vec[j];
  }
}
```

- How can one `MPI_Scatter()` the rows of a matrix?
- What assumptions must be true about the matrix data?

Answers: Scatter a Matrix

- Typically matrix must be allocated in **one block of memory** - single `malloc()`
- Allows a single `MPI_Scatter()` to scatter groups of rows

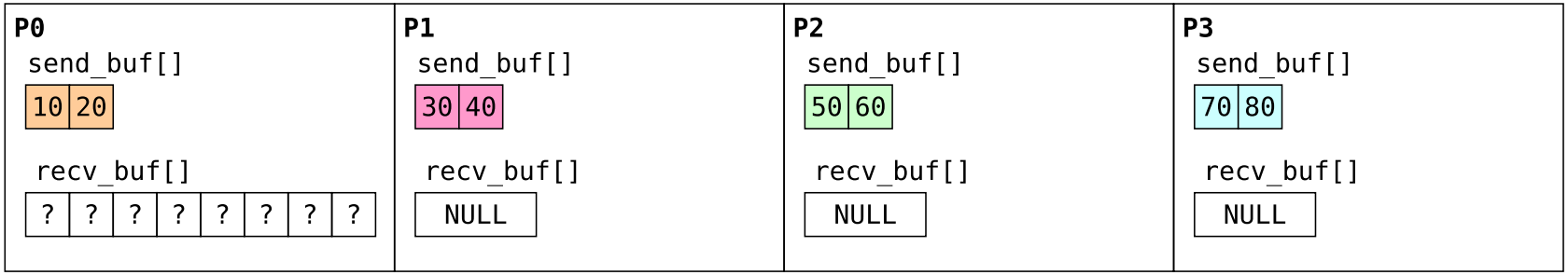
```
{
  // allocate data for all of matrix
  double *all = malloc(rows*cols * sizeof(double));

  // allocate / assign row pointers within single block
  double **mat = malloc(rows * sizeof(double*));
  for(int i=0; i<rows; i++){
    mat[i] = &all[i*cols];
  }

  mat[i][j] = 5.5;                    // assign via row pointer
}
```

Answers: Scatter a Matrix

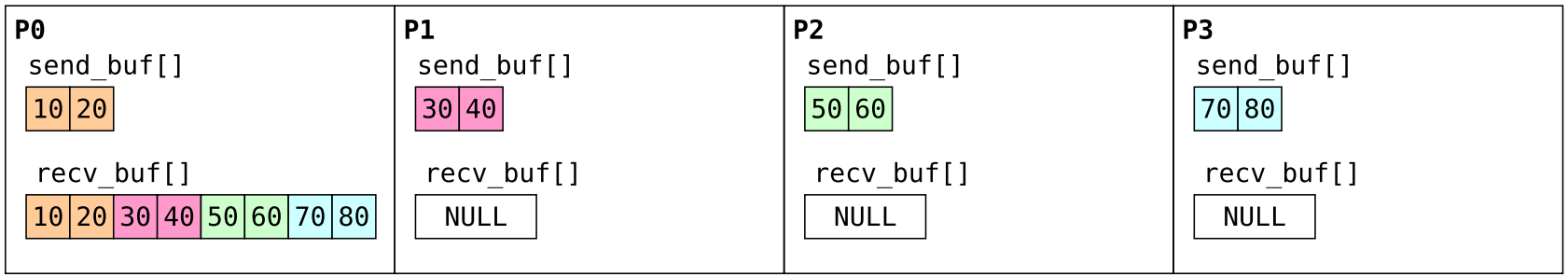
```
{
  double *all = NULL;

  // root reads in matrix rows
  if(rank == root_proc){
    all = malloc(rows*cols * sizeof(double));
    fread(all, sizeof(double), rows*cols, infile);
  }

  // set up and perform scatter
  int rows_per_proc  = rows / nprocs;
  int elems_per_proc = rows_per_proc * cols;
  double *myrows = malloc(sizeof(double) * elems_per_proc);
  MPI_Scatter(all,    elements_per_proc, MPI_INT,
              myrows, elements_per_proc, MPI_INT,
              root_proc, MPI_COMM_WORLD);
}
```

Gather from All to One

Before Call Begins



`MPI_Gather(send_buf,2,MPI_INT, recv_buf,2,MPI_INT, 0,MPI_COMM_WORLD);`



After Call Completes

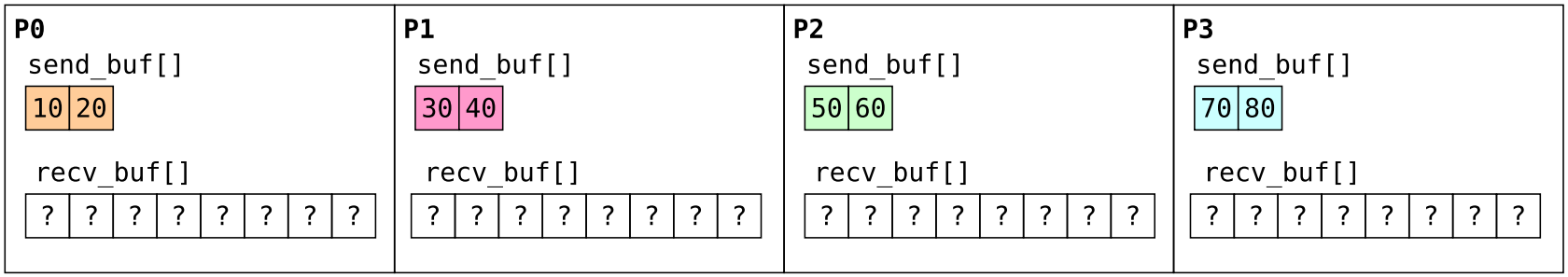
- Every processor has data in `send` buffer
- Root processor needs all data ordered by `proc_id`
- Root ends with all data in a `receive` buffer

Gather Example

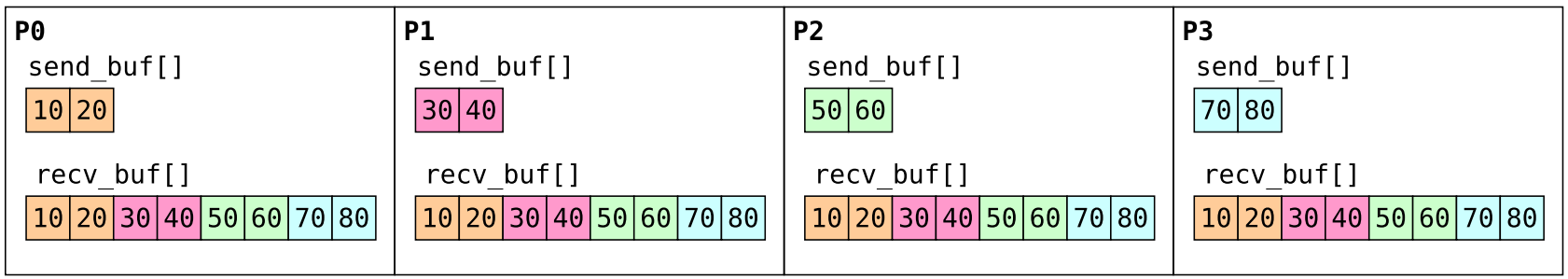
```
// collective_gather_demo.c:
int total_elements = 16;
int elements_per_proc = total_elements / total_procs;

// Everyone allocates for their share of data including root
send_buf = malloc(sizeof(int) * elements_per_proc);

// Each proc fills data[] with "unique" values
int x = 1;
for(i=0; i<elements_per_proc; i++){
  send_buf[i] = x;
  x *= (procid+2);
 }
// data[] now filled with unique values on each proc

// Root allocates root_data to be filled with gathered data
if(procid == root_proc){
  recv_buf = malloc(sizeof(int) * total_elements);
}

// Everyone calls gather, root proc receives, others send
MPI_Gather(send_buf, elements_per_proc, MPI_INT,
           recv_buf, elements_per_proc, MPI_INT,
           root_proc, MPI_COMM_WORLD);
// recv_buf[] now contains each procs send_buf[] in order on root
```

All-Gather: Everyone to Everyone

Before Call Begins




```
MPI_Allgather(send_buf,2,MPI_INT, recv_buf,2,MPI_INT, 0,MPI_COMM_WORLD);
```



After Call Completes

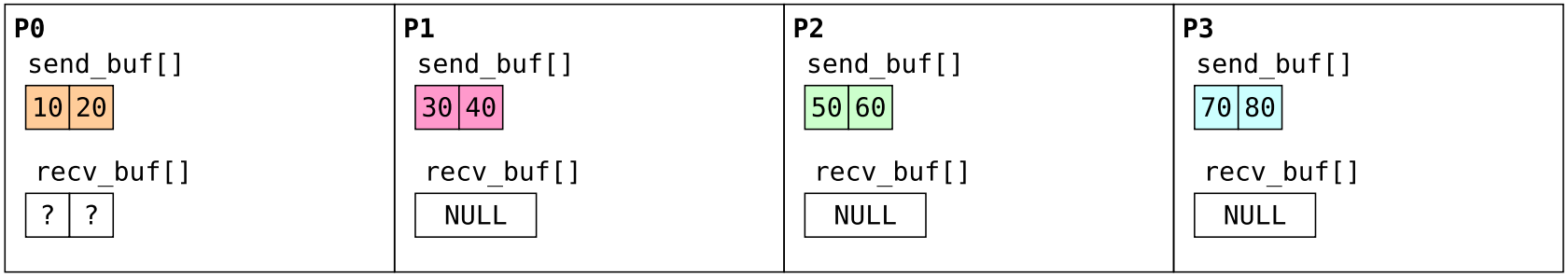
- Every processor has data in `send` buffer
- **All** processors need all data ordered by `proc_id`
- All procs end with all data in `receive` buffer

All-Gather Example

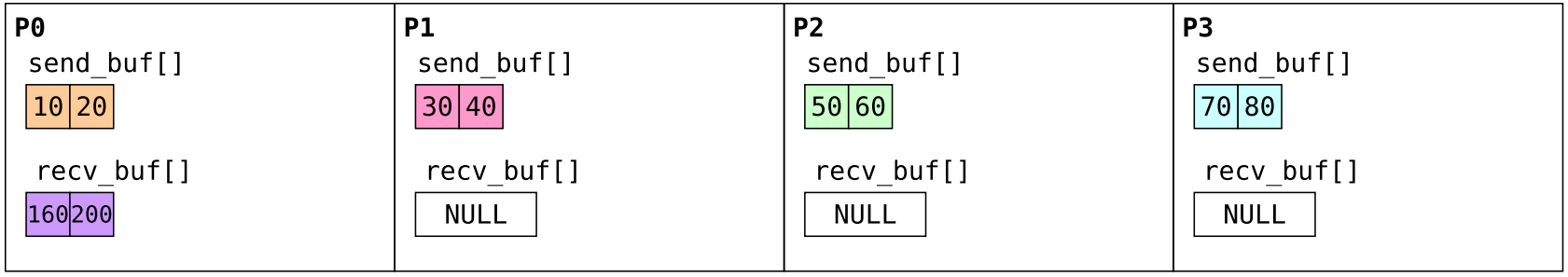
```
// collective_allgather_demo.c
// Everyone allocates for their share of data including root
send_buf = malloc(sizeof(int) * elements_per_proc);

// Each proc fills send_buf[] with "unique" values
int x = 1;
for(i=0; i<elements_per_proc; i++){
  send_buf[i] = x;
  x *= (proc_id+2);
}
// send_buf[] now filled with unique values on each proc

// Everyone allocates all_data to be filled with gathered data
recv_buf = malloc(sizeof(int) * total_elements);

// Everyone calls all-gather, everyone sends and receives
MPI_Allgather(send_buf, elements_per_proc, MPI_INT,
              recv_buf, elements_per_proc, MPI_INT,
              MPI_COMM_WORLD);
// recv_buf[] now contains each procs data in order on all procs
```

Reduction: All to One

Before Call Begins



`MPI_Reduce(send_buf,recv_buf,2,MPI_INT, MPI_SUM,0,MPI_COMM_WORLD);`



After Call Completes

- Every processor has data in `send` buffer
- Root processor needs all data **reduced**
  - Reduction operation is transitive
  - Several pre-defined via constants
  - Common: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`
- Root ends with reduced data in `receive` buffer

Reduce Example

```
// collective_reduce_demo.c:
{ // Each proc fills send_buf[] with unique values
  int x = 1;
  for(i=0; i<total_elements; i++){
    send_buf[i] = x;
    x *= (procid+2);
  }
  // send_buf[] now filled with unique values on each proc

  // Root allocates recv_buf[] to be filled with reduced data
  if(procid == root_proc){
    recv_buf = malloc(sizeof(int) * total_elements);
  }

  // Everyone calls reduce, root proc receives,
  // others send and accumulate
  MPI_Reduce(send_buf, recv_buf, total_elements, MPI_INT,
             MPI_SUM, // operation to perform on each element
             root_proc, MPI_COMM_WORLD);
  // recv_buf[] now contains each procs send data summed up
}
```

Note: Reduction's Array Argument

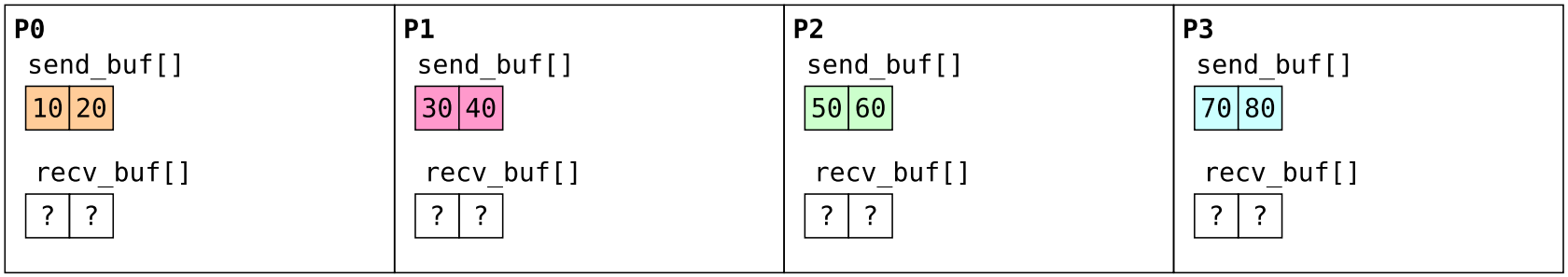
- `MPI_Reduce()` works on a `data[]` argument like others
- Reduction happens for each element so that

```
root_proc = 0;
MPI_Reduce(send_buf, recv_buf, total_elements, MPI_INT,
           MPI_SUM, root_proc, MPI_COMM_WORLD);
// results in
P0.recv_buf[0] = P0.send_buf[0]+P1.send_buf[0]+P2.send_buf[0]+...
P0.recv_buf[1] = P0.send_buf[1]+P1.send_buf[1]+P2.send_buf[1]+...
P0.recv_buf[2] = P0.send_buf[2]+P1.send_buf[2]+P2.send_buf[2]+...
...
```

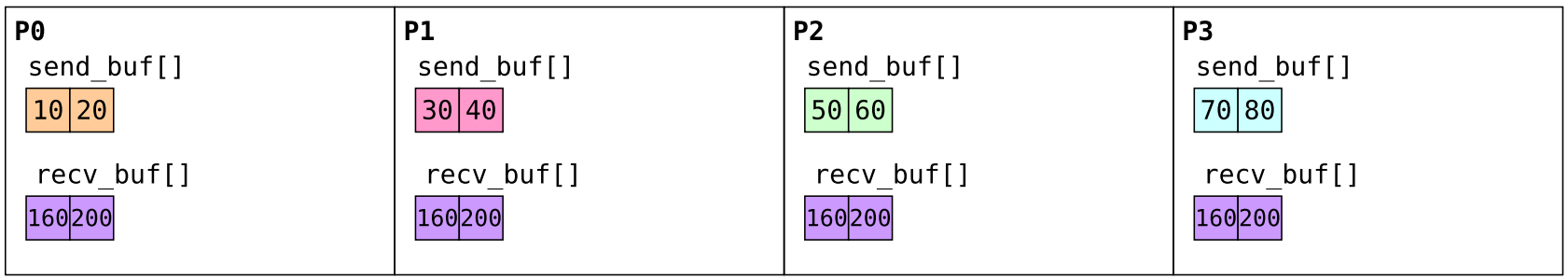
- To get a single sum, Procs should iterate on their own array THEN `MPI_Reduce()` on a single vlaue

Reduction for All: All-Reduce

Before Call Begins



`MPI_Allreduce(send_buf,recv_buf,2,MPI_INT, MPI_SUM,MPI_COMM_WORLD);`



After Call Completes

- Every processor has data in `send_buf[]`
- All processors need all data **reduced**
- All procs end with reduced data in a `recv_buf[]`

Allreduce Example

```
{ // Each proc fills send_buf[] with unique values
  int x = 1;
  for(i=0; i<total_elements; i++){
    send_buf[i] = x;
    x *= (procid+2);
  }
  // send_buf[] now filled with unique values on each proc

  // Everyone allocates recv_buf[] to be filled with reduced data
  recv_buf = malloc(sizeof(int) * total_elements);

  // Everyone calls reduce, everyone sends and receives
  MPI_Allreduce(send_buf, recv_buf, total_elements, MPI_INT,
                MPI_SUM, // operation to perform on each element
                MPI_COMM_WORLD);
  // recv_buf[] now contains sum of each procs send_buf[]
}
```

In-place Reduction

- Occasionally want to do reductions in-place: send and receive buffers are the same.
- May be useful in upcoming assignment
- Use `MPI_IN_PLACE` for the send buffer

```
{ // Everyone calls reduce, everyone sends and receives
  MPI_Allreduce(MPI_IN_PLACE,   // no destination buffer - use data
                data,           // reduction is placed here
                total_elements, MPI_INT,
                MPI_SUM,        // op to perform on each element
                MPI_COMM_WORLD);
  // data[] now contains each procs data[], min elements
}
```

Summary of Collective Communications

Operation	MPI Function	Synopsis
		Individual
Send	`MPI_Send`	One-to-one send
Receive	`MPI_Recv`	One-to-one receive
Send/Receive	`MPI_Sendrecv`	One-to-one send/receive
		Collective
Barrier	`MPI_Barrier`	All wait for stragglers
Broadcast	`MPI_Bcast`	Root to all, all data copied
Scatter	`MPI_Scatter`	Root to all, slices of data copied
Gather	`MPI_Gather`	All to root, slices ordered on Root
Reduce	`MPI_Reduce`	All to root, data reduced on Root
All-Gather	`MPI_Allgather`	All to all, data ordered
All-Reduce	`MPI_Allreduce`	All to all, data reduced
		Not Discussed
Prefix	`MPI_Prefix`	All-to-all, data ordered/reduced
All-to-AllP	`MPI_Alltoall`	All-to-all, personal messages

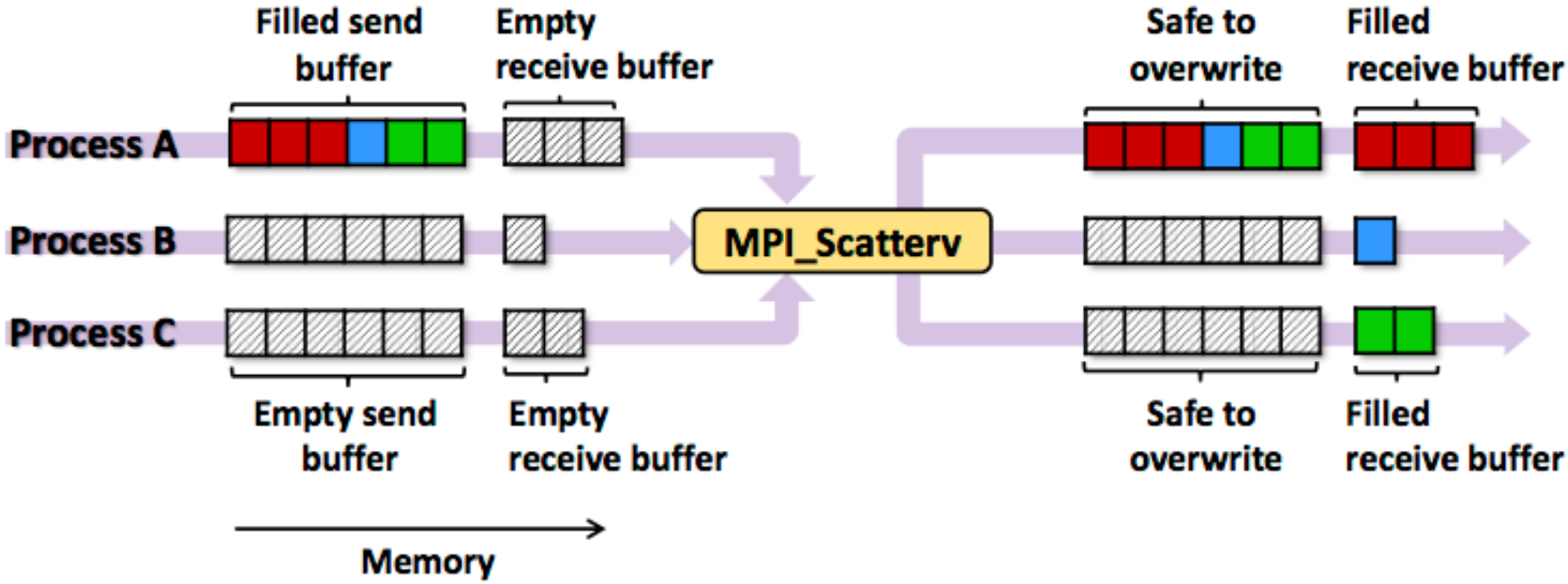
Vector Versions

- Collective comm ops like `MPI_Scatter()` assume same amount of data to/from each processor
- Not a safe, general assumption (e.g. `len % P != 0`)
- *Vector*[1] versions of each comm op exist which relax these assumptions, allow arbitrary data counts per proc
- Provide additional arguments indicating
  - `counts[]`: How many elements each proc has
  - `displs[]`: Offsets elements are/will be stored in master array

Operation	Equal counts	Different counts
Broadcast	`MPI_Bcast()`	
Scatter	`MPI_Scatter()`	`MPI_Scatterv()`
Gather	`MPI_Gather()`	`MPI_Gatherv()`
All-Gather	`MPI_Allgather()`	`MPI_Allgatherv()`
Reduce	`MPI_Reduce()`	
All-Reduce	`MPI_Allreduce()`	

[1] "Vector" here means extra array arguments, NOT hardware-level parallelism like "Vector Instruction"

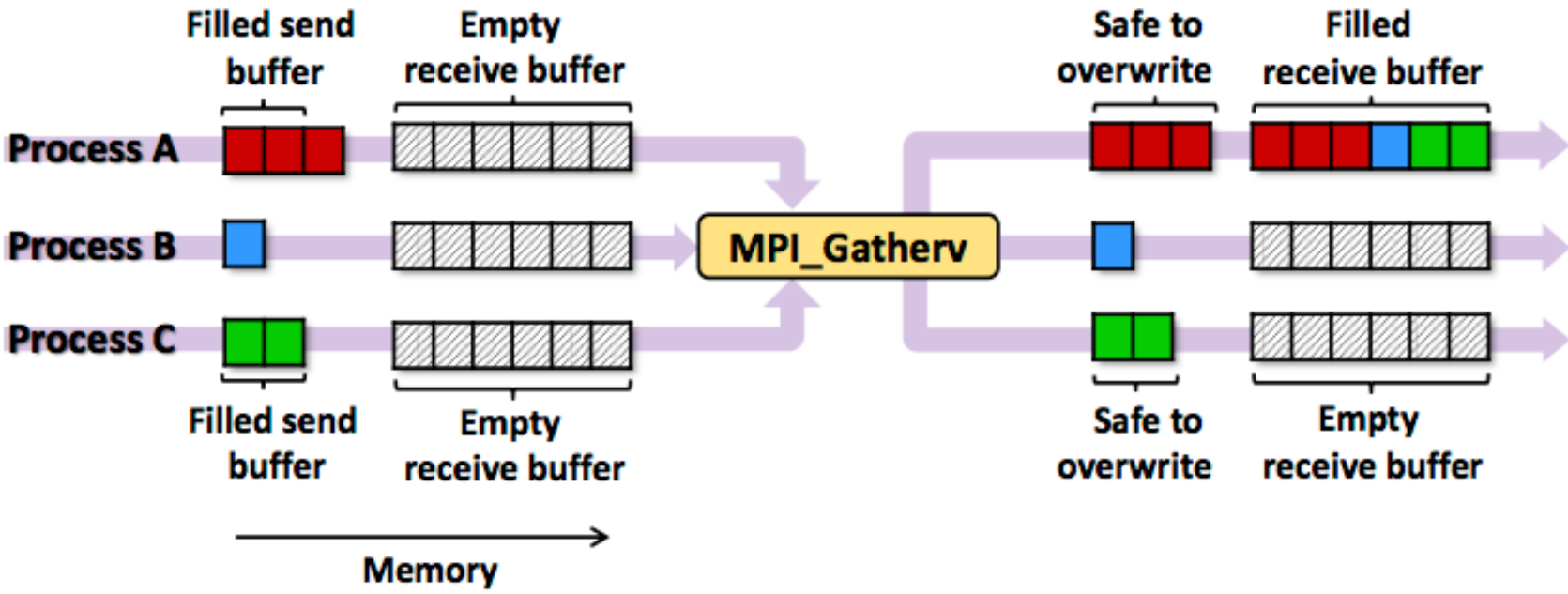
MPI_Scatterv Example



Source: SKIRT Docs

```
//                P0 P1 P2
int counts[] = { 3, 1, 2};
int displs[] = { 0, 3, 4};
//                 P0  P0  P0  P1  P2  P2
int send[] =    { 10, 20, 30, 40, 50, 60 };
int *recv = malloc(counts[rank] * sizeof(int));
MPI_Scatterv(send, counts, displs, MPI_INT,
             recv, counts[rank], MPI_INT,
             0, MPI_COMM_WORLD);
```

MPI_Gatherv Example



Source: SKIRT Docs

```
int total = 6;
int counts[] = { 3, 1, 2};
int displs[] = { 0, 3, 4};
int send[counts[rank]];
int *recv, i;
for(i=0; i<counts[rank]; i++){
  send[i] = rank*(i+1);
}
```

```
recv = (rank!=0) ? NULL :
  malloc(total * sizeof(int));

MPI_Gatherv(
 send, counts[rank],   MPI_INT,
 recv, counts, displs, MPI_INT,
 0, MPI_COMM_WORLD);
```

Dynamic Count and Displacements for Vector Comm Ops

- Common prob: # of procs does not evenly divide data size
- Use the vector versions of collective ops
- To calculate counts and displacements and spread work evenly, use a pattern like the below (see `collective_scatterv_demo.c`)

```
int total_elements = 16;
int *counts = malloc(total_procs * sizeof(int));
int *displs = malloc(total_procs * sizeof(int));

// Divide total_elements as evenly as possible: lower numbered
// processors get one extra element each.
int elements_per_proc = total_elements / total_procs;
int surplus           = total_elements % total_procs;
for(i=0; i<total_procs; i++){
  counts[i] = (i < surplus) ? elements_per_proc+1 : elements_per_proc;
  displs[i] = (i == 0) ? 0 : displs[i-1] + counts[i-1];
}
// counts[] and displs[] now contain relevant data for a scatterv,
// gatherv, all-gatherv calls
```

Barriers

```
MPI_Barrier(MPI_COMM_WORLD);
```

- Causes all processors to synchronize at the given line of code
- Early arrivers idle while other procs catch up
- To be avoided if possible as it almost always incurs idle time
- Unavoidable in some select scenarios
- Can be useful in debugging to introduce barriers

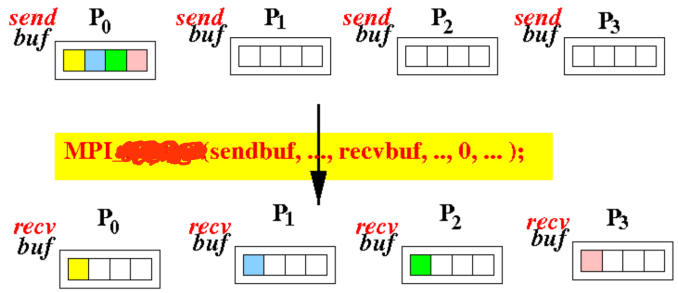
Basic Debugging Discpline

Q: How do I debug Open MPI processes in parallel?
A: This is a difficult question...
– *OpenMPI FAQ on Debugging*

- Commercial Parallel Debuggers exist, TotalView is popular
- For small-ish programs...
  Debug Printing + Valgrind + Effort + Patience
  will usually suffice

```
> mpirun -v -np 4 valgrind ./my_program arg1 arg2
```

Exercise: MPI Collective Comm Review



1. Which MPI Collective Communication Operation does the above picture represent?
2. Draw a similar picture for MPI All-Gather
3. What are common operations work with a Reduction?
4. Which collective communication operations would be useful in the following settings:
   - At the beginning of a computation, the root processor needs to distribute rows of a matrix read from a data file to all other processors
   - After each processor finishes some computations using its own rows, all processors need the sum of all columns in the matrix

Answers: MPI Collective Comm Review

1. Which MPI Collective Communication Operation does the above picture represent?
   Scatter / `MPI_Scatter`
2. Draw a similar picture for MPI All-Gather
   See slide 15
3. What are common operations work with a Reduction?
   Addition/Sum, Multiply/Product, Min, Max
4. Which collective communication operations would be useful in the following settings:
   - At the beginning of a computation, the root processor needs to distribute rows of a matrix read from a data file to all other processors
     Scatter the rows
   - After each processor finishes some computations using its own rows, all processors need the sum of all columns in the matrix
     Local sum of columns, All-Reduce on local Column sums