

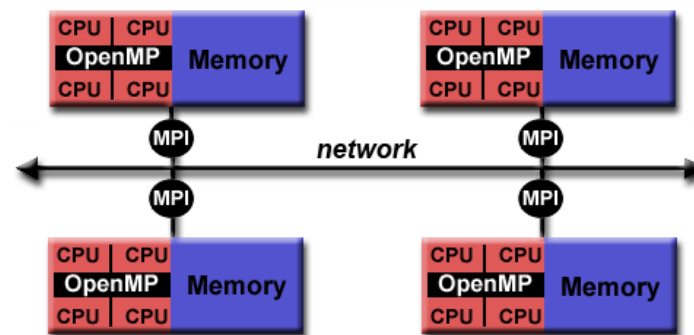
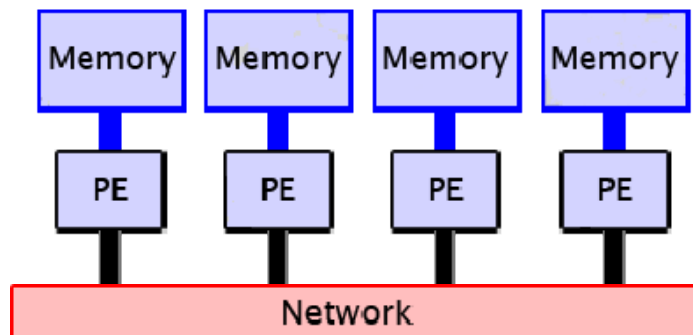


# Distributed memory programming using MPI

CS121 Parallel Computing  
Spring 2021

# Distributed memory programming

- Processes have local memory, and communicate by sending messages over network.
- Use standard C / C++ / Fortran. Call library for communication functions.
- Message Passing Interface (MPI)
  - Library developed by academics and industry in early 1990's.
    - Standardized clutter of existing HPC languages and tools.
  - Goals are portability, efficiency and flexibility.
  - Three versions
    - MPI 1.1-1.3, Jun 1995 – May 2008
    - MPI 2.1-2.2, Sep 2008 – Sep 2009
    - MPI 3.1, Jun 2015
  - Defines an interface of message passing routines, but not implementation.
  - Vendors create own implementations optimized for their hardware.
  - MPICH (Argonne National Laboratory) and OpenMPI are most widely used implementations.
  - Also commercial implementations from Intel, Microsoft, etc.



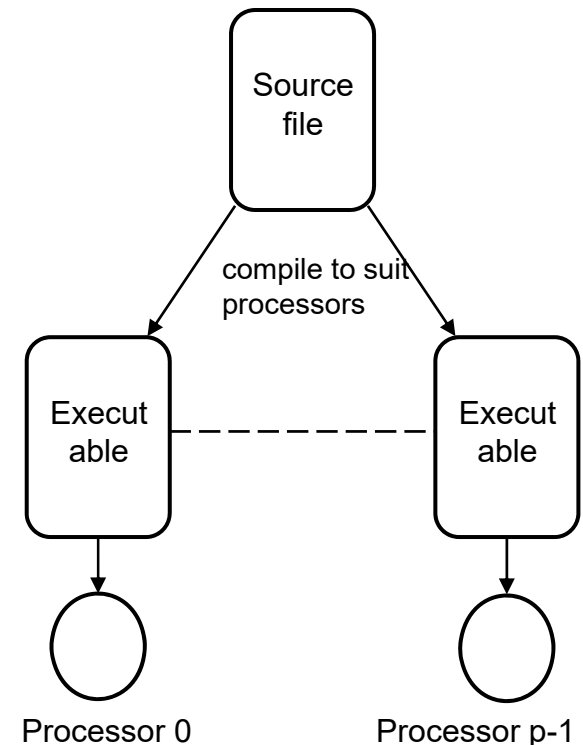


# Structure of MPI

- MPI contains four main types of functions.
- Process creation
  - Divide the program into separate processes that can be executed on different processors.
- Sending and receiving messages
  - Point to point vs collective communication.
  - In contrast to shared memory programming where processes interact by reading and writing shared data in memory.
  - Newer versions of MPI also support shared memory style (one way) remote memory access.
- Defining new derived data types.
- Defining and structuring communication groups.
  - Communicators organize processes performing common task.
  - Define topologies for efficient communication.

# Process model

- MPI based on SPMD (single program multiple data).
- Same program runs on different processors.
  - Processes do different things based on their ID.
  - Different from Single Instruction Multiple Data (SIMD)!
    - Processes aren't synchronized, and can execute different instructions or different branches of code.
- Process mapping
  - Typically map one process to each processor / core.
    - More processes can cause thrashing, though may also help reduce idling.
  - Programmer can't control the mapping.





# Process creation example

- Include `mpi.h`
- Must call `MPI_Init` and `MPI_Finalize` before and after using MPI functions.

```
#include <mpi.h>

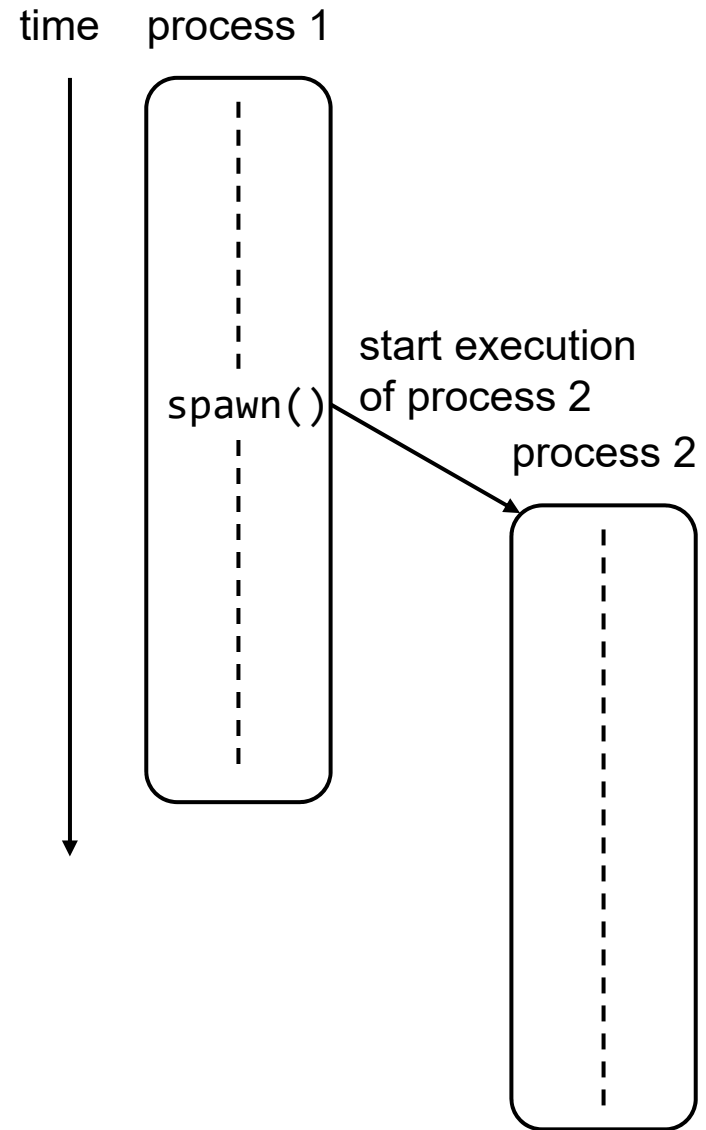
main(int argc, char *argv[]) {
    int npes, myrank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d, Hello World!\n",
           myrank, npes);
    MPI_Finalize(); }
```

- Start program with `mpirun -hostfile hosts -np n myprog`
- Output appears on host starting `myprog`.
- Messages may get printed in arbitrary order, because processes run asynchronously.

# MPMD model

- Separate programs for each processor.
  - Multi program multi data.
- Usually takes master-slave approach.
  - One processor executes master process, other processes started from within master process.
  - Dynamic process creation.
  - Higher overhead from starting processes, but sometimes easier to understand and program.
  - Used e.g. in Parallel Virtual Machine (PVM) programming system.





# MPI in a nutshell

- Many functions, but most programs require only a few functions.
- Initialization.
- Point-to-point communication.
  - Send, receive.
- Collective communication.
  - Broadcast, scatter / gather, all to all, reduce, scan, barrier.
- Derived data types.
  - Contiguous, indexed, struct.
- Communicators, virtual topologies.
- More info at <http://www.mpi-forum.org/docs/>



# Communicators

- Communication domain, aka communicator, is a set of processes that can communicate with each other.
  - A program can define multiple communicators.
  - Processes can belong to multiple communication domains.
  - Used to isolate and organize communication.
- Communicators have type `MPI_Comm`.
- All communication must specify a communicator.
- `MPI_COMM_WORLD` default communicator created automatically.
- Communicator has a size (number of processes in communicator).
  - `int MPI_Comm_size(MPI_Comm comm, int *size)`
- Each process in communicator has a rank (ID).
  - `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Used to allow different processors to do different things.





# Sending messages

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest,  
int tag, MPI_Comm comm)
```


- buf is the data to send, i.e. a program variable.
- There are count items in buf, e.g. the size of the array.
- Buffer items must have type MPI\_Datatype.
  - MPI\_CHAR, MPI\_SHORT, MPI\_INT, MPI\_FLOAT, MPI\_BYTE, MPI\_PACKED
  - Can also send user defined (derived) types.
- Message sent to process in communicator comm with rank dest.
- tag used to differentiate multiple messages between source and destination.
- Messages between pair of processes sent / received in FIFO order.



# Receiving messages

```
int MPI_Recv(void *buf, int count, MPI_Datatype type, int source,
int tag, MPI_Comm comm, MPI_Status *status)
```

- buf gives location to store received message.
- count must be at least number of items to receive.
  - If message too large, MPI\_ERROR\_TRUNCATE will occur.
- Message received from process with rank source in communicator comm.
  - Source can also be set to MPI\_ANY\_SOURCE to receive from any process.
- Will only receive message with tag tag.
  - Tag can also be set to MPI\_ANY\_TAG.
- status useful when receiving using wildcards.
  - MPI\_Status is a struct with fields MPI\_SOURCE, MPI\_TAG, MPI\_ERROR to get info on wildcard message.
- Fairness not guaranteed. If two sends match a receive, only one send completes.



```

#include "mpi.h"
#include <stdio.h>

main(int argc, char *argv[]) {
    int numtasks, rank, dest, source, rc, count, tag=1;
    char inmsg, outmsg='x';
    MPI_Status Stat;    // required variable for receive routines

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // task 0 sends to task 1 and waits to receive a return message
    if (rank == 0) {
        dest = 1;
        source = 1;
        MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    }

    // task 1 waits for task 0 message then returns a message
    else if (rank == 1) {
        dest = 0;
        source = 0;
        MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }

    // query receive Stat variable and print message details
    MPI_Get_count(&Stat, MPI_CHAR, &count);
    printf("Task %d: Received %d char(s) from task %d with tag %d \n",
           rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

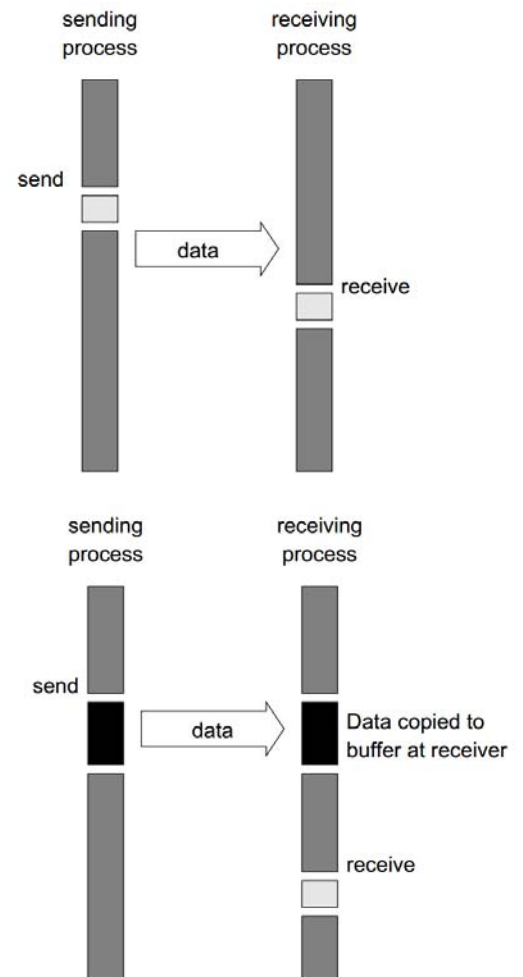
    MPI_Finalize();
}

```

Source: <https://computing.llnl.gov/tutorials/mpi/>

# Blocking communication

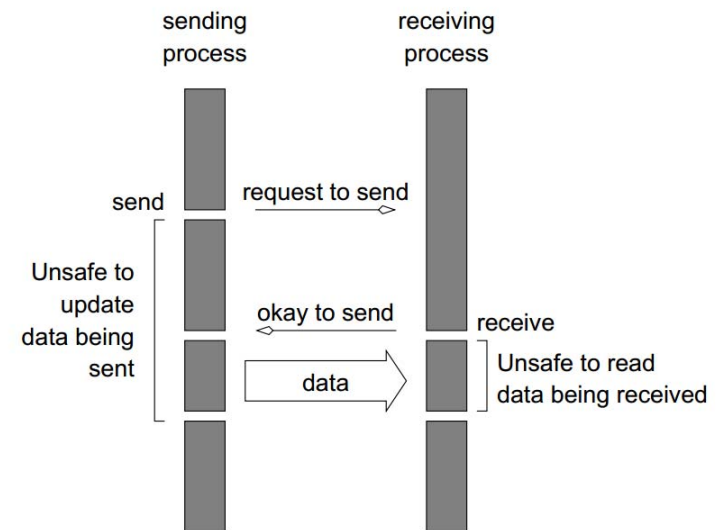
- MPI\_Send is blocking: the function won't return immediately after being called.
- Blocking ensures the data is correctly sent.
  - Sender and receiver aren't synchronized, so sender must wait until receiver is ready.
- Waiting time depends on if an MPI implementation uses a hardware send buffer.
  - If there's a buffer, MPI\_Send returns after data copied from application buffer to hardware buffer.
    - Data is later transferred to HW buffer of receiver.
  - With no buffer, sender waits until matching receive occurred.
    - At this point, the data is either in the receiver's hardware or application buffer.
- MPI\_Recv always causes process to block until data is received into its store buffer.



Source: Introduction to  
Parallel Computing.  
Grama et al.

# Nonblocking communication

- Would be more efficient if process can immediately resume execution after calling send or receive.
- This requires a HW buffer.
  - Process continues with computation while data is transferred from process buffer to HW buffer.
- But must not modify process buffer until transfer complete.
- Need functions to detect when transfer complete.



Source: Introduction to Parallel Computing.  
Grama et al.



# Nonblocking communication

```
int MPI_Isend(void *buf, int count, MPI_Datatype type, int  
dest, int tag, MPI_Comm comm, MPI_Request *req);
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype type, int  
source, int tag, MPI_Comm comm, MPI_Request *req);
```

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *status);
```

```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
```

- MPI\_Isend and MPI\_Irecv return immediately after being called.
- But must still know when operations complete.
- Use a request object to identify the operation.
  - This object used later to test completion.
  - Can also use to make process wait (block) until operation completes.

# Deadlock example 1

- Must be careful about ordering of sends or receives. Otherwise processes can deadlock, i.e. wait forever.

```
int a[10], b[10], myrank;
MPI_Status status;
...
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1,
             MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2,
             MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2,
             MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1,
             MPI_COMM_WORLD);}
```

```
int a[10], b[10], myrank;
MPI_Status status;
...
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1,
             MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2,
             MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 1,
             MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 2,
             MPI_COMM_WORLD);}
```

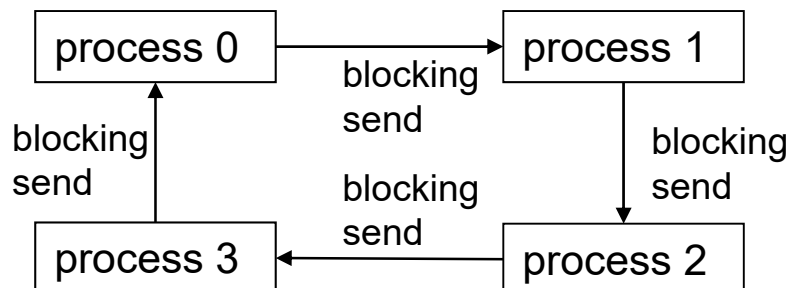
- ❑ If buffering, then probably no deadlock.
- ❑ If no buffering, processes will deadlock waiting to receive from each other.
- ❑ Cause is that tags are sent and received in different orders

- ❑ Resolve the deadlock by using same tag order for send and recv.

# Deadlock example 2

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes,
         1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1)%npes,
         1, MPI_COMM_WORLD);
```

- ❑ Here, *npes* processes in a ring. Process *i* sends to *i+1*, receives from *i-1*.
- ❑ No deadlock with buffering, since `MPI_Send` returns and `MPI_Recv` can occur.
- ❑ Without buffering, processes deadlock blocking for matching receive.



```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 0) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes,
             1, MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1)%npes,
             1, MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1)%npes,
             1, MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes,
             1, MPI_COMM_WORLD);
}
```

- ❑ Solution is to first let even numbered processes send and odd processes recv, then let odd processes send and even processes recv.
- ❑ This is cumbersome. It would be nice to have built-in way to avoid deadlock.



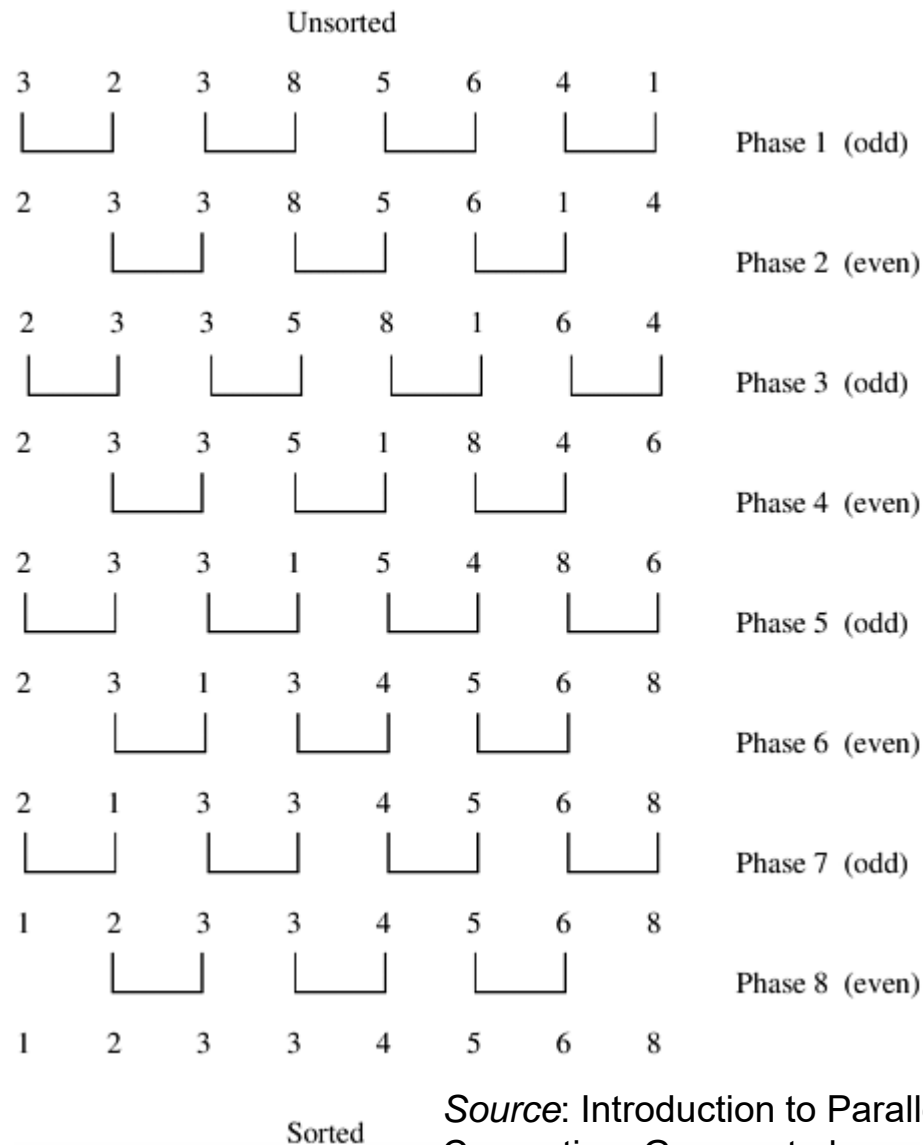
# Send and receive simultaneously

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`

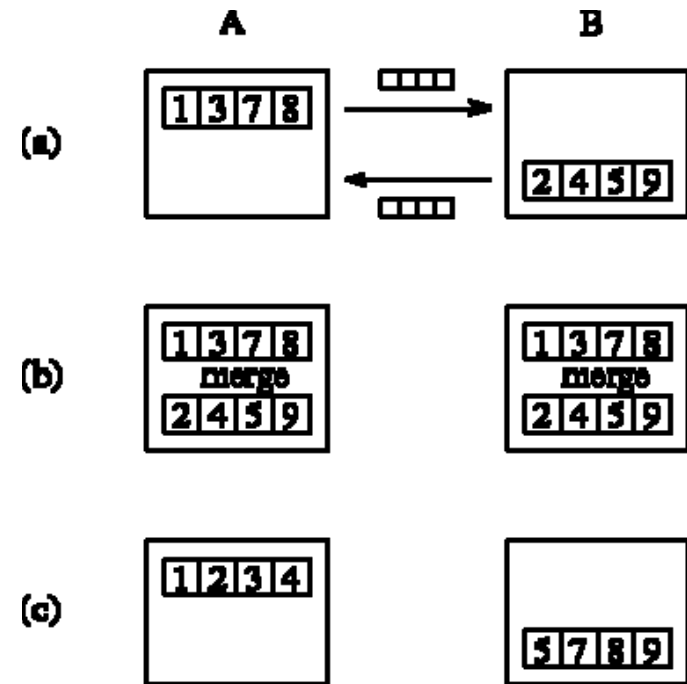
```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Sendrecv(a, 10, MPI_INT, (myrank+1)%npes, 1, b, 10, MPI_INT,
             (myrank-1)%npes, 1, MPI_COMM_WORLD, &status);
```

- MPI\_Sendrecv sends a message and starts a receive before blocking.
- Will block until send completes and receiving application buffer receives message.
  - No deadlock on ring.

# Even-odd sort



Source: Introduction to Parallel Computing. Grama et al.



- ❑ If each processor has multiple values, then instead of simply swapping them, do compare-split step.
- ❑ This combines the values from two processes, and puts the smaller half in left process, larger half in right process.

# Even-odd sort in MPI

```
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

main(int argc, char *argv[])
{
    int n; /* The total number of elements to be sorted */
    int npes; /* The total number of processes */
    int myrank; /* The rank of the calling process */
    int nlocal; /* The local number of elements, and the
        array that stores them */
    int *elmnts; /* The array that stores the local
        elements */
    int *relmnts; /* The array that stores the received
        elements */
    int oddrank; /* The rank of the process during odd-
        phase communication */
    int evenrank; /* The rank of the process during even-
        phase communication */
    int *wspace; /* Working space during the compare-split
        operation */

    int i;
    MPI_Status status;

    /* Initialize MPI and get system information */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    n = atoi(argv[1]);
    nlocal = n/npes; /* Compute the number of elements to
        be stored locally. */
```

```
/* Allocate memory for the various arrays */
elmnts = (int *)malloc(nlocal*sizeof(int));
relmnts = (int *)malloc(nlocal*sizeof(int));
wspace = (int *)malloc(nlocal*sizeof(int));

/* Fill-in the elmnts array with random elements */
srandom(myrank);
for (i=0; i<nlocal; i++)
    elmnts[i] = random();

/* Sort the local elements using the built-in
quicksort routine */
qsort(elmnts, nlocal, sizeof(int), IncOrder);

/* Determine the rank of the processors that myrank
needs to communicate with during the */
/* odd and even phases of the algorithm */
if (myrank%2 == 1) {
    oddrank = myrank-1;
    evenrank = myrank+1;
}
else {
    oddrank = myrank+1;
    evenrank = myrank-1;
}
```

Source: Introduction to Parallel Computing. Grama et al.

# Even-odd sort in MPI

```
/* Set first processor's communication ranks */
if (oddrank == -1 || oddrank == npes)
    oddrank = MPI_PROC_NULL;
if (evenrank == -1 || evenrank == npes)
    evenrank = MPI_PROC_NULL;

/* Main loop of the odd-even sorting algorithm */
for (i=0; i<npes-1; i++) {
    if (i%2 == 1) /* Odd phase */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT,
            oddrank, 1, relmnts, nlocal, MPI_INT,
            oddrank, 1, MPI_COMM_WORLD, &status);
    else /* Even phase */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT,
            evenrank, 1, relmnts, nlocal, MPI_INT,
            evenrank, 1, MPI_COMM_WORLD, &status);

    CompareSplit(nlocal, elmnts, relmnts, wspace,
        myrank < status.MPI_SOURCE);
}

free(elmnts); free(relmnts); free(wspace);
MPI_Finalize();
}
```

```
CompareSplit(int nlocal, int *elmnts, int *relmnts,
    int *wspace, int keepsmall) {
    int i, j, k;

    for (i=0; i<nlocal; i++)
        wspace[i] = elmnts[i];
    /* Copy the elmnts array into the wspace array */

    if (keepsmall) { /* Keep nlocal smaller elts */
        for (i=j=k=0; k<nlocal; k++) {
            if (j == nlocal || (i < nlocal &&
                wspace[i] < relmnts[j]))
                elmnts[k] = wspace[i++];
            else
                elmnts[k] = relmnts[j++];
        }
    }
    else { /* Keep the nlocal larger elements */
        for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
            if (j == 0 || (i >= 0 && wspace[i] >=
                relmnts[j]))
                elmnts[k] = wspace[i--];
            else
                elmnts[k] = relmnts[j--];
        }
    }
}

int IncOrder(const void *e1, const void *e2) {
    return (*((int *)e1) - *((int *)e2));
}
```



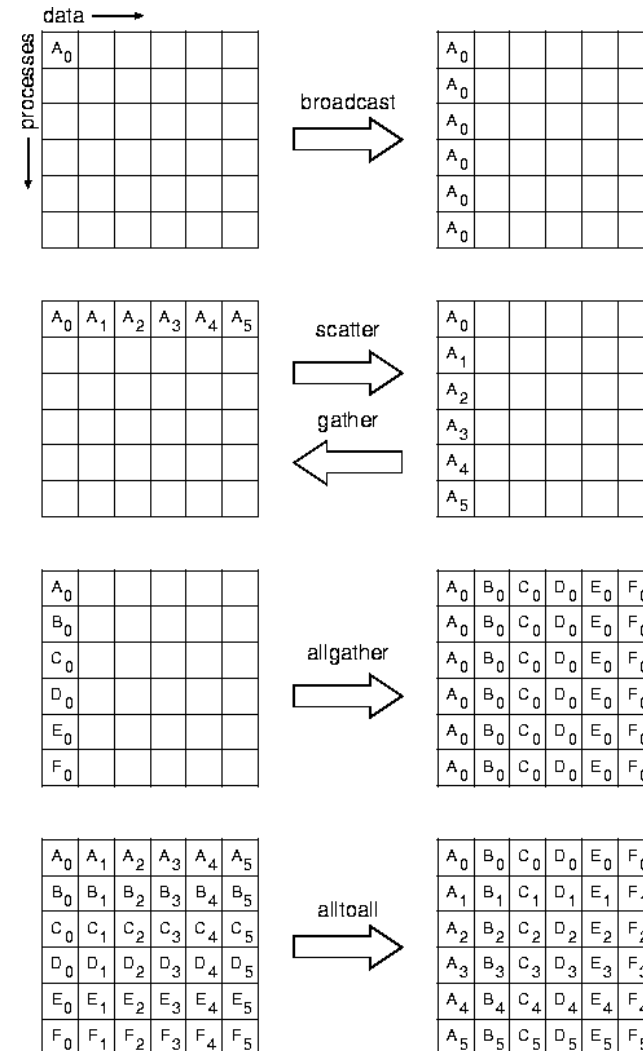
# Collective communication

- Collective communication are routines that provide message passing between a group of processes.
  - One-to-many, e.g. broadcast, scatter
  - Many-to-one, e.g. gather, reduce
  - Many-to-many, e.g. All-to-all
- Collective communication may be implemented using point-to-point routines.
- Specialized implementation of collective communication is likely to be more efficient.
- Collectives are blocking in MPI 1 and 2, but MPI 3 includes nonblocking collectives.

# Collective communication

- As with point to point communication, need to specify the communicator.

- MPI\_Bcast()
- MPI\_Reduce()
  - MPI\_Allreduce()
- MPI\_Scan()
- MPI\_Gather()
- MPI\_Scatter()
  - MPI\_Scatterv()
- MPI\_Alltoall()
  - MPI\_Alltoallv()
- MPI\_Barrier()

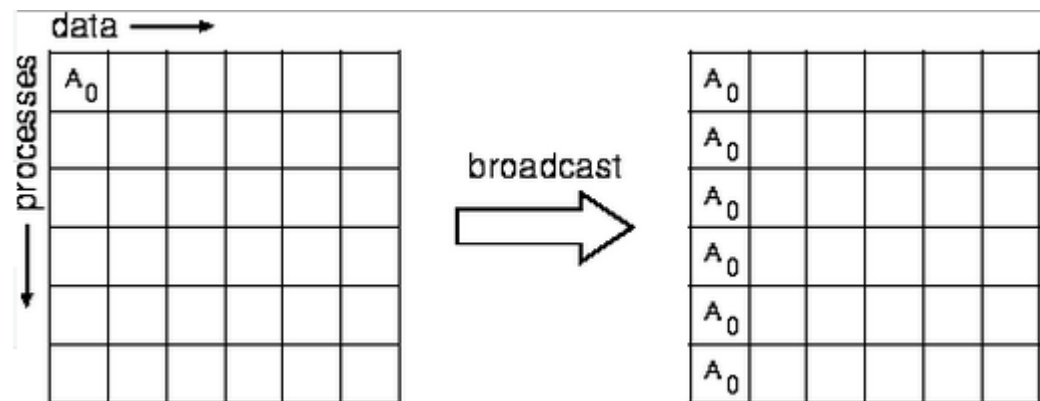


Source: <http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/coll-fig1.gif>

# Broadcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type, int
source, MPI_Comm comm)
```

- Send the data in buf at the source process to all processes (including source) in comm.
  - Non-source nodes store the incoming data in their buf.
- Note that even though the message is coming from source, all processes, including the receiving ones, call MPI\_Bcast().
  - See the example on the next slide.





# Broadcast example

```
#include <mpi.h>

int main(int argc, char** argv) {
    int myrank;
    int buf;
    const int root=0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == root) {
        buf = 1; }

    /* All procs call bcast.  buf is sent from
       root to all procs */
    MPI_Bcast(&buf, 1, MPI_INT, root,
              MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

- ❑ This code causes process root to send buf (i.e. the value 1) to all the processes.
- ❑ Since MPI uses an SPMD model, all processes run the same code.
- ❑ So even the non-root processes do MPI\_Bcast(&buf, 1, MPI\_INT, root, MPI\_COMM\_WORLD)
- ❑ However, since their myrank != root, then instead of sending, the MPI\_Bcast() will cause them to receive.
- ❑ Think of MPI\_Bcast() as saying “participate in a broadcast”, with the type of participation depending on the process’s rank.
- ❑ The other collective operations work similarly, i.e. all procs run the same code, but do different things depending on their rank.

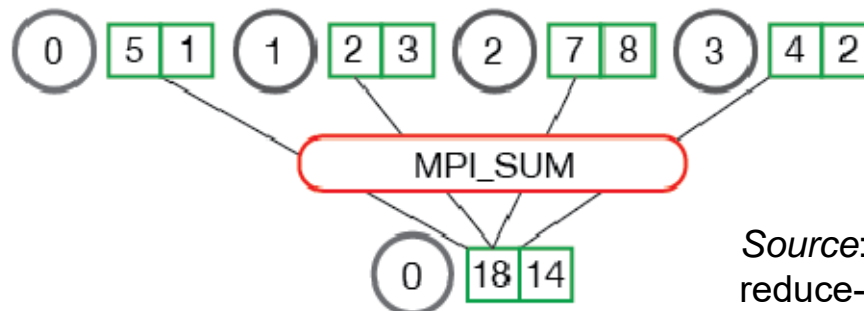


# Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype type, MPI_Op op, int target, MPI_Comm comm)
```

- Combine the elements stored in the buffer sendbuf of each process in comm using the operation op, and store the combined values in recvbuf of process target.
- **Ex** Add the values from all the processes.
- Operations include MPI\_MAX, MPI\_MIN, MPI\_SUM, MPI\_PROD, MPI\_LAND, MPI\_MAXLOC.
  - MPI\_MAXLOC p returns a pair of values (v,l) in recvbuf, where v is the max value, and l is the smallest ranked process with the value.

MPI\_Reduce



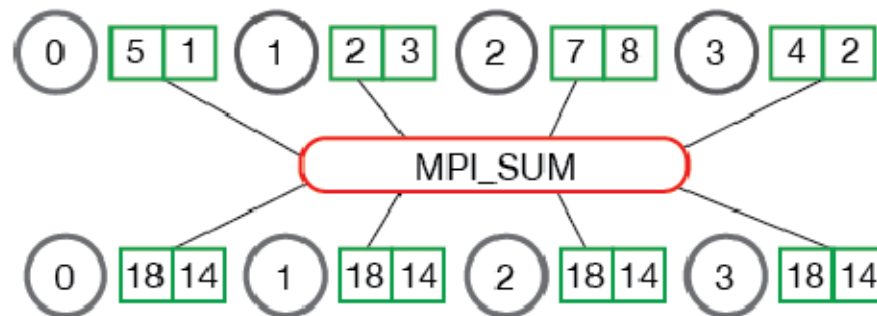
Source: <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

# Allreduce

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

- Same as reduce, but store the result at all the processes.

MPI\_Allreduce

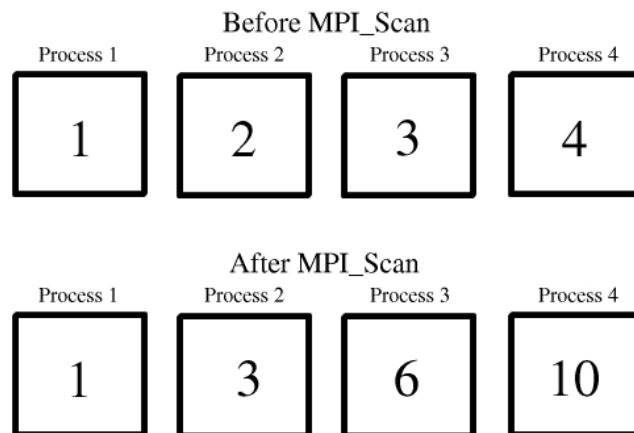


Source: <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

# Scan

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

- List the data in the sendbuf's of all the processes by process rank.
- Process  $i$  applies  $op$  (elementwise) to the first  $i$  sendbuf's and stores the result in its recvbuf.
- Ex Add values from the processes.
- This operation is also called prefix sum.
  - Can apply other operators besides sum, as in reduce.



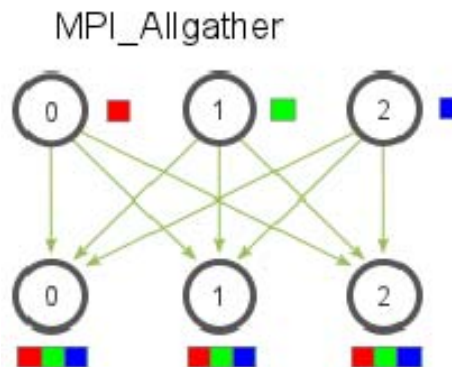
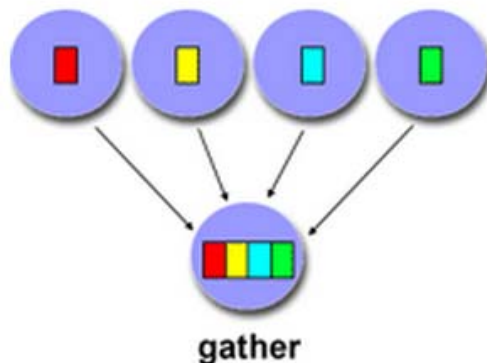
Source: [http://www.rc.usf.edu/tutorials/classes/tutorial/mpi/images/scan\\_ex.jpg](http://www.rc.usf.edu/tutorials/classes/tutorial/mpi/images/scan_ex.jpg)

# Gather

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int target, MPI_Comm comm)
```

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- Gather collects the sendbuf's from all the processes in comm into the recvbuf of process target.
  - If  $k$  processes in comm each with sendbuf of size  $t$ , then target will receive  $k \cdot t$  items.
- Allgather collects this data at all the processes.

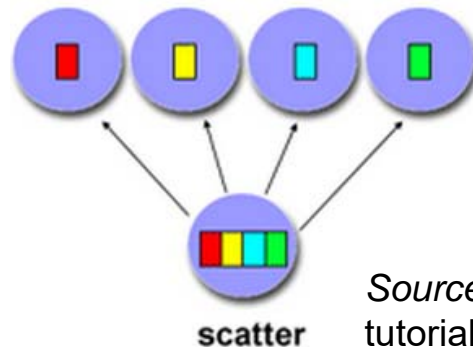


# Scatter

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int source, MPI_Comm comm)
```

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, MPI_Comm comm)
```

- Scatter sends the data at location  $i \times \text{recvcount}$  in sendbuf to the recvbuf of  $i$ 'th process in comm.
  - All scattered data have the same size.
- Scatterv sends data of different sizes to different processes.
  - Sizes of data specified in sendcounts.
  - Data to send to process  $i$  starts at location  $\text{displs}[i]$  in sendbuf.
  - For the receiving processes (i.e. all processes other than source), the first 4 arguments don't matter.
  - Each receiving process sets recvcount to the number of items it expects to get.



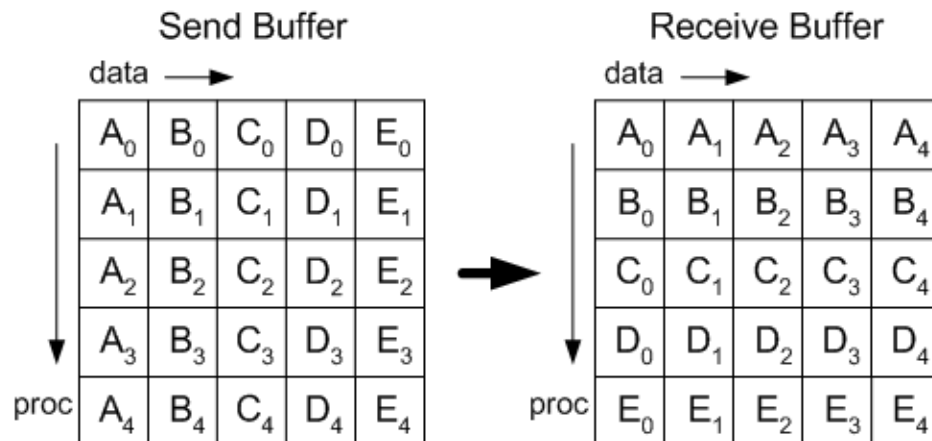
Source: <https://computing.llnl.gov/tutorials/mpi/>

# Alltoall

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *rdispls,
MPI_Datatype recvtype, MPI_Comm comm)
```

- Alltoall sends to process  $i$  sendcount elements from sendbuf starting from location  $i * \text{sendcount}$  of sendbuf.
- Alltoallv is similar, but allows sending different size data to each process.

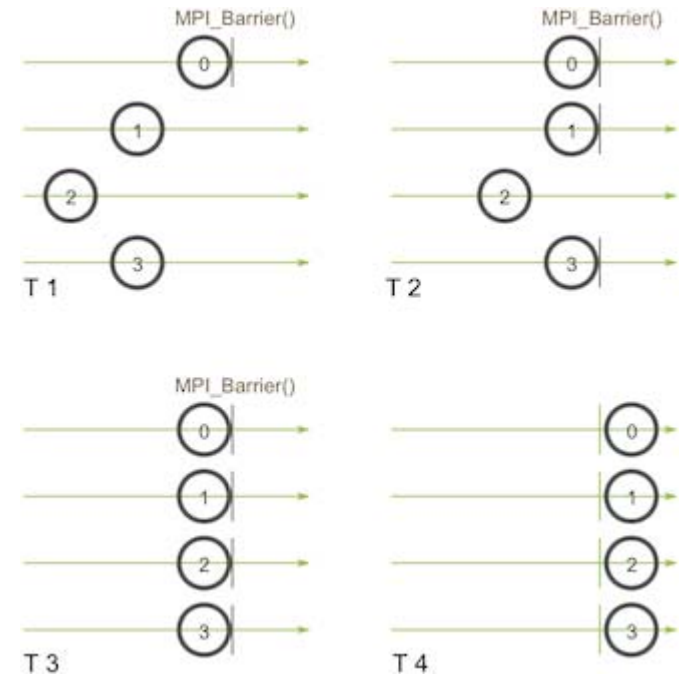


Source: <https://cvw.cac.cornell.edu/MPIcc/alltoall>

# Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

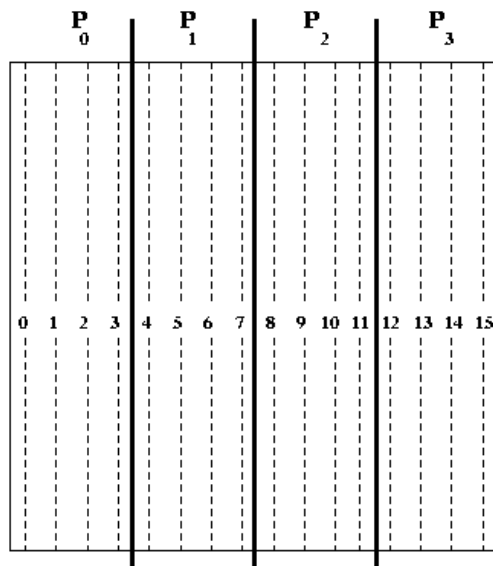
- Used to synchronize all the processes in comm at a certain point in the code before any of them can proceed.
  - All processes in comm must reach the barrier. Otherwise, all processes will block forever.
- Can use multiple barriers.
  - All processes must reach the b'th barrier before proceeding to barrier b+1.



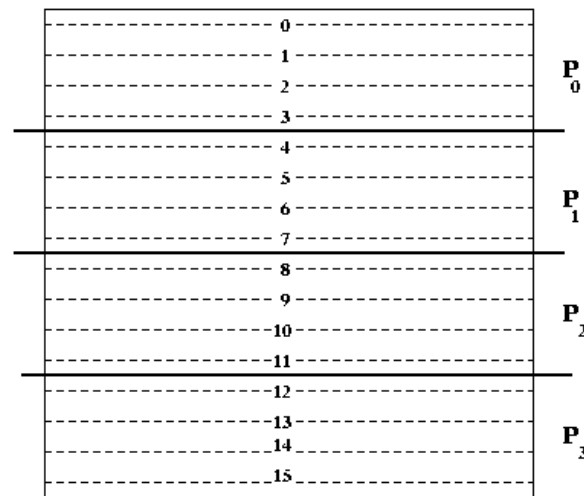
Source: <http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/barrier.png>

# Storing a matrix

- Linear algebra one of the main applications for parallel computing.
- A large matrix can be stored in a distributed fashion on the parallel processors using row or column stripping.
- Even with columnwise block stripping, we store columns in row-major format.



(a) Columnwise block stripping



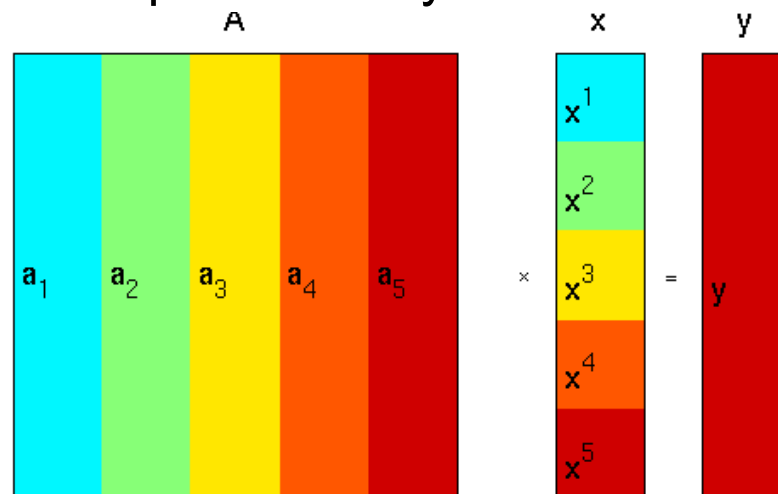
(b) Rowwise block stripping

Source: <http://www.cse.iitd.ernet.in/~dheerajb/MPI/Document/onedblk.gif>



# Column-wise matrix vector mult

- Compute  $y = A \cdot x$  using  $p$  processes.
  - $A$  is an  $n \times n$  matrix,  $x$  and  $y$  are  $n \times 1$  vectors.
  - Each process holds  $n/p$  columns from  $A$  and  $n/p$  values from  $x$ .
- Each process does dot product on its columns with its portions of  $x$ .
  - Process ends up with  $n$  (partial) dot products.
- $y[i] = \text{sum of the } p \text{ partial dot products for the } i\text{'th row held at the different processes.}$ 
  - Do a sum reduce for each row of  $y$ .
- Finally, distribute  $n/p$  values of  $y$  to each of the processes.



Source: [https://www.informatik.uni-konstanz.de/fileadmin/informatik/ag-sauepe/Webpages/lehre/na\\_08/Lab1/1\\_Preliminaries/html/matrixVectorProduct.html](https://www.informatik.uni-konstanz.de/fileadmin/informatik/ag-sauepe/Webpages/lehre/na_08/Lab1/1_Preliminaries/html/matrixVectorProduct.html)



# Column-wise matrix vector mult

```
ColMatrixVectorMultiply(int n, double *a,
    double *x, double *y, MPI_Comm comm)
{
    int i, j;
    int nlocal;
    double *py;
    double *fy;
    int npes, myrank;
    MPI_Status status;

    /* Get communicator information */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);
    nlocal = n/npes;

    /* Allocate memory for arrays storing
    intermediate results. */
    py = (double *)
        malloc(n*sizeof(double));
    fy = (double *)
        malloc(n*sizeof(double));
```

```
/* Compute partial-dot products that
correspond to local columns of A*/
    for (i=0; i<n; i++) {
        py[i] = 0.0;
        for (j=0; j<nlocal; j++)
            py[i] += a[i*nlocal+j]*x[j];
    }

    /* Sum-up results by performing an
    element-wise reduction operation */
    MPI_Reduce(py, fy, n, MPI_DOUBLE,
        MPI_SUM, 0, comm);

    /* Redistribute fy in a fashion similar to
    vector x */
    MPI_Scatter(fy, nlocal, MPI_DOUBLE,
        y, nlocal, MPI_DOUBLE, 0, comm);

    free(py); free(fy);
}
```

Source: Introduction to Parallel  
Computing. Grama et al.



# Derived data types

- MPI comes with a number of standard built in types.
- To send more complicated data structures, programmer defines derived types.
- Four main types, contiguous, vector, indexed and struct.
- `MPI_Type_commit(&datatype)` commits defined datatype.
- `MPI_Type_extent(&datatype)` returns size in bytes of the datatype.
- `MPI_Type_free(&datatype)` Deallocates a datatype, prevent memory exhaustion.



# Defining new datatypes

- `MPI_Type_contiguous(count, oldtype, &newtype)`
  - Place count copies of oldtype contiguously.
- `MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)`
  - Similar to contiguous, use count copies of oldtype each of blocklength, with stride bytes in between.
- `MPI_Type_indexed (count, blocklens[], offsets[], old_type, &newtype)`
  - Similar to vector, except blocklens and offsets vectors (of size count) specify size and offset of each element in the datatype.
- `MPI_Type_struct (count, blocklens[], offsets[], old_types, &newtype)`
  - Form a datatype according to a completely defined map of the component data types.

```

1  #include "mpi.h"
2  #include <stdio.h>
3  #define SIZE 4
4
5  main(int argc, char *argv[]) {
6  int numtasks, rank, source=0, dest, tag=1, i;
7  float a[SIZE][SIZE] =
8      {1.0, 2.0, 3.0, 4.0,
9        5.0, 6.0, 7.0, 8.0,
10       9.0, 10.0, 11.0, 12.0,
11       13.0, 14.0, 15.0, 16.0};
12  float b[SIZE];
13
14  MPI_Status stat;
15  MPI_Datatype columntype;    // required variable
16
17
18  MPI_Init(&argc,&argv);
19  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
21
22  // create vector derived data type
23  MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
24  MPI_Type_commit(&columntype);
25
26  if (numtasks == SIZE) {
27      // task 0 sends one element of columntype to all tasks
28      if (rank == 0) {
29          for (i=0; i<numtasks; i++)
30              MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
31      }
32
33      // all tasks receive columntype data from task 0
34      MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
35      printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
36            rank,b[0],b[1],b[2],b[3]);
37  }
38  else
39      printf("Must specify %d processors. Terminating.\n",SIZE);
40
41  // free datatype when done using it
42  MPI_Type_free(&columntype);
43  MPI_Finalize();
44  }

```

## MPI\_Type\_vector

count = 4; blocklength = 1; stride = 4;  
MPI\_Type\_vector(count, blocklength, stride, MPI\_FLOAT,  
&columntype);

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

MPI\_Send(&a[0][1], 1, columntype, dest, tag, comm);

2.0	6.0	10.0	14.0
-----	-----	------	------

1 element of  
columntype

rank= 0	b= 1.0	5.0	9.0	13.0
rank= 1	b= 2.0	6.0	10.0	14.0
rank= 2	b= 3.0	7.0	11.0	15.0
rank= 3	b= 4.0	8.0	12.0	16.0

Source: <https://computing.llnl.gov/tutorials/mpi>

```

1  #include "mpi.h"
2  #include <stdio.h>
3  #define NELEM 25
4
5  main(int argc, char *argv[]) {
6  int numtasks, rank, source=0, dest, tag=1, i;
7
8  typedef struct {
9      float x, y, z;
10     float velocity;
11     int n, type;
12 } Particle;
13 Particle p[NELEM], particles[NELEM];
14 MPI_Datatype particletype, oldtypes[2]; // required variables
15 int blockcounts[2];
16
17 // MPI_Aint type used to be consistent with syntax of
18 // MPI_Type_extent routine
19 MPI_Aint offsets[2], extent;
20
21 MPI_Status stat;
22
23 MPI_Init(&argc, &argv);
24 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
25 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
26
27 // setup description of the 4 MPI_FLOAT fields x, y, z, velocity
28 offsets[0] = 0;
29 oldtypes[0] = MPI_FLOAT;
30 blockcounts[0] = 4;
31
32 // setup description of the 2 MPI_INT fields n, type
33 // need to first figure offset by getting size of MPI_FLOAT
34 MPI_Type_extent(MPI_FLOAT, &extent);
35 offsets[1] = 4 * extent;
36 oldtypes[1] = MPI_INT;
37 blockcounts[1] = 2;
38
39 // define structured type and commit it
40 MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);
41 MPI_Type_commit(&particletype);
42
43 // task 0 initializes the particle array and then sends it to each task
44 if (rank == 0) {
45     for (i=0; i<NELEM; i++) {
46         particles[i].x = i * 1.0;
47         particles[i].y = i * -1.0;
48         particles[i].z = i * 1.0;
49         particles[i].velocity = 0.25;
50         particles[i].n = i;
51         particles[i].type = i % 2;
52     }
53     for (i=0; i<numtasks; i++)
54         MPI_Send(particles, NELEM, particletype, i, tag, MPI_COMM_WORLD);
55 }
56
57 // all tasks receive particletype data
58 MPI_Recv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &stat);
59
60 printf("rank= %d   %3.2f %3.2f %3.2f %3.2f %d %d\n", rank, p[3].x,
61        p[3].y, p[3].z, p[3].velocity, p[3].n, p[3].type);
62
63 // free datatype when done using it
64 MPI_Type_free(&particletype);
65 MPI_Finalize();
66 }

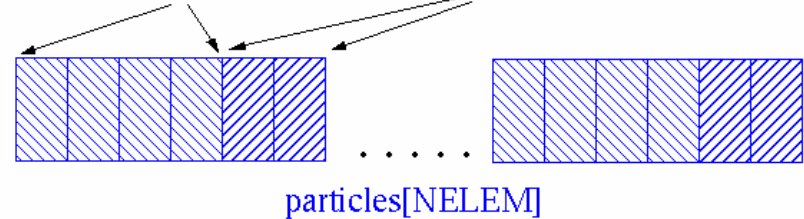
```

## MPI\_Type\_struct

```
typedef struct { float x,y,z,velocity; int n,type; } Particle;
Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);
```

```
count = 2; oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT
offsets[0] = 0; offsets[1] = 4 * extent;
blockcounts[0] = 4; blockcounts[1] = 2;
```



```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);
```

```
MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

rank= 0	3.00	-3.00	3.00	0.25	3	1
rank= 2	3.00	-3.00	3.00	0.25	3	1
rank= 1	3.00	-3.00	3.00	0.25	3	1
rank= 3	3.00	-3.00	3.00	0.25	3	1

# More on communicators

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #define NPROCS 8
4
5  main(int argc, char *argv[]) {
6      int rank, new_rank, sendbuf, recvbuf, numtasks,
7          ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
8      MPI_Group orig_group, new_group; // required variables
9      MPI_Comm new_comm; // required variable
10
11     MPI_Init(&argc,&argv);
12     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
14
15     if (numtasks != NPROCS) {
16         printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
17         MPI_Finalize();
18         exit(0);
19     }
20
21     sendbuf = rank;
22
23     // extract the original group handle
24     MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
25
26     // divide tasks into two distinct groups based upon rank
27     if (rank < NPROCS/2) {
28         MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
29     }
30     else {
31         MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
32     }
33
34     // create new new communicator and then perform collective communication
35     MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
36     MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
37
38     // get rank in new group
39     MPI_Group_rank(new_group, &new_rank);
40     printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
41
42     MPI_Finalize();
43 }
```

- Communicators are a programming aid to group processes that perform a common task.
- Collective computations are done within a communicator.
  - MPI 2 allows collectives between communicators.
- Processes have a unique rank in each communicator they belong to.

rank= 7	newrank= 3	recvbuf= 22
rank= 0	newrank= 0	recvbuf= 6
rank= 1	newrank= 1	recvbuf= 6
rank= 2	newrank= 2	recvbuf= 6
rank= 6	newrank= 2	recvbuf= 22
rank= 3	newrank= 3	recvbuf= 6
rank= 4	newrank= 0	recvbuf= 22
rank= 5	newrank= 1	recvbuf= 22



# Virtual topologies

- Map a communicator of processes to a virtual topology.
- Virtual topology may not correspond to the physical topology of underlying hardware.
- Nevertheless, allows MPI implementation to try to optimize mapping of virtual to physical topology, e.g. mapping a virtual mesh to a physical hypercube.
- Also a programming convenience, to match topology of application communications to virtual topology.
  - A 3D physics code can use a virtual 3D mesh topology.
- Main topologies are regular torus and graph.
  - `MPI_Cart_create(MPI_Comm oldcomm, int ndim, int dims[], int qperiodic[], int qreorder, MPI_Comm *newcomm)`
  - `MPI_Dist_graph_create_adjacent(MPI_Comm oldcomm, int indegree, int sources[], int sourceweights[], int outdegree, int dests[], int destweights[], MPI_Info info, int qreorder, MPI_Comm *newcomm)`



```

1  #include "mpi.h"
2  #include <stdio.h>
3  #define SIZE 16
4  #define UP 0
5  #define DOWN 1
6  #define LEFT 2
7  #define RIGHT 3
8
9  main(int argc, char *argv[]) {
10     int numtasks, rank, source, dest, outbuf, i, tag=1,
11         inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL},
12         nbrs[4], dims[2]={4,4},
13         periods[2]={0,0}, reorder=0, coords[2];
14
15     MPI_Request reqs[8];
16     MPI_Status stats[8];
17     MPI_Comm cartcomm; // required variable
18
19     MPI_Init(&argc,&argv);
20     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
21
22     if (numtasks == SIZE) {
23         // create cartesian virtual topology, get rank, coordinates, neighbor ranks
24         MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
25         MPI_Comm_rank(cartcomm, &rank);
26         MPI_Cart_coords(cartcomm, rank, 2, coords);
27         MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
28         MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);
29
30         printf("rank= %d coords= %d %d neighbors(u,d,l,r)= %d %d %d %d\n",
31             rank, coords[0], coords[1], nbrs[UP], nbrs[DOWN], nbrs[LEFT],
32             nbrs[RIGHT]);
33
34         outbuf = rank;
35
36         // exchange data (rank) with 4 neighbors
37         for (i=0; i<4; i++) {
38             dest = nbrs[i];
39             source = nbrs[i];
40             MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
41                 MPI_COMM_WORLD, &reqs[i]);
42             MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
43                 MPI_COMM_WORLD, &reqs[i+4]);
44         }
45
46         MPI_Waitall(8, reqs, stats);
47
48         printf("rank= %d inbuf(u,d,l,r)= %d %d %d %d\n",
49             rank, inbuf[UP], inbuf[DOWN], inbuf[LEFT], inbuf[RIGHT]);
50     } else
51         printf("Must specify %d processors. Terminating.\n", SIZE);
52
53     MPI_Finalize();
54 }

```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

```

rank= 0 coords= 0 0 neighbors(u,d,l,r)= -1 4 -1 1
rank= 0 inbuf(u,d,l,r)= -1 4 -1 1
rank= 8 coords= 2 0 neighbors(u,d,l,r)= 4 12 -1 9
rank= 8 inbuf(u,d,l,r)= 4 12 -1 9
rank= 1 coords= 0 1 neighbors(u,d,l,r)= -1 5 0 2
rank= 1 inbuf(u,d,l,r)= -1 5 0 2
rank= 13 coords= 3 1 neighbors(u,d,l,r)= 9 -1 12 14
rank= 13 inbuf(u,d,l,r)= 9 -1 12 14
...
...
rank= 3 coords= 0 3 neighbors(u,d,l,r)= -1 7 2 -1
rank= 3 inbuf(u,d,l,r)= -1 7 2 -1
rank= 11 coords= 2 3 neighbors(u,d,l,r)= 7 15 10 -1
rank= 11 inbuf(u,d,l,r)= 7 15 10 -1
rank= 10 coords= 2 2 neighbors(u,d,l,r)= 6 14 9 11
rank= 10 inbuf(u,d,l,r)= 6 14 9 11
rank= 9 coords= 2 1 neighbors(u,d,l,r)= 5 13 8 10
rank= 9 inbuf(u,d,l,r)= 5 13 8 10

```



# MPI 2 and 3

## ■ MPI 2 added

- ☐ Dynamic process creation.
- ☐ Collectives across communicators.
- ☐ Parallel I/O.
- ☐ One sided communication.
  - Access memory on a remote process without it “expecting it”.

## ■ MPI 3 added

- ☐ Nonblocking collectives.
- ☐ Better support for multi-threading.
- ☐ Improved one sided communications.