MPI and Communication Patterns

Chris Kauffman

CS 499: Spring 2016 GMU

Mini-exam 1 back

Stat	Val	
Count	34	
Average	35.44	88.6%
Median	36.50	91.3%
Standard Deviation	3.32	8.3%

Results overall good

Logistics

Today

- Mini-exams back
- HW 2 Overview
- Finish up discussion of `heat`
- Collective Communication

Reading: Grama Ch 6 + 4

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

Career Fair!

- 11:00 a.m.- 4:00 p.m.
- Dewberry Hall, Johnson Center
- Wed 2/17: Science/Tech
- Thu 2/18: Business/Non-tech

From Last Time

- 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

- `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
```

Exercise: MPI version of HW1's heat.c

- How should data in H divided among procs?
- Is communication required?
- How would one arrange MPI_Send / MPI_Recv calls?
- How much data needs to be transferred and between who?
- When the computation is finished, how can all data be displayed?

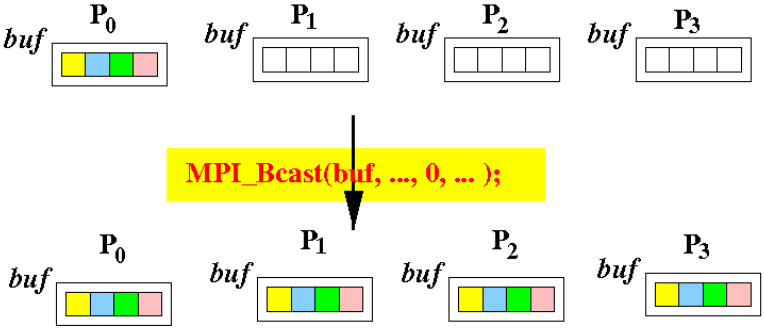
Where might the following be used?

```
int MPI_Init(int *argc, char ***argv) ;
int MPI_Finalize() ;
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status);
```

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



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

Broadcast Example Code

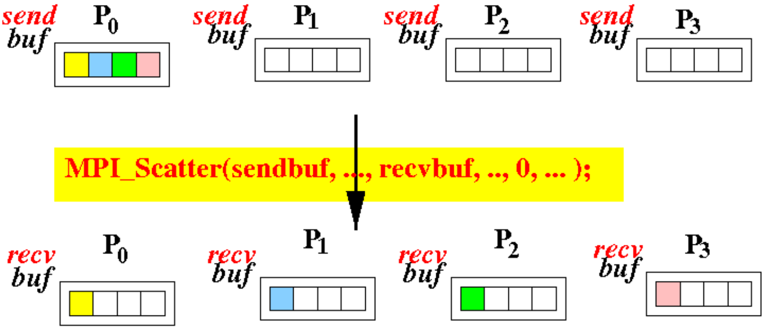
In `broadcast_demo.c`

```
// 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

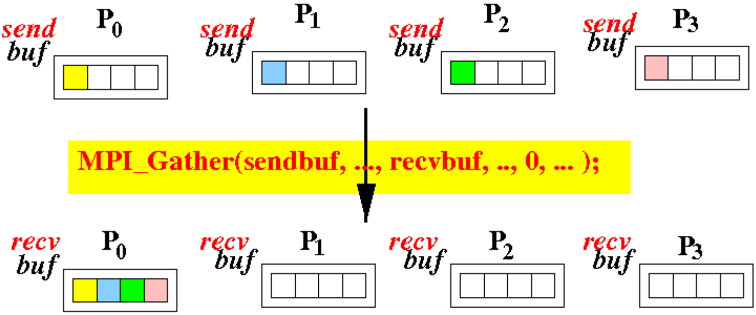
In scatter_demo.c

```
// 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[]
```

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

In `gather_demo.c`

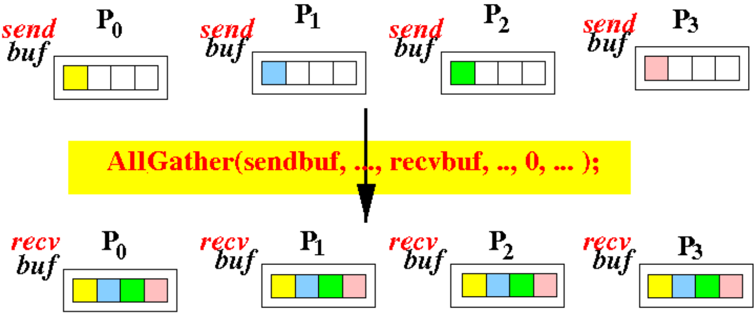
```
// Everyone allocates for their share of data including root
data = malloc(sizeof(int) * elements_per_proc);

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

In `allgather_demo.c`

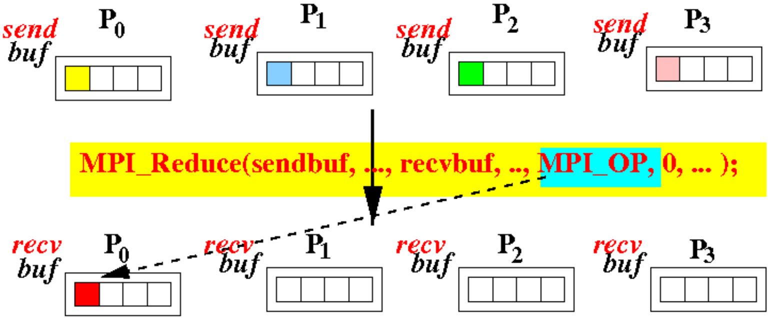
```
// Everyone allocates for their share of data including root
data = malloc(sizeof(int) * elements_per_proc);

/* 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

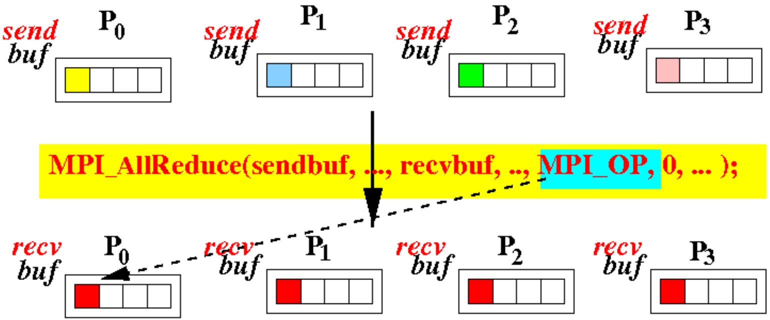
Reduce Example

```
{ // 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: All to All



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, data, total_elements, MPI_INT,
                MPI_SUM, // operation to perform on each element
                MPI_COMM_WORLD);
  // data[] now contains each procs data[], min elements
}
```

Summary of Communications

Operation	MPI Function	Synopsis	HW2?
		Individual	
Send	`MPI_Send`	One-to-one send	
Receive	`MPI_Recv`	One-to-one receive	
Send/Receive	`MPI_Sendrecv`	One-to-one send/receive	X
		Collective	
Barrier	`MPI_Barrier`	All wait for stragglers	-
Broadcast	`MPI_Bcast`	Root to all else, same data	X
Scatter	`MPI_Scatter`	Root to all else, different data	X
Gather	`MPI_Gather`	All to root, data ordered	X
Reduce	`MPI_Reduce`	All to root, data reduced	
All-Gather	`MPI_Allgather`	All to all, data ordered	X
All-Reduce	`MPI_Allreduce`	All to all, data reduced	X
		Not Discussed	
Prefix	`MPI_Prefix`	All-to-all, data ordered/reduced	
All-to-AllP	`MPI_Alltoall`	All-to-all, personal messages	

Exercise: Plan for Pagerank

```
PROCEDURE PAGERANK:

load N by N matrix LINKS from file

// Normalize LINKS matrix
allocate COL_SUMS array size N
fill COL_SUMS with sum of each column of LINKS
divide each entry A[r,c] by COLSUM[c]

// Setup rank arrays
allocate CUR_RANKS array size N
allocate OLD_RANKS array size N
initialize elements of OLD_RANKS to 1/N

// Main loop to iteratively compute pageranks
repeat
  CUR_RANKS = LINKS * OLD_RANKS // matrix mult
  verity sum of CUR_RANKS is 1  // error checking
  DIFF = sum(abs(CUR_RANKS - OLD_RANKS))
  if DIFF < tolerance
    exit loop
  copy CUR_RANKS to OLD_RANKS
end
```

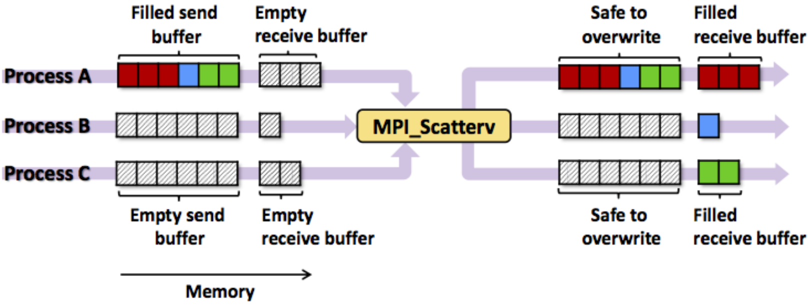
- Where are there opportunities for parallelization?
- Which collective communication operations will be required and where would you put them?
- Where will the answer be stored at the end of the day?

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* versions of each comm op exist which relax these assumptions
- 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`	

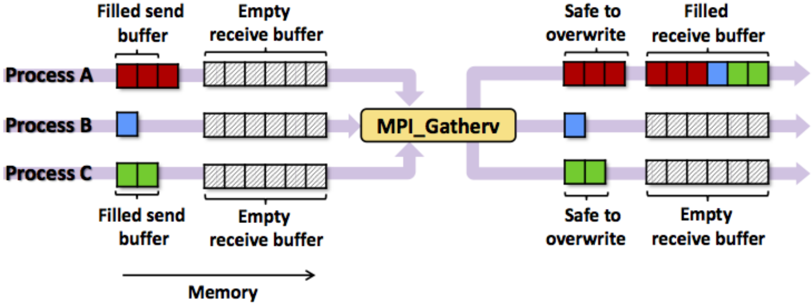
MPI_Scatterv Example



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 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`)

```
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
```