MPI Status object

- We have been putting an MPI_Status object at the end of our Recv calls
- This object contains three elements
  - `MPI_SOURCE` — the sender of the message
  - `MPI_TAG` — the tag the sender used
  - `MPI_ERROR` — any error code from the Recv
- We can also determine the number of items received using a function call
- MPI_Get_count(*stat, type, *count)

MPI Status object

- We use the Status object when our receive uses wildcards such as MPI_ANY_TAG or MPI_ANY_SOURCE
- We may still want to know who sent us some data in order to communicate back to them
- In some communications the amount of data being received may not be know a priori
- Will need to know how much of our array contains valid data

Collective Operations

- Collective communications occur on all processes in the communicator
- Since we are just using MPI_COMM_WORLD that means all proccesses in the calculation
- Three main classes
  - Global synchronization
  - Global communications
  - Global reductions

Collective Operations

- Collectives insist that quantity of data being sent matches that being received
- Collectives are blocking - if one MPI task doesn't call the function then everyone else waits
- There are no tags so all calls to collective operations are matched in strict calling order

Collective Syncronization

- We have already met this
- MPI_Barrier(comm);
- All tasks in the communicator wait until everyone has called the barrier function

Collective Communication

- Three main forms
  - Root process sends data to everyone — broadcast and scatter
  - Root receives data from everyone — gather
  - Each process communicates with everyone else — allgather and alltoall

MPI_Bcast

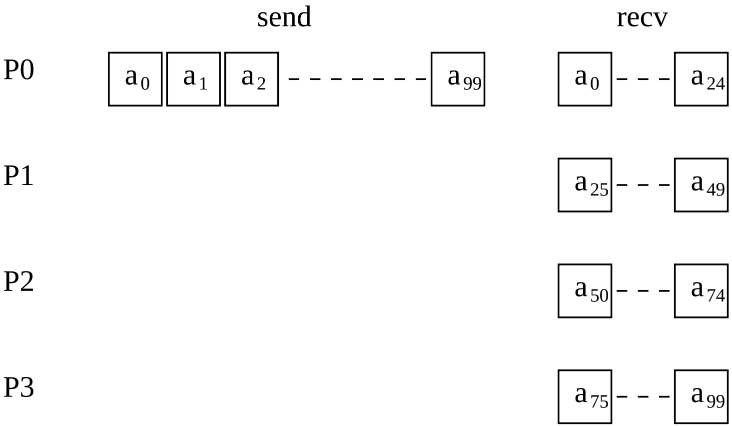
- MPI_Bcast(start, count, datatype, root, communicator);
- The process with rank *root* sends the data to all other processes in the communicator
- These processes store the data at start
- After this operation each node has a full copy of all the data

MPI_Scatter

- MPI_Scatter(send, sendcount, sendtype, recv, recvcount, recvtype, root, communicator);
- The root process sends different sections of the send buffer to each process
- All processes (including the root) put this data in the recv buffer
- sendcount and recvcount are usually identical as are sendtype and recvtype
- MPI insists that $\mathit{sendcount} \times \mathit{sizeof}(\mathit{sendtype}) = \mathit{recvcount} \times \mathit{sizeof}(\mathit{recvtype})$

MPI_Scatter

- Example — on root process we have *int send[100]*, on each proc *int recv[25]* and our simulation has 4 proc
- MPI_Scatter(send, 25, MPI_INT, recv, 25, MPI_INT, 0, MPI_COMM_WORLD);

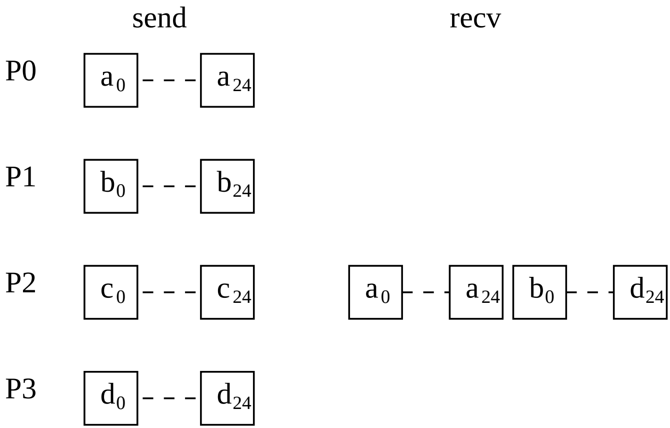


MPI_Gather

- MPI_Gather does the opposite of MPI_Scatter
- MPI_Gather(send, sendcount, sendtype, recv, recvcount, recvtype, root, communicator);
- The root process receives copies of the send buffer from each process
- The data is assembled in rank order in the recv buffer

MPI_Gather

- MPI_Gather(send, 25, MPI_INT, recv, 25, MPI_INT, 2, MPI_COMM_WORLD);



MPI_Reduce

- MPI_Reduce(send, recv, count, datatype, operation, root, communicator);
- The data in the send buffer of each process is reduced by the operation
- These operations can be addition, multiplication, maximum, minimum etc.
- The result is stored in the recv buffer on the root proccess

All collectives

- MPI also provides versions of Reduce and Gather where the results happen on all processes
- MPI_Allreduce(send, recv, count, type, operation, communicator);
- MPI_Allgather(send, sendcount, sendtype, recv, recvcount, recvtype, communicator);
- Note there are no root processes in these calls
- In the *non all* versions the recv buffers on the non root processes can be NULL
- In the *all* versions each process must have sufficient space in the recv buffer

MPI_Alltoall

- Finally, there is a function that allows all processes to send different messages to every process at the same time
- MPI_Alltoall(send, sendcount, sendtype, recv, recvcount, recvtype, communicator);
- Each process sends sendcount items, starting from $rank \times sendcount$, from its send buffer
- Each process receives sendcount items from each other process and stores it in the recv buffer starting from $rank \times sendcount$
- Errors occur if the send and/or recv buffers are not large enough