Collective Communication

- Collective communications occur on all processes in the communicator
- Since we are just using MPI_COMM_WORLD that means all proccesses in the calculation
- Note - all processes must call the function otherwise other processes will become blocked

Collective Communication

- MPI_Bcast(. . .) sends information to all processes
- MPI_Reduce(. . .) combines data from all processes and returns it to one process
- MPI_Scatter(. . .) splits up a large data set across all processes
- MPI_Gather(. . .) does the opposite of MPI_Scatter
- In some senses MPI_Barrier() is also a collective function even though no data is passed around

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_Reduce

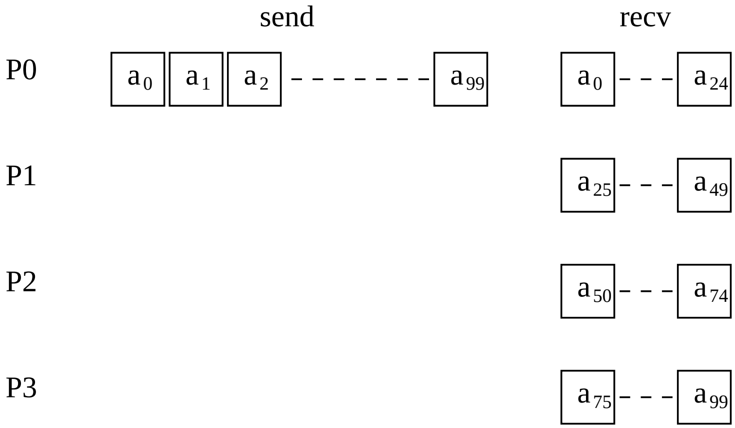
- MPI_Reduce(send, recv, count, datatype, operation, root, communicator);
- The data in the send buffer of each process is reduced by the operation
- These operations can be addition, multiplication, maximum, minimum etc.
- The result is stored in the recv buffer on the root proccess

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 $sendcount \times sizeof(sendtype) = recvcount \times sizeof(recvtype)$

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);

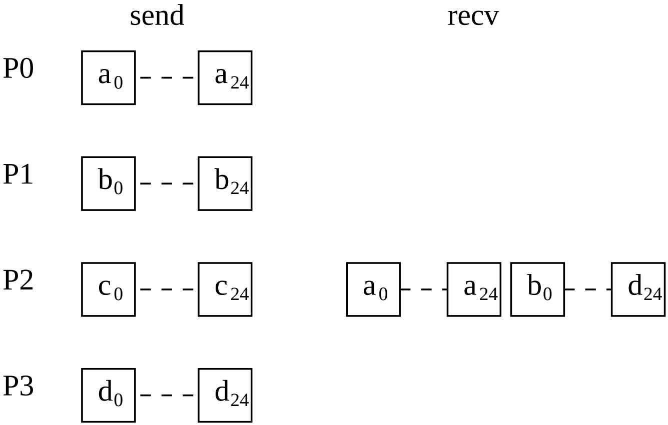


MPI_Gather

- 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);



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

MPI_Sendrecv

- When passing data between processes in the Jacobi example we had to be careful to get the MPI_Send and MPI_Recv calls in the right order
Proc 0	Proc 1
MPI_Send(...) MPI_Recv(...)	MPI_Recv(...) MPI_Send(...)
- This type of communication pattern is very common
- MPI provides the MPI_Sendrecv() function to get around the problem of deadlocking calls
- MPI_Sendrecv(send, sendcount, sendtype, dest, sendtag, recv, recvcount, recvtype, source, recvtag, communicator, stat);
- Basically mash the send and receive functions into one call