ECE 574 – Cluster Computing
Lecture 19

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

23 October 2026

Announcements

- HW#5 was graded
- Midterm grades?
- HW#7 will be posted
- Forgot from last time, in MPI_Recv can specify `MPI_ANY_SOURCE` when listening, so you don't have to force listeners to know what rank a message is coming from

HW#5 (pthreads) Review – Measurements

- When looking at speedup, use the `real` value from time, not the other ones
- Why was it not perfectly linear? Remember Amdahl's Law?

Speedup	2	4	8	16	32	64
real	1.8	3.2	4.5	5.1	5.0	5.4
convole	1.8	4.5	7.2	13.5	14.9	14.2
combine	2.0	5.5	9.3	18.0	18.3	17.0

MPI Brief Review

- We looked at setting up MPI
- Looked at point-to-point where we use Send and Recv to send one data block at a time
- You can imagine that's inefficient (both in coding but in runtime) if have to do this for all 1 million nodes in a large system

Could we manually Send Data to All Ranks?

- Rank 0 could send to each individual, take a while
- Some sort of tree, 0 to 1 and 2, 1 sends to 3 and 4, etc.
- Can we broadcast instead?

Collective Communication

- All must participate or there can be problems.
- Do not take tag arguments
- Can only operate on MPI defined data types, not custom

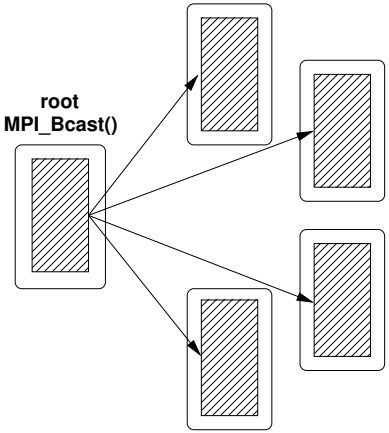
Collective Communication Operations

- Synchronization – all processes wait
- Data Movement – broadcast, scatter-gather
  - scatter = take one structure and split among processes
  - gather = take data from all processes and combine it
- Reduction – one process combines results of all others

MPI_Barrier()

- All processes wait at this point.
- `MPI_Barrier (comm)`

MPI_Bcast()

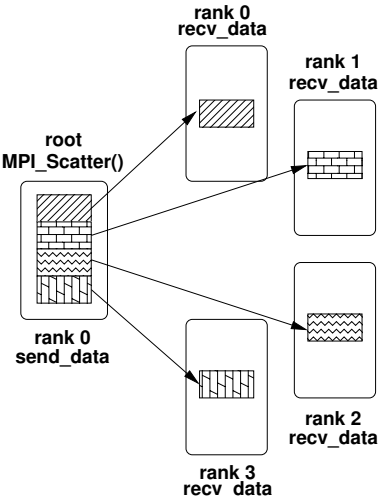


MPI_Bcast() – notes

```
MPI_Bcast (&buffer,count,datatype,root,comm);
```

- Sends data from the *root* rank to each other rank.
- Is blocking; when encountering a Bcast all nodes wait until they have received the data.
- There is no need to receive; the root sends the data and all other ranks will receive, just with the one command
- After command executes, all ranks will hsave same data in buffer

MPI_Scatter()

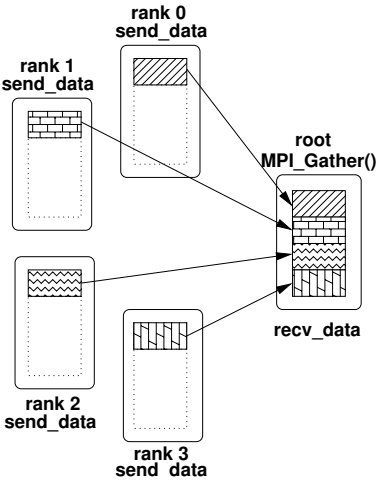

rank 0
recv_data
rank 1
recv_data
root
MPI_Scatter()
rank 0
send_data
rank 2
recv_data
rank 3
recv_data

MPI_Scatter() – notes

```
MPI_Scatter (&send_data,sendcnt,sendtype,
    &recv_data,recvcnt,recvtype,root,comm);
```

- Copies `sendcnt` sized chunks of sendbuf to each rank's `recvbuf`
- root also gets a share of data (just a local copy)
- Can use MPI_IN_PLACE as the recv_data to avoid needing separate input and output arrays
- Note: copies to beginning of buffer

MPI_Gather()


rank 0
send_data
rank 1
send_data
root
MPI_Gather()
recv_data
rank 2
send_data
rank 3
send_data

MPI_Gather() – notes

```
MPI_Gather (&send_data,sendcnt,sendtype,
    &recv_data,recvcount,recvtype,root,comm);
```

- Copies `recvcount` sized chunks of sendbuf from each rank to `recvbuf` in root, offset by recvcount for full result
- **NOTE** values start at beginning of each rank's sendbuf
- Can use MPI_IN_PLACE as the send_data to avoid needing separate input and output arrays (complex though, see example)

Scatter/Gather Boundary issues

- *NOTE* If the size of the data you are sending is not an even multiple of the number of ranks you'll have to manually handle the extra
- How?
  - Have the root manually handle the extra at end?
  - Pad your data to be a multiple of number of ranks and ignore the extra?
  - `MPI_Scatterv()` and `MPI_Gatherv()` routines let you send vectors (chunks of varying length) but complex to use

MPI_Scatterv()

```
int MPI_Scatterv (&send_data,sendcounts[],
        displs[],sendtype,&recv_data,recvcount,
        recvtype,root,comm);
```

- Vector scatter
- Send non-contiguous chunks
- In addition to regular scatter parameters, a list of start offsets and lengths.

MPI_Gatherv()

```
int MPI_Gatherv(&send_data,sendcount,sendtype,
    &recv_data, recvcounts[], displs[],
    recvtype, root, comm);
```

- Vector gather
- Can gather non-contiguous chunks
- In addition to regular scatter parameters, a list of start offsets and lengths.

MPI_Reduce()

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

- Operations
  - MPI_MAX,MPI_MIN – max, min
  - MPI_SUM – sum
  - MPI_PROD – product
  - MPI_LAND, MPI_BAND – logical/bitwise and
  - MPI_LOR,MPI_BOR – logical/bitwise OR
  - MPI_LXOR,MPI_BXOR – logical/bitwise XOR
  - MPI_MAXLOC,MPI_MINLOC – value and location
  - Can also create custom

MPI_Allgather()

- Gathers, to all
- Equivalent of gathering back to root, then rebroadcasting to all

MPI_Allreduce()

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

- Like an MPI_Reduce followed by an MPI_Bcast
- Once the reduction is done, broadcasts the results to all processes

MPI_Reduce_scatter()

- Does a reduction, then scatters the results

MPI_Alltoall()

- Scatter data from all to all

MPI_Scan()

- Lets you do partial reductions.

Custom Data Types

- You can create custom data types that aren't the MPI default, sort of like structures.
- Open question: can you just cast your data into integers and uncast on the other side? This is not recommended and might have issues on a heterogeneous cluster

Groups vs Communicators

- Can create custom groups if you don't want to broadcast to all.
- Use groups to create Communicators, then can use instead of WORLD

Virtual Topologies

- Your workload might map to a geometric shape (grid or graph)
- In a mesh type problem you might only want to talk to the 4 surrounding ranks and none of the others, so might be handy if can be placed in hardware to take advantage of that
- Doesn't have to match underlying hardware

MPI Examples

See the provided tar file with example code.

Running MPI code

- `mpiexec -np 4 ./mpi_test`
  Runs on 4 ranks
  note the space between np and 4 is important and things won't work if you leave it out
- You'll often see `mpirun` instead. Some implementations have that, but it's not the official standard way.

Running MPI code with slurm

- `sbatch -n X time_coarse.sh`
  Runs on X ranks
  Even on multi-node cluster might run some on same machine if it has multiple cores.

Send Example

- mpi_send.c
- Run with `mpiexec -np 4 ./mpi_send`
- Send 1 million integers (each with value of 1) to each node
- Each adds up 1/4th then sends only the sum (a single int) back
- Notice this is a lot like pthreads where we have to do a lot of work manually.

Send Example – Things to note

- `MPI_Init()` at start
  passes command line args, on most implementations this will essentially broadcast the command line args across all ranks so
- `MPI_Comm_size()` to get number of ranks
- `MPI_Comm_rank()` to get our rank
- `MPI_Send()` in this case only from rank 0
- `MPI_Recv()` can use status value to get size, source, and tag

Blocking vs NonBlock Example

- `mpi_nonblock.c`
- Uses `Isend()` which doesn't block
- Shows code using `MPI_Test()` to see if done and `MPI_Wait()` to wait until completion

Wtime (Wallclock Time) Example

- `mpi_wtime.c`
- Same as previous example. but with timing
- Unlike PAPI, the time is returned as a floating point value

Barrier Example

- `mpi_barrier.c`
- Each machine sleeps some time based on rank
- All wait at barrier until last one arrives
- Note: seeing all printfs because in this case all ranks on same machine. This might not happen when running on a real cluster

Bcast Example

- `mpi_bcast.c`
- Same buffer on each machine
- At the broadcast function, one sends its version of the buffer and the rest wait until they receive the value.
- In the end they all have the same value

Scatter Example

- `mpi_scatter.c`
- Instead of sending all of A, breaks it into chunks and sends it to B in each rank.
- Note that while the program runs ordered as expected, the printfs might not reflect this
- Why would sendcount/recvcount ever be different? (is it a waste having two parameters)? Possibly so you can have equivalent data types (1000 x 1 byte vs 1x1000 byte) as arguments

Gather Example

- `mpi_gather.c`
- Each rank has its own copy of A which it sets to entirely its rank number
- Then a gather happens on rank0, of one int each. So what should B have in it? (0, 1, 2, 3, ...)
- What happens if prime number of ranks like 7. Boundary issue.

Gather Offset Example

- `mpi_gather_offset.c`
- Way to gather *not* from start of array
- Have to do some pointer mater

Gatherv Example

- `mpi_gatherv.c`
- Need to allocate counts and offsets arrays and fill in.
- Can special case to handle uneven ending.

Gatherv MPI_IN_PLACE

- `mpi_gatherv_in_place.c`
- Turns out you have to special case rank 0 and use MPI_IN_PLACE, for other ranks just set receive buffer to NULL

Reduce Example

- `mpi_reduce.c`
- Instead of waiting in a loop for tasks finishing and then adding up the results one by one, use a reduction instead.
- Many MPI routines are convenience things that could be done by a sequence of separate commands.

Hints for Homework #7

- Note, this is just brief to get you started
- We'll cover this much more in depth next lecture

Homework #7 Broadcasting

- Only load jpeg in Rank 0
- Send xsize/ysize/depth to all ranks so they can allocate space
  How? Send array of 3 ints with the info
- Once rank receives the xsize/ysize/dept calloc() the memory for the pixel data on non rank0
- Next broadcast the pixel data to all
- Do generic convolve based on limits like with pthread code

Homework #7 Gathering

- Gather results back, NOTE: gather from an array expects the results to start at beginning, not middle. If your result image has only 1/N of the image with the rest black it's usually because you don't offset properly in the gather
- In Rank 0 only combine
- In Rank 0 only write out jpeg data
- On haswell-ep we'll use sbatch
  `sbatch -n X ./shell.sh`
  The provided shell script has mpirun inside, sbatch is smart enough to update things so the right number of ranks are started based on the -n parameter