Where are we in COP 4338

C fundamentals

Unix Fundamentals

Memory

Files

Processes

Threads

Interprocess communication

Parallel Computing

Networking

Optional Advanced Topics

Parallel Computing

MPI Walkthrough

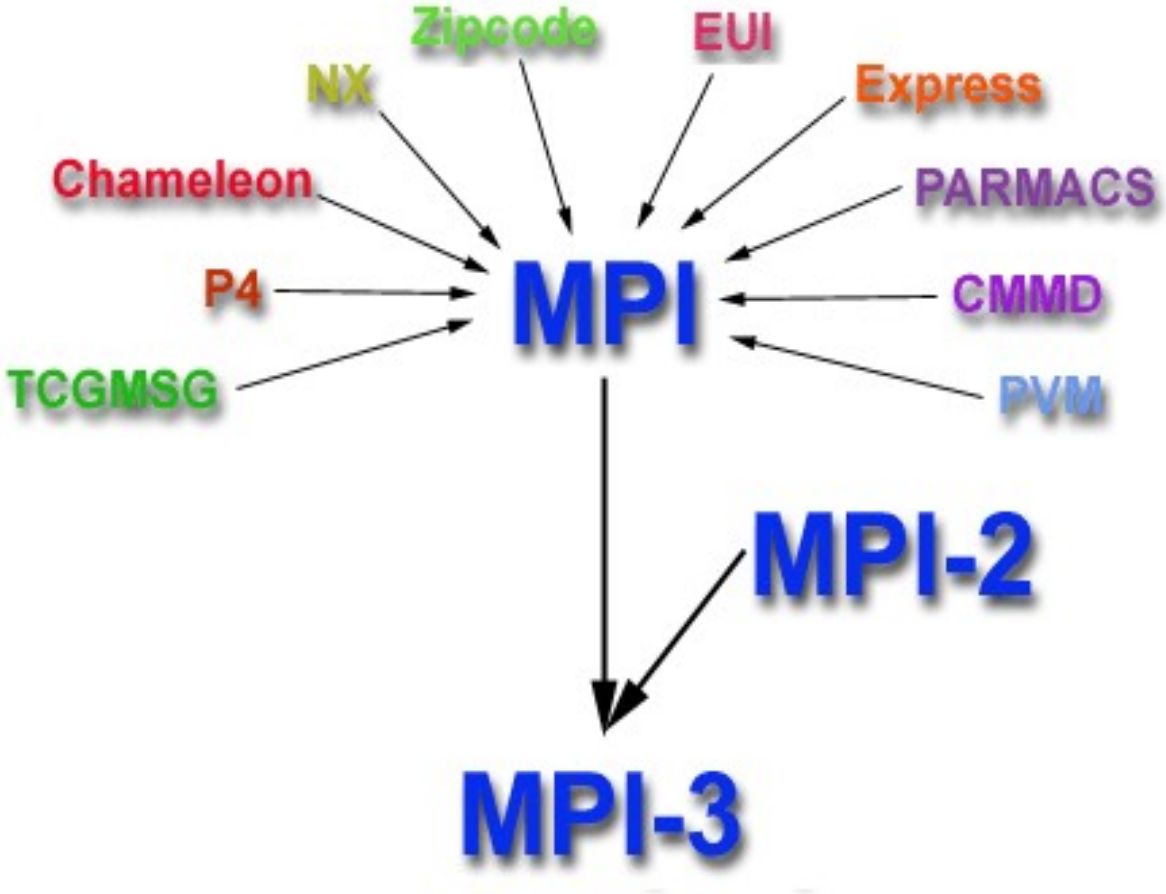
A Walk Through the Maze

- A bit of a history.
- Basics.
- Environment management routines.
- Point-to-point communication routings.
- Collective communication routings.
- Derived data types.

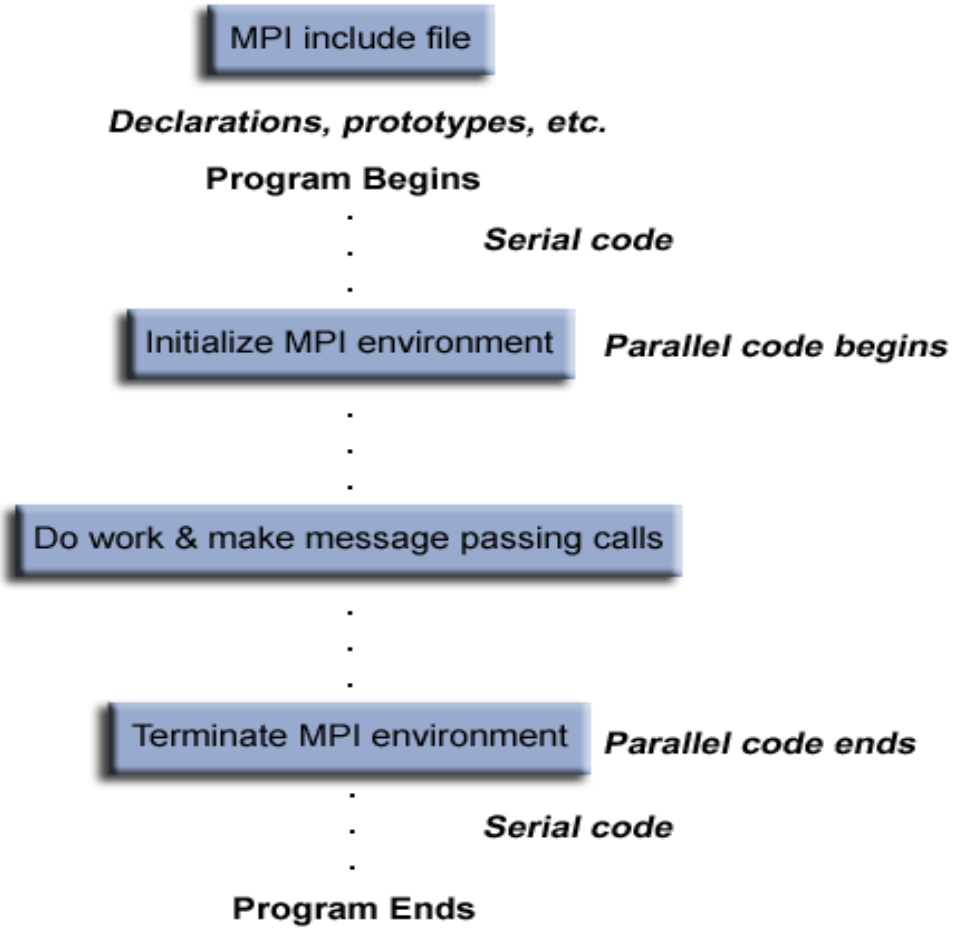
History of MPI

- Late 1980s: vendors had unique libraries.
- 1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab.
- 1992: Work on MPI standard begun.
- 1994: Version 1.0 of MPI standard.
- 1997: Version 2.0 of MPI standard.
- Today: MPI is dominant message passing library standard.

History of MPI



History of MPI



Basics

- Header file: #include <mpi.h>
- Format of MPI calls:

  ret = MPI_Function(parameters...);

  Returns MPI_Success if successful.

General Programming Structure

```
#include <mpi.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    /* Do work and call MPI routines */
    MPI_Finalize();
    return 0;
}
```

Environment Management Routines

- MPI_Init(&argc, &argv);
- MPI_Comm_size(comm, &size);
- MPI_Comm_rank(comm, &rank);
- MPI_Wtime();
- MPI_Wtick();
- MPI_Finalize();

Point-to-Point Communications

- Message-passing between two and only two different MPI processes: one sending and one receiving.
- Different flavors of send and receive:
  - Synchronous vs. asynchronous.
  - Blocking vs. non-blocking.
  - Buffered vs. unbuffered.

MPI

Point to Point Communication

- A message is sent from a sender to a receiver
- There are several variations on how the sending of a message can interact with the program

Synchronous

- A synchronous communication does not complete until the message has been received
  - A FAX or registered mail

Asynchronous

- An asynchronous communication completes as soon as the message is on the way.
  - A post card or email

Point-to-Point Communication Routine Arguments

sendrecv.c

Point-to-Point Communication Routine Arguments

- **Buffer**: program address space that references the data to be sent or received.
- **Count**: number of data elements.
- **Type**: either predefined or user created data types.
- **Source**: the rank of the originating process of the message (wildcard `MPI_ANY_SOURCE`).
- **Destination**: the rank of the processes where the message should be delivered.

More Arguments

- **Tag**: a non-negative integer to indicate the type of a message (wildcard: MPI_ANY_TAG).
- **Communicator**: the set of processes for which the source and destination fields are valid.
- **Status**: a pointer to the MPI_Status structure, used by a receive operation to indicate the source and the tag of the received message, as well as the actual bytes received.
- **Request**: used as a handle to later query the state of a non-blocking operation.

MPI Predefined Data Types

- **MPI_CHAR**
- **MPI_BYTE**
- **MPI_DOUBLE**
- **MPI_FLOAT**
- **MPI_INT**
- **MPI_LONG**
- **MPI_LONG_DOUBLE**
- **MPI_SHORT**
- **MPI_UNSIGNED_CHAR**
- **MPI_UNSIGNED**
- **MPI_UNSIGNED_LONG**
- **MPI_UNSIGNED_SHORT**
- **MPI_PACKED**

Point-to-Point Communications

- MPI_Send(&buf, count, type, dest, tag, comm);
  - Blocking send: buffer can be reused after function returns.
  - Implemented either by copying the message into system buffer, or copying the message into the matching receive buffer.
- MPI_Recv(&buf, count, type, src, tag, comm, &status);
  - Blocking receive.
- MPI_Ssend(&buf, count, type, dest, tag, comm);
  - Synchronous send: returns only when a matching receive has been posted.
  - Send buffer can be reused after function returns.

Point-to-Point Communications

mpi-block.c

Point-to-Point Communications

- MPI_Rsend(&buf, count, type, dest, tag, comm);
  - Ready send: send may only be started if the matching receive is already posted. Error otherwise.
  - Buffer can be reused after function returns.
- MPI_Bsend(&buf, count, type, dest, tag, comm);
- MPI_Buffer_attach(&buf, size);
- MPI_Buffer_detach(&buf, &size);
  - Buffered send: the outgoing message may be copied to the user-specified buffer space, so that the sending process can continue execution.
  - User can attach or detach memory used to buffer messages sent in the buffered model.

Point-to-Point Communications

- MPI_Sendrecv(&sendbuf, sendcnt, sendtype, dest, sendtag, &recvbuf, recvcnt, recvtype, src, recvtag, comm, &status);
  - Combines the sending of a message and receiving of a message in a single function call.
  - Can be more efficient.
  - Guarantee deadlock will not occur.
- MPI_Probe(src, tag, comm, &status);
  - Checks for an incoming message without actually receiving it.
  - Particularly useful if you want to allocate a receive buffer based on the size of an incoming message.

Point-to-Point Communications

- MPI_Isend(&buf, count, type, dest, tag, comm, &request);
  - Non-blocking (or immediate) send.
  - It only posts a request for send and returns immediately.
  - Do not access the send buffer until completing the receive call with **MPI_Wait**.
- MPI_Irecv(&buf, count, type, src, tag, comm, &request);
  - Non-blocking (or immediate) receive.
  - It only posts a request for receive and returns immediately.
  - Do not access the receive buffer until completing the receive call with **MPI_Wait**.

Point-to-Point Communications

- MPI_Test(&request, &flag, &status);
  - Determines if the operation associated with a communication request has been completed.
  - If flag=**true**, you can access **status** to find out the message information (source, tag, and error code)**.**
- MPI_Wait(&request, &status);
  - Waits until the non-blocking operation has completed.
  - You can access **status** to find out the message information.

Point-to-Point Communications

- MPI_Issend(&buf, count, type, dest, tag, comm, &request);
  - Non-blocking synchronous send.
- MPI_Ibsend(&buf, count, type, dest, tag, comm, &request);
  - Non-blocking buffered send.
- MPI_Irsend(&buf, count, type, dest, tag, comm, &request);
  - Non-blocking ready send.
- MPI_Iprobe(src, tag, comm, &flag, &status);
  - Non-blocking function that checks for incoming message without actually receiving the message.

MPI

- Blocking and Non-blocking

  Blocking operations only return when the operation has been completed
  - Printer

- Non-blocking operations return right away and allow the program to do other work
  - TV Capture Cards (Can record one channel and still be watching another one)

- Collective Communications

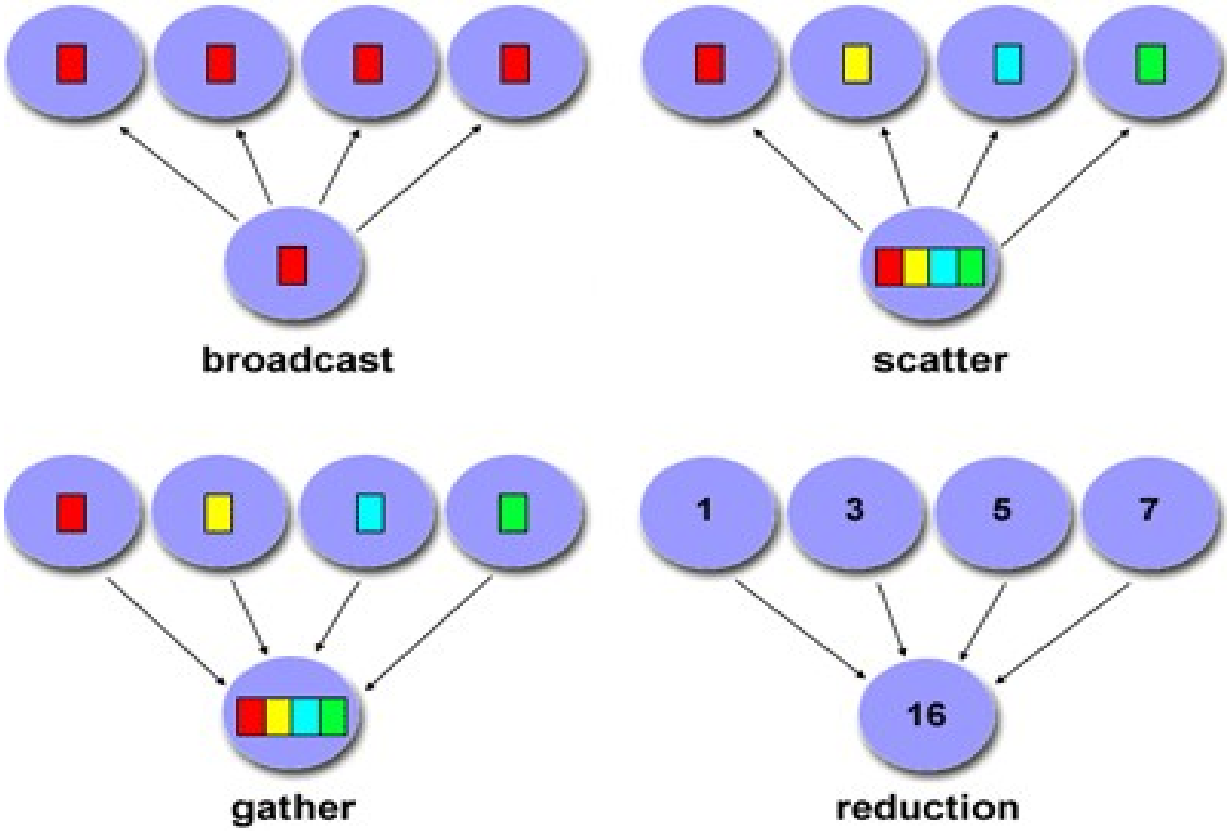
  Point-to-point communications involve pairs of processes.

  Many message passing systems provide operations which allow larger numbers of processes to participate

Collective Communications

- All processes in a communicator must participate by calling the collective communication routine.
- Purpose of collective operations:
  - Synchronization: e.g., barrier.
  - Data movement: e.g., broadcast, gather, scatter.
  - Collective computations: e.g., reduction.
- Things to remember:
  - Collective operations are blocking.
  - Can only be used with MPI predefined types (no derived data types).
  - Cannot operate on a subset of processes; it’s all or nothing.

Collective Communications



MPI

Types of Collective Transfers

- Barrier
  - Synchronizes processors
  - No data is exchanged but the barrier blocks until all processes have called the barrier routine
- Broadcast (sometimes multicast)
  - A broadcast is a one-to-many communication
  - One processor sends one message to several destinations
- Reduction
  - Often useful in a many-to-one communication

Collective Communications

- MPI_Barrier(comm);
  - Performs a barrier synchronization.
- MPI_Bcast(&buf, count, type, root, comm);
  - Allows one process to broadcast a message to all other processes.
- MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm);
  - A group of elements held by the root process is divided into equal-sized chunks, and one chunk is sent to every process.
- MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm);
  - The root process gathers data from every process.

Collective Communications

mpi_scatter.c

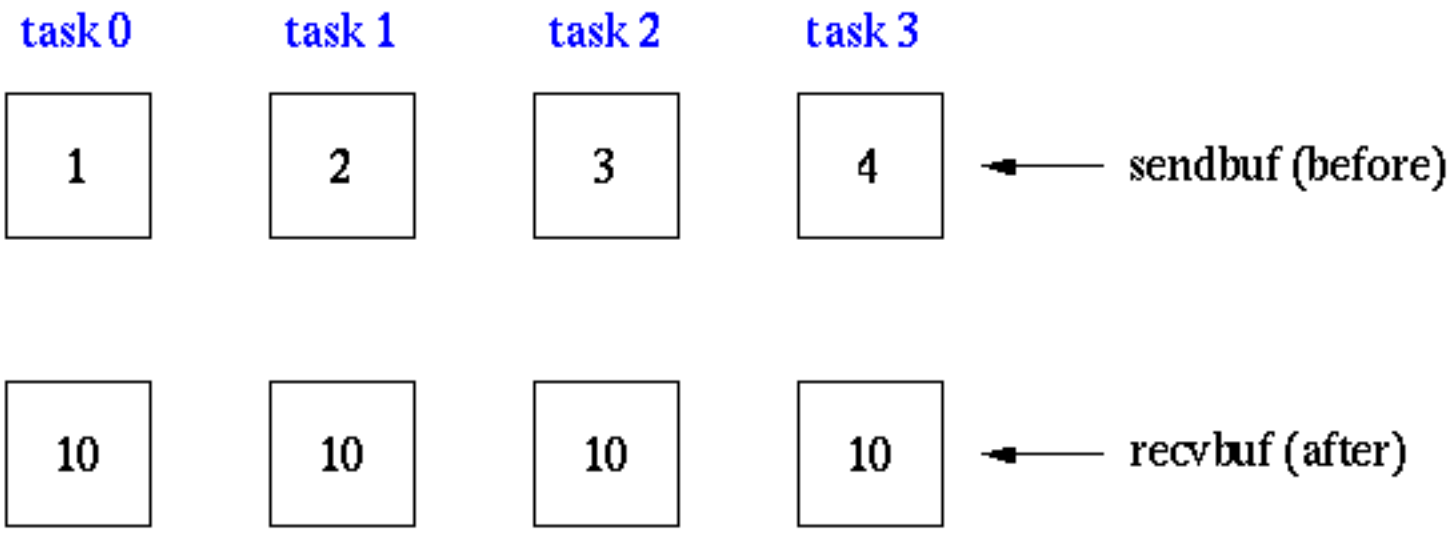
Collective Communications

- MPI_Allgather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm);
  - All processes gather data from every processes.
- MPI_Alltoall(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, comm);
  - Performs an all-to-all communication among all processes.
- MPI_Reduce(&sendbuf, &recvbuf, count, type, op, root, comm);
  - Reduction operation.
- MPI_Allreduce(&sendbuf, &recvbuf, count, type, op, comm);
- MPI_Reduce_scatter(&sendbuf, &recvbuf, recvcnt, type, op, comm);
- MPI_Scan(&sendbuf, &recvbuf, count, type, op, comm);

MPI_Allreduce

Perform and associate reduction operation across all tasks in the group and place the result in all tasks

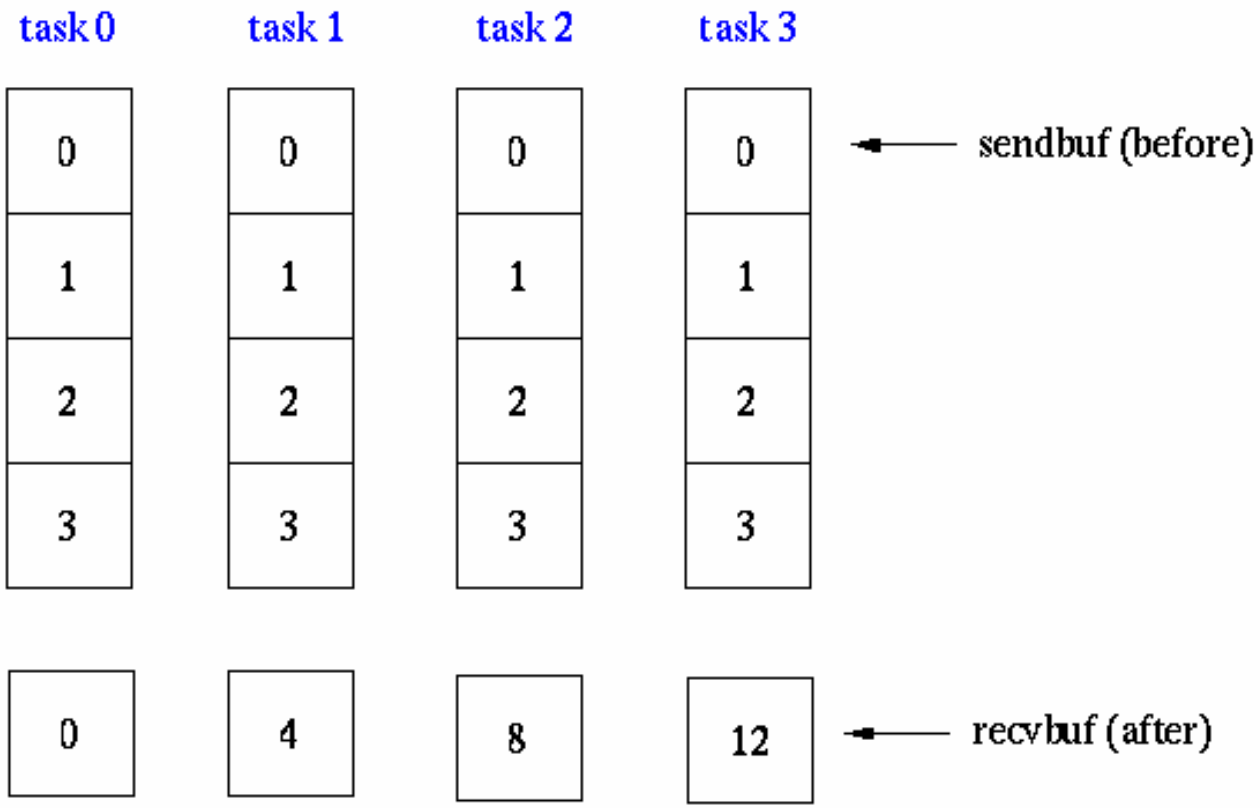
```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
              MPI_COMM_WORLD);
```



MPI_Reduce_scatter

Perform reduction operation on vector elements across all tasks in the group, then distribute segments of result vector to tasks

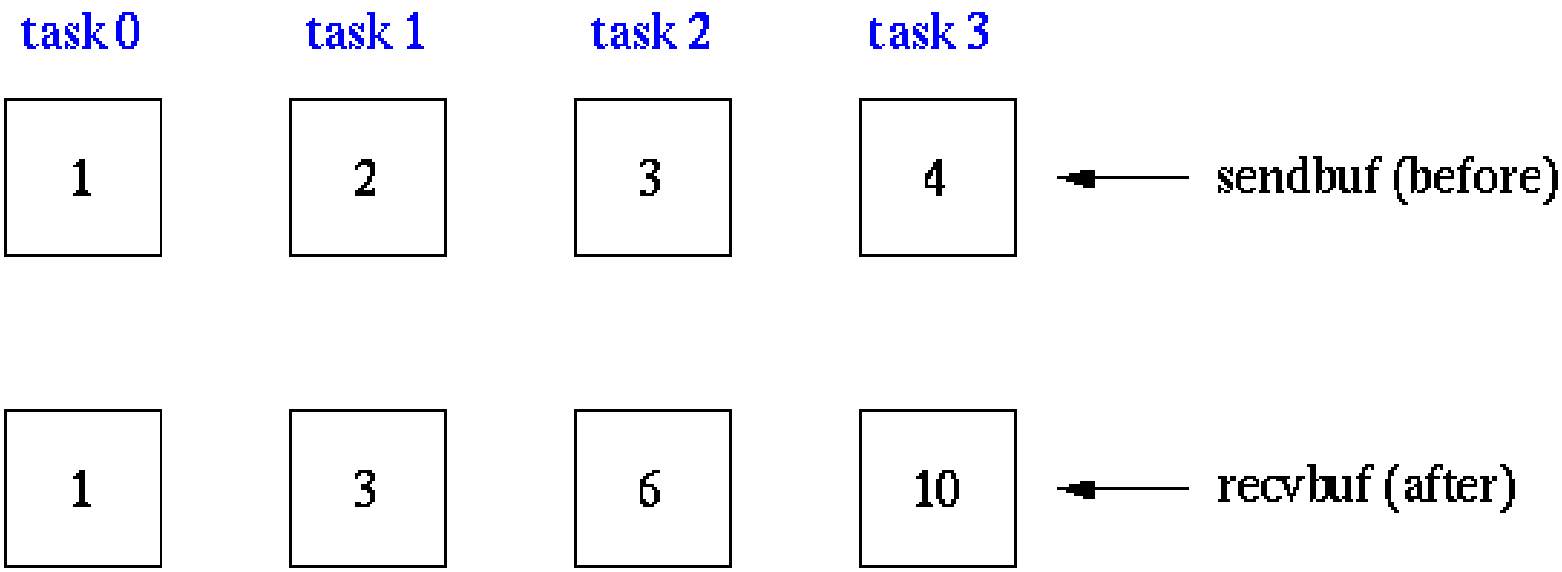
```
recvcount = 1;
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount, MPI_INT, MPI_SUM,
                   MPI_COMM_WORLD);
```



MPI_Scan

Computes the scan (partial reductions) of data on a collection of processes

```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
                  MPI_COMM_WORLD);
```



Derived Data Types

- MPI allows you to define your own data structures.
- Why do I need this?
- You can construct several data types:
  - **Contiguous**: an array of the same data types.
  - **Vector**: similar to contiguous, but allows for regular gaps (stride) in the displacements.
  - **Indexed**: an array of displacements of the input data types.
  - **Struct**: a general data structure.

Derived Data Types

- MPI_Type_contiguous(count, oldtype, &newtype);
  - Create a new data type consisting of **count** copies of the old type.
- MPI_Type_vector(count, blocklength, stride, oldtype, &newtype);
  - Create a new data type that consists of **count** blocks of length **blocklength**. The distance between blocks is called stride.
- MPI_Type_indexed(count, blocklengths[], displacements[], oldtype, &newtype);
  - Create a new data type of blocks with arbitrary displacements.

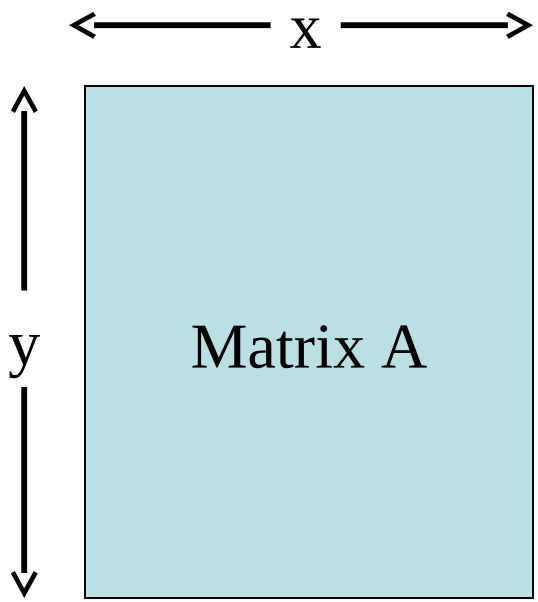
Derived Data Types

contiguous.c
mpivector.c

Derived Data Types

- MPI_Type_struct(count, blocklength[], displacements[], oldtypes[], &newtype);
  - Create a generic data structure.
- MPI_Type_extent(type, &extent);
  - Returns the extent of a data type -- the number of bytes a single instance of data type would occupy in a message (depending on hardware data alignment requirement).
- MPI_Type_commit(&newtype);
  - New data types must be committed before use.
- MPI_Type_free(&type);
  - Deallocate a data type object.

Contiguous Type




```
MPI_Type_contiguous(x, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
```