MPI and Collective Communication Patterns

Chris Kauffman

Last Updated:
Tue Feb 8 09:37:37 AM CST 2022

Logistics

Reading: Grama Ch 6 + 4

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

Assignments

A2 will go up soonand feature MPI Coding

Today

- ▶ More MPI programming
- ▶ Discuss Comm. Patterns

Thursday

- ▶ 40-min lecture
- ▶ 35-min Mini-Exam 1

Exercise: MPI Basics Review

- ▶ What are the two basic operations required for distributed memory parallel programming?
- ▶ Describe some variants for these operations.
- ▶ What is a very common library for doing distributed parallel programming?
- ▶ How do the two main operations look in that library?
- ▶ How does one compile/run programs with this library?

Answers: MPI Basics Review

- ▶ `send(data,count,dest)` and
  `receive(data,count,source)` are the two essential ops for
  distributed parallel programming
- ▶ `send/receive` can be
  - ▶ blocking: wait for the partner to link up and complete the
    transaction
  - ▶ non-blocking: don't wait now but check later to before
    using/changing the message data
  - ▶ buffered: a special area of memory is used to facilitate the
    sends more efficiently
- ▶ MPI: The Message Passing Interface, common distributed
  memory programming library
- ▶ 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);
  ```
- ▶ 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 broadcast to all procs here and gather all results here" vs
    "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."
- ▶ 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

Broadcasting One-to-All

MPI_Bcast(buf, ..., 0, ...);
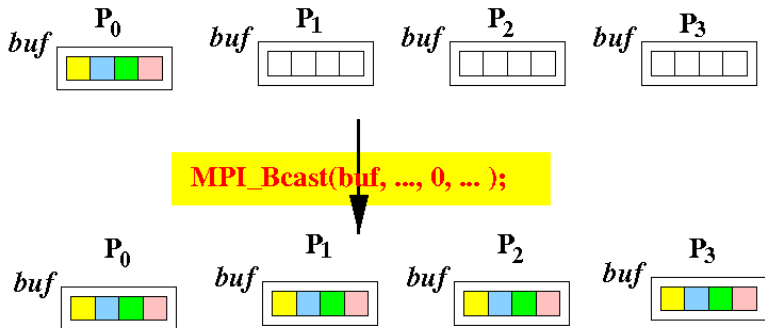
Source: Shun Yan Cheung Notes on MPI

- ▶ Root processor wants to transmit `data` buffer to all processors
- ▶ Broadcast distributes to all procs
- ▶ Each proc gets same stuff in `data` buffer

6

Broadcast Example Code

In `broadcast_demo.c`

```c
// Everyone allocates
data = (int*)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
```

Scatter from One To All
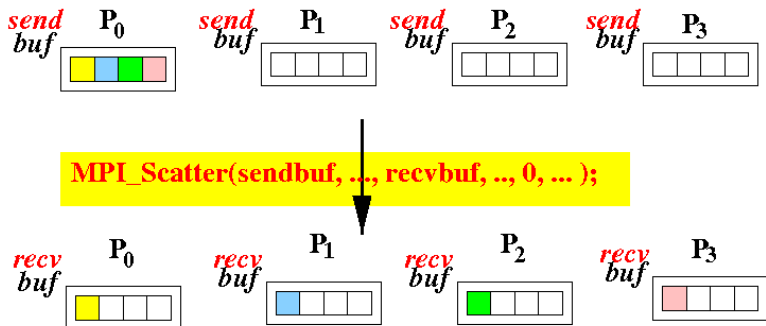
Source: Shun Yan Cheung Notes on MPI

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

Scatter Example

In scatter_demo.c

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

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

// Everyone calls scatter, root proc sends, others receive
MPI_Scatter(root_data, elements_per_proc, MPI_INT,
            data,      elements_per_proc, MPI_INT,
            root_proc, MPI_COMM_WORLD);
// data[] now filled with unique portion from root_data[]
```
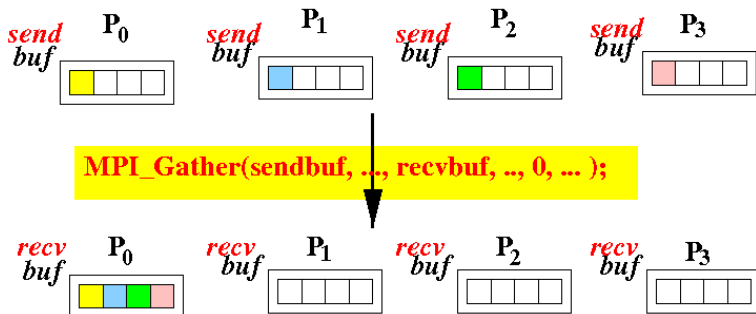
Exercise: Scatter a Matrix

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

Gather from All to One

Source: Shun Yan Cheung Notes on MPI

▶ 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

```c
// gather_demo.c
int total_elements = 16;
int elements_per_proc = total_elements / total_procs;

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

// Each proc fills data[] with "unique" values
int x = 1;
for(i=0; i<elements_per_proc; i++){
  data[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){
  root_data = malloc(sizeof(int) * total_elements);
 }

// Everyone calls gather, root proc receives, others send
MPI_Gather(data,      elements_per_proc, MPI_INT,
           root_data, elements_per_proc, MPI_INT,
           root_proc, MPI_COMM_WORLD);
// root_data[] now contains each procs data[] in order
```
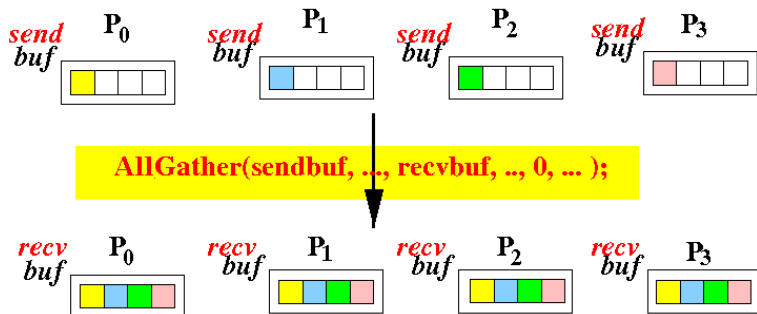
All Gather: Everyone to Everyone

Source: Shun Yan Cheung Notes on MPI

▶ 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

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

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

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

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

Reduction: All to One
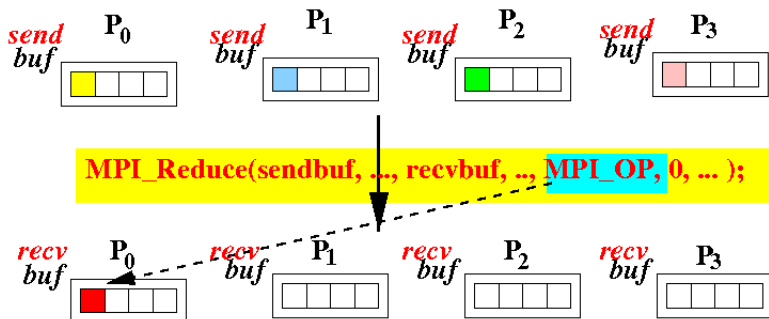
Source: Shun Yan Cheung Notes on MPI

- ▶ 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`
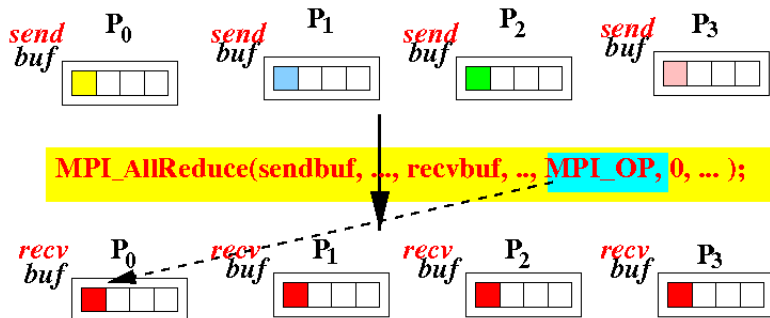- ▶ Root ends with reduced data in `receive` buffer

Reduce Example

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

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

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

Reduction for All: All-Reduce

Source: Shun Yan Cheung Notes on MPI

- ► Every processor has data in `send` buffer
- ► All processors need all data **reduced**
- ► All procs end with reduced data in a `receive` buffer

Allreduce Example

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

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

  // Everyone calls reduce, everyone sends and receives
  MPI_Allreduce(data, reduced_data, total_elements, MPI_INT,
                MPI_SUM, // operation to perform on each element
                MPI_COMM_WORLD);
  // reduced_data[] now contains each procs data[] summed up
}
```

In-place Reduction

▶ Occasionally want to do reductions in-place: send and receive buffers are the same.

▶ Useful for updating pagerank array in HW2

▶ 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 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 assumption for many problems (Pagerank)
- ▶ *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"
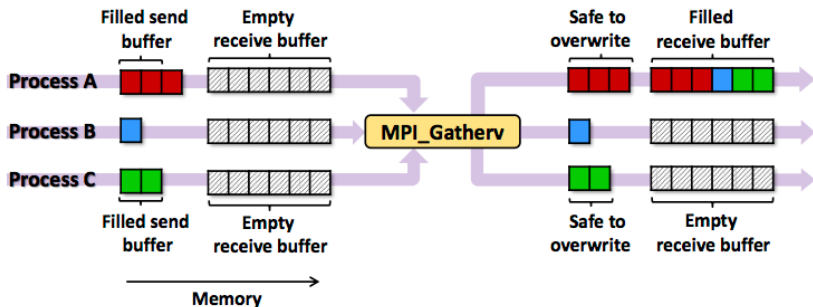
MPI_Scatterv Example
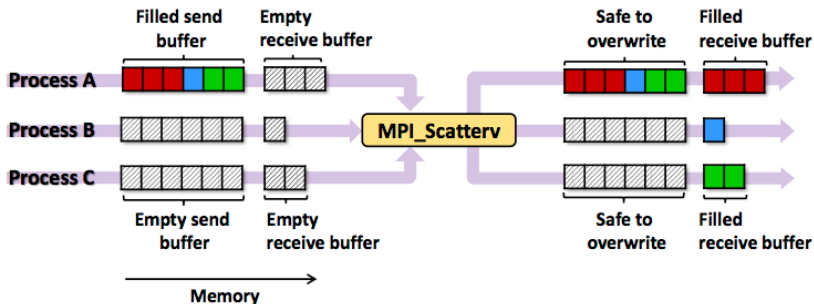
Source: SKIRT Docs

```
//                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;                    recv = (rank !=0) ? null :
int counts[] = { 3, 1, 2};          malloc(total * sizeof(int));
int displs[] = { 0, 3, 4};
int send[counts[rank]];           MPI_Gatherv(
int *recv, i;                      send, counts[rank],  MPI_INT,
for(i=0; i<counts[rank]; i++){     recv, counts, displs, MPI_INT,
  send[i] = rank*(i+1);            0, MPI_COMM_WORLD);
}
```

Dynamic Count and Displacements for Vector Comm Ops

▶ Common problem: # of procs does not evenly divide input size

▶ Use the vector versions of collective ops

▶ To calculate counts and displacements and spread work evenly, use a pattern like the below (see scatterv_demo.c)

```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

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

▶ Commercial Parallel Debuggers exist, TotalView is popular
▶ For small-ish programs debug printing + Valgrind + Effort
  will usually suffice

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