– Typeset by FoilTEX –

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

FORTRAN and MPI

Message Passing Interface (MPI)

Day 2

Course plan:

- MPI - General concepts
- Communications in MPI
  – Point-to-point communications
  – Collective communications
- Parallel debugging
- Advanced MPI: user-defined data types, functions
  – Linear Algebra operations
- Advanced MPI: communicators, virtual topologies
  – Parallel sort algorithms
- Parallel performance. Summary. Tendencies

Basic notions and definitions

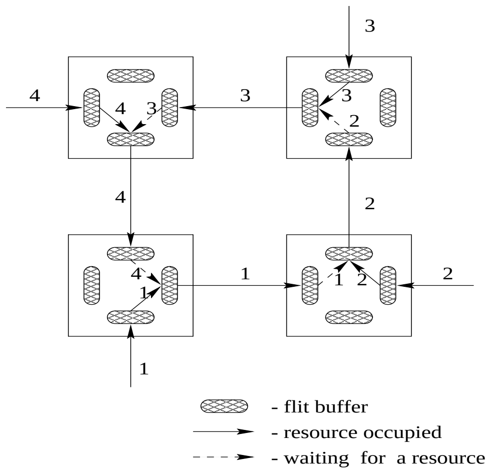
The fundamental characteristics of a communication network are:

network topology	direct (static) or dynamic networks
routing policy	specifies how messages (respectively, parts of a message, called *packages*) choose paths through the network
flow control policy	deals with allocation of network resources, namely, communication channels (links) and buffers, to packages as they are processed through the network

A common technique in modern networks is to divide the message in packages, and the packages further in small units, called *flow-control units* (*flits*), and communicate them in a pipelined fashion.

If, while traversing the network, the message requests a resource (a channel or a buffer) which is in use by some other message, the message cannot proceed further and is blocked. When messages are blocked due to waiting for mutually occupied resources, a *deadlock* occurs.

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS DCAMM
Communications on parallel architectures



Deadlock situation with four messages.

Deadlocks can be avoided by using appropriate routing techniques.

Routing

• ***deterministic*** routing $--\,>$ the message is communicated via a fixed path, connecting the source and the destinations, determined during the initialization of the communication. *Deadlock-free but limits the network performance.*

• ***adaptive*** routing $--\,>$ the route can change depending on the particular network situation. *Better network performance but higher chance for deadlocks*.

$T(A,p)(=T_p) = T_{comp} + T_{comm}$

$\max\{T_{comp}, T_{comm}\} \leq T_p \leq T_{comp} + T_c$!!!

$T_{comp} = T_s(A) + \frac{T_p(A)}{p}$

$T_{comm} = \tau + b\,\ell\,N$, where

- τ startup time, including
  - time to establish a connection between the source processor and the router;
  - time to determine the route by executing the routing algorithm;
  - time to prepare the message by adding a header, trailer and error correction information.
- b the time needed to transfer one word along a connection link (*per-word-transfer* time)
- ℓ the links to be traversed
- N the amount of words to be transfered

$\frac{1}{b}$ - channel bandwidth

The basic communication operations

(i) moving data from one processor to another

(ii) moving the same data packet from one processor to all others – *one-to-all* broadcast or just a *broadcast operation*

(iii) moving a different message from each processor to every other processor – *all-to-all* broadcast.

(iv) scattering (gathering) data from (in) one processor to (from) all others. In the scatter operation, a node sends a packet to every other processor. Gather is dual to scatter.

(v) multiscattering or multigathering of data. The multiscatter operation consists of a scatter from every node. Multigather is defined similarly. The difference between the broadcast (ii) and the scatter (iv) is that in the scatter operations a *different data set* is sent to every processor.

Point-to-point communications

MPI provides a set of **SEND** and **RECEIVE** functions that allow the communication of **typed** data with an associated **tag**.

Typing of the message contents is necessary for heterogeneous support.

The **tag** allows selectivity of messages at the receiving end: one can receive on a particular tag, or one can wild-card this quantity, allowing reception of messages with any tag.

MPI provides **blocking** and **nonblocking** send and receive functions.

In the **blocking** version, send call blocks until the send buffer can be reclaimed as well as the receive functions blocks until the receive buffer actually contains the contents of the message.

The **nonblocking** **send** and **receive** functions allow the possible overlap of message transmittal with computation, or the overlap of multiple message with one-another.

Message envelope

Source	for send-operations implicitly determined by the identity of the message sender
Destination	specified by the **dest** argument; the range of valid values for **dest** is $0, 1, \ldots, n-1$; this range includes the *rank* of the sender, so each process may send a message to itself
Communicator	specified by **comm** argument; represents a communication domain; default communication domain is **MPI_COMM_WORLD**
Tag	specified by the **tag** argument; the range of valid values for **tag** is $0, 1, \ldots, impl\_dep$, where the value of $impl\_dep$ is implementation dependent; MPI requires that $impl\_dep$ be not less than 32767

Both blocking and nonblocking communications have **modes**, which allow to choose the semantics of the send operation. The four **modes** are:

- **standard** - the completion of the send does not necessarily mean that the matching receive has started, and no assumption should be made in the application program about whether the out-going data is buffered by MPI;

- **buffered** - the user can guarantee that a certain amount of buffering space is available;

- **synchronous** - rendezvous semantics between sender and receiver is used;

- **ready** - the user asserts that the matching receive already has been posted.

Standard Send

Using standard send means that the mode of sending may be synchronous or buffered (see below). This means that upon completion, although the send buffer can be safely re-used, the message may or may not have arrived at the destination.

It should not be assumed that sending will complete before receiving begins. Therefore, two machines should not use blocking standard sends to exchange messages as this may cause a deadlock.

Processes need to guarantee to eventually receive all messages that have been sent to them, otherwise a network overload may occur and an error may occur.

Synchronous Send

A synchronous send does not complete until acknowledgement of receipt is received. A synchronous send is slower than a standard or buffered send since the send process remains idle until the receive process catches up. However, as an advantage, synchronous sending is safer and more predictable as a network cannot be overloaded as long as processes guarantee they will eventually receive the message.

Buffered Send

A buffered send copies the message to a system buffer before the message is then received from this buffer.

This mode of sending guarantees to complete immediately and so is quicker than standard sending. It is also more predictable, if the network overloads then an error will be caused. Unfortunately, it cannot be assumed that adequate pre-allocated buffer space will exist and therefore a buffer must be specifically created, attached to (and subsequently detached from) a buffered send.

A buffered send attaches a buffer using the routine "MPI_Buffer_attach", called before the send call, and detaches the buffer using "MPI_Buffer_detach", called after the send has completed.

Ready Send

Similar to a buffered send, a ready send completes immediately. The communication is guaranteed to succeed is a matching receive is already posted.

However, if a matching receive does not exist the outcome is undefined. This distinguishes the ready send mode from all other modes of sending.

Ready sends are mainly used when performance is critical. For the user who is not so concerned about efficiency the mode is not recommended. As with buffered send, the blocking and non-blocking versions are equivalent.

RECEIVE

Messages are received by posting a call to MPI_Recv that matches a posted MPI send. For the receive call to be successful, the datatype argument must be identical to the datatype specified in the equivalent argument in the send call.

A receive call matches a send call through the "source" and "tag" arguments. This means that a process will only receive a message from the specified source, with a specified tag.

It is possible to use the constants `MPI_ANY_SOURCE` and `MPI_ANY_TAG` respectively for these arguments, allowing the receipt of a message from any process, with any tag.

Rules of Point to Point Communication

- Messages do not overtake each other. If a process sends two messages and another process posts two matching receives, the messages will be received in the order that they were sent.

- It is not possible for a matching send and receive to remain outstanding. Hopefully both the send and receive complete, but for example if two sends (receives) are posted with one matching receive (send), then one send (receive) will fail.

- The message sent by the send call must have the same datatype as the message expected by the receive type. The datatypes posted should be MPI datatypes.

Blocking SEND

MPI_SEND (buf, count, datatype, dest, tag, comm, status)

IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	datatype	datatype of each entry
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

```
int MPI_SEND(void* buf, int count, MPI_Datatype
  datatype, int dest, int tag, MPI_Comm comm)
```


Blocking RECEIVE

MPI_RECV(buf, count, datatype, source, tag, comm, status)

IN	buf	initial address of receive buffer
IN	count	number of entries to receive
IN	datatype	datatype of each entry
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	return status

```
MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
<type> buf(*)
INTEGER count, datatype, source, tag, comm, status(MPI_STATUS_SIZE), ierror
```

```
 if (me.ne.0) then
    call MPI_RECV(nnode,1,MPI_INTEGER,0,1,
>                        MPI_COMM_WORLD,status,ierr)
    call MPI_RECV(nedge,1,MPI_INTEGER,0,2,
>                        MPI_COMM_WORLD,status,ierr)
    call MPI_RECV(nface,1,MPI_INTEGER,0,3,
>                        MPI_COMM_WORLD,status,ierr)
 else
    do iPE=1,nPEs-1
    call MPI_SEND(NodePerProc(iPE),1,MPI_INTEGER,iPE,1,
>                        MPI_COMM_WORLD,status,ierr)
    call MPI_SEND(EdgePerProc(iPE),1,MPI_INTEGER,iPE,2,
>                        MPI_COMM_WORLD,status,ierr)
    call MPI_SEND(FacePerProc(iPE),1,MPI_INTEGER,iPE,3,
>                        MPI_COMM_WORLD,status,ierr)
    enddo
 endif
```


Point-to-point communications: **Combined SEND-RECEI**

MPI_SENDRECV executes a blocking send and receive operation. Both send and receive use the same communicator, but may have distinct tag arguments. The send and receive buffers must be disjoint.

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN	sendbuf	initial address of send buffer
IN	sendcount	number of entries to send
IN	sendtype	type of entries in the send buffer
IN	dest	rank of destination
IN	sendtag	send tag
OUT	recvbuf	initial address of receive buffer
IN	recvcount	number of entries to receive
IN	recvtype	datatype of each entry
IN	source	rank of source
IN	recvtag	recv tag
IN	comm	communicator
OUT	status	return status

```
      do iPE=1,nPEs-1
         do inode=1,NodePerProc(iPE)
      call MPI_SENDRECV(Node_local(1,inode,iPE),2,
     >                      MPI_DOUBLE_PRECISION,0,   1
     >                  Node(1,inode),2,
     >                      MPI_DOUBLE_PRECISION,iPE,1
     >                  MPI_COMM_WORLD,status,ierr)
          enddo
      enddo
```



```
c    --------- fetch from EAST: [xv(i,j,k) = x(i+distx,j,k)]
      if (NEWS27(1) .ne. 999) then
      call MPI_SENDRECV(xv(nanrx+1,1,1),1,type_fixed_x,NEWS27(1),
     >                  xv(nanrx,1,1),  1,type_fixed_x,NEWS27(1),
     >                   MPI_COMM_WORLD,status,ierr)
      endif
c    ---------- fetch from NORTH: [xv(i,j,k) = x(i,j+disty,k)]
      if (NEWS27(3) .ne. 999) then
      call MPI_SENDRECV(xv(1,nanry+1,1),1,type_fixed_y,NEWS27(3),
     >                  xv(1,nanry,1),  1,type_fixed_y,NEWS27(3),
     >                   MPI_COMM_WORLD,status,ierr)
      endif
```

```
      do ib = 1,Cross_node_no
         iblock=Cross_Node_list(ib)
         do ip=1,Node_PE_local(0,iblock)
            iPE=Node_PE_local(ip,iblock)
            call MPI_SENDRECV(K(1,1,iblock),4,
     >                            MPI_DOUBLE_PRECISION,iPE,1,
     >                            K_tmp(1,1,iPE,ib),4,
     >                            MPI_DOUBLE_PRECISION,iPE,1,
     >                            MPI_COMM_WORLD,stat,ierr)
         enddo
      enddo
```


Nonblocking SEND/RECEIVE

MPI_ISEND(buf, count, datatype, dest, tag, comm, status, request)

MPI_IRECV(buf, count, datatype, source, tag, comm, status, request)

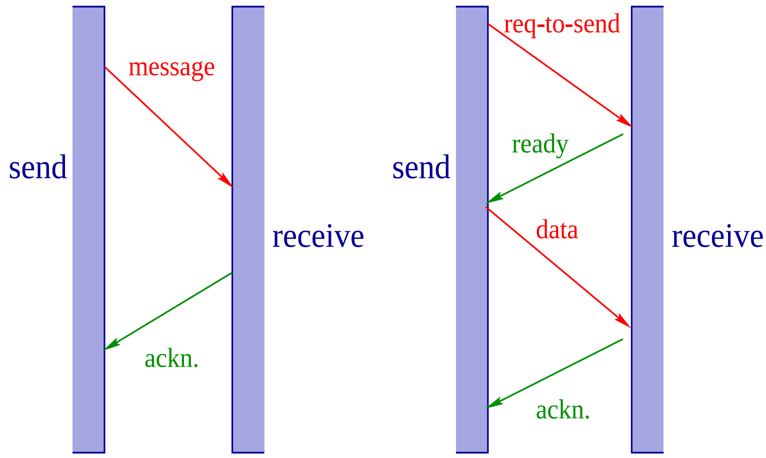
OUT request request handle

These calls allocate a request object and return a handle to it in `request` which is used to query the status of the communication or wait for completion.

Completion operations

MPI_WAIT(request,status)	returns when the operation identified by `request` is completed
MPI_TEST(request,flag,status)	returns `flag=true` if the operation identified by `request` is completed or `flag=false` otherwise


```
...
!start communication
call MPI_ISEND(B(1,1),n,MPI_REAL,left, tag,comm,req(1),ierr)
call MPI_ISEND(B(1,m),n,MPI_REAL,right,tag,comm,req(2),ierr)
call MPI_IRECV(A(1,1),n,MPI_REAL,left, tag,comm,req(3),ierr)
call MPI_IRECV(A(1,m),n,MPI_REAL,right,tag,comm,req(4),ierr)
! do some computational work
...
! Complete communication
do i=1,4
   call MPI_WAIT(req(i),status(1,i),ierr)
end
```



Short protocol Long protocol

Out of order communications with nonblocking messages

```
call MPI_COMM_RANK(comm,rank,ierr)
if (rank .eq. 0) then
   call MPI_SEND(sendbuf1, count, MPI_REAL,1,1,comm,ie
   call MPI_SEND(sendbuf2, count, MPI_REAL,1,2,comm,ie
else   ! ranl = 1
   call MPI_IRECV(recvbuf2, count, MPI_REAL,0,2,comm,r
   call MPI_IRECV(recvbuf2, count, MPI_REAL,0,1,comm,r
   call MPI_WAIT(req1, status, ierr)
   call MPI_WAIT(req2, status, ierr)
endif
```

If both blocking SEND and RECV were used, the first message has to be copied and buffered before the second SEND can be proceeded.

Persistent communication requests are associated with nonblocking send and receive operations.

Situation: communication with the same argument list is repeatedly executed within the inner loop of a parallel computation.

(1) MPI persistent communications can be used to reduce communications overhead in programs which repeatedly call the same point-to-point message passing routines with the same arguments. They minimize the software overhead associated with redundant message setup.

(2) An example of an application which might benefit from persistent communications would be an iterative, data decomposition algorithm that exchanges border elements with its neighbors. The message size, location, tag, communicator and data type remain the same each iteration.

Step 1: Create persistent requests

The desired routine is called to setup buffer location(s) which will be sent/received. The five available routines are:

`MPI_Recv_init`	Creates a persistent receive request
`MPI_Bsend_init`	Creates a persistent buffered send request
`MPI_Rsend_init`	Creates a persistent ready send request
`MPI_Send_init`	Creates a persistent standard send request
`MPI_Rsend_init`	Creates a persistent ready send request
`MPI_Ssend_init`	Creates a persistent synchronous send request

Step 2: Start communication transmission

Data transmission is begun by calling either of the `MPI_Start` routines.

`MPI_Start`	Activates a persistent request operation
`MPI_Startall`	Activates a collection of persistent request operations

Step 3: Wait for communication completion

Because persistent operations are non-blocking, the appropriate `MPI_Wait` or `MPI_Test` routine must be used to insure their completion.

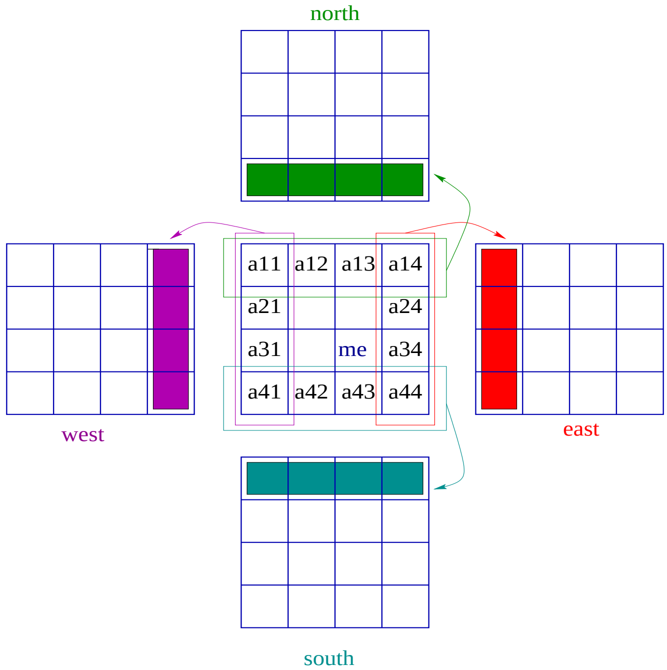
Step 4: Deallocate persistent request objects

When there is no longer a need for persistent communications, the programmer should explicitly free the persistent request objects by using the `MPI_Request_free()` routine.

MPI_SEND_INIT(buf, count, type, dest, tag, comm, request)
MPI_RECV_INIT(buf, count, type, source, tag, comm, request)
MPI_START(request)
MPI_STARTALL(count,array-of-requests)
MPI_REQUEST_FREE(request)

IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	type	datatype of each entry
IN	dest	rank of destination
IN	source	rank of source
IN	tag	tag
IN	comm	communicator
OUT	request	request handle

```
for (i=0; i<size-1; i++) {                                /* Setup */
    MPI_SEND_INIT(sendbuf,counts[(rank+1)%size],
                  type, right, i, MPI_COMM_WORLD, &request[2*i] );
    MPI_RECV_INIT(recvbuf,counts[(rank+i-1+size)%size],
                  type, left, i, MPI_COMM_WORLD, &request[2*i+1] );
  }
while (!done)                                  /* Run pipeline */
      <copy local data into sendbuf>
      for (i=0;i<size; i++) {
          MPI_STATUS stat[2];
          if (i != size - 1)
             MPI_STARTALL(2,&request[2*i]);
          < compute using sendbuf>
          if (i != size - 1)
             MPI_WAITALL(2, &request[2*i], stat);
          < copy recvbuf into sendbuf>
          }
    <compute new data>
  }
for (i=0; i<2*(size-1); i++) {                  /* Free requests */
    MPI_REQUEST_FREE(&request[i] );
   }
```

```
for i=1:n
    do something
    receive from P_src, tag=1
    do something else
    send to P_dest,     tag=2
end
```

Use constant tag within a loop: if for one processor it takes longer to finish the current iteration i, it may end up with receiving the data from iteration $i+1$ for processor P_{src}.

The same may happen if the same tag is used in two parts of the code which are not separated explicitly by a barrier.

Result: a nondeterministic code which may finish correctly from time to time, give wrong results some of the time, and other time just crash.

Collective communications

◇ Transmission of data and synchronization among all processes in a group.

Restrictions:

◇ amount of data send must match exactly that of data received;
◇ collective functions only in blocking version;
◇ No `tag`, thus the calls are matched according to the order of execution;
◇ only 'normal' mode, i.e., a collective function returns as soon as its participation in the overall communication is completed.

- Barrier synchronization across all processes;
- Global communication functions
  - Broadcast from one to all processes;
  - Gather data from all processes to one process;
  - Scatter data from one to all processes;
  - Scatter/Gather data from all processes to all processes;
- Global reduction operation such as *sum, max, min, etc.*

All the listed functions (excepts *broadcast*) can be found in two variants:

(a) simple - where all items are messages of the same size;

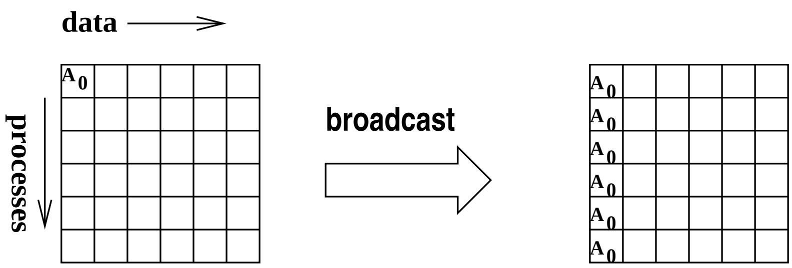
(b) vector - where each item may be of a different size.

MPI_BARRIER(comm, ierr)

MPI_BARRIER blocks the caller until all processes have called it. The call returns at any process only after all processes have entered the call.

BROADCAST

MPI_BCAST(buffer, count, datatype, root, comm)



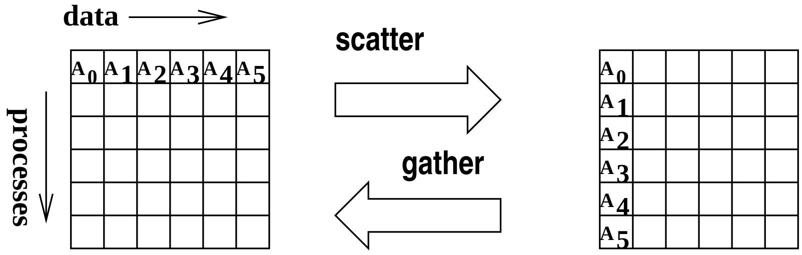

```
 call MPI_BCAST(Discoef,2*ndisco,MPI_DOUBLE_PRECISION,0,
>                MPI_COMM_WORLD,status,ierr)
```

MPI_BCAST broadcasts a message from the process with rank `root` to all processes in the group. The argument `root` must have identical value on all processes and `comm` must represent the same communication domain. On return the contents pf the `root`'s communication buffer is copied to all processes.

```
MPI_COMM comm;
int array[100];
int root=0;
....
      call MPI_BCAST(array, 100, MPI_INT, root, comm);
....
      call MPI_BCAST(Discoef,2*ndisco,MPI_DOUBLE_PRECISION,0,
     >               MPI_COMM_WORLD,status,ierr)
```


GATHER

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)



Gather 100 integers from every proc to root.
(i) Everybody allocates space for the `receive` buffer.

```
MPI_COMM comm;
int gsize, sendarray[100];
int root=0, *rbuf;
....
MPI_COMM_SIZE(comm,&gsize);
rbuf = (int*)malloc(gsize*100*sizeof(int));
MPI_GATHER(sendarray,100,MPI_INT,rbuf,100,MPI_INT,root,comm)
....
```


Gather 100 integers from every proc to root.
(ii) Only `root` allocates space for the `receive` buffer.

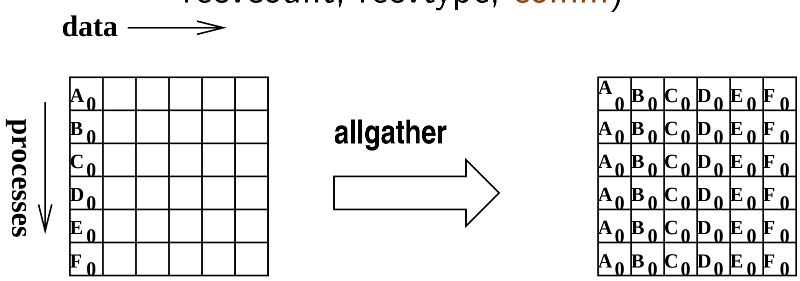
```
MPI_COMM comm;
int gsize, sendarray[100];
int root=0, myrank, *rbuf;
....
MPI_COMM_RANK(comm,myrank);
if ( myrank == root ){
    MPI_COMM_SIZE(comm,&gsize);
    rbuf = (int*)malloc(gsize*100*sizeof(int));
    }
MPI_GATHER(sendarray,100,MPI_INT,rbuf,100,MPI_INT,root,comm)
....
```

Gather 100 integers from every proc to root.
(iii) Use derived datatype.

```
MPI_COMM comm;
int gsize, sendarray[100];
int root, *rbuf;
MPI_DATATYPE rtype;
....
MPI_COMM_SIZE(com,&gsize);
MPI_TYPE_CONTIGUOUS(100, MPI_INT, &rtype);
rbuf = (int*)malloc(gsize*100*sizeof(int));
MPI_GATHER(sendarray, 100, MPI_INT, rbuf, 100, rtype,root,comm)
....
```

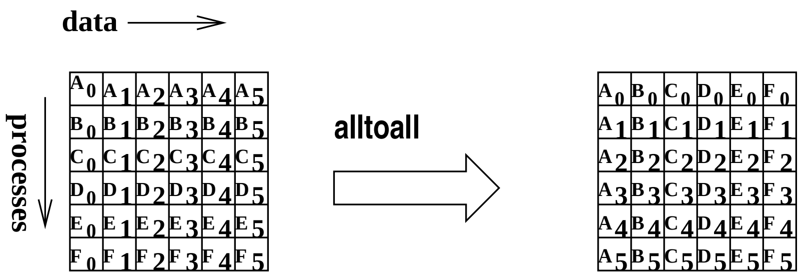

All-GATHER

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)




```
MPI_Comm comm;
int gsize, sendarray[100];
int *rbuf;
...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_ALLGATHER( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

ALL-TO-ALL communication



MPI_ALLTOALL(sendbuf, sendcount, sendtype,
recvbuf,recvcount, recvtype, comm)

REDUCE and ALL REDUCE

Name (operation)	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_LOR	logical or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

```
c  dot_product: compute a scalar product
      subroutine dot_product(global,x,y,n)
      implicit none
      include "mpif.h"
      integer n,i,ierr
      double precision global,x(n),y(n)
      double precision tmp,local
      local  = 0.0d0
      global = 0.0d0
      do i=1,n
         local = local + x(i)*y(i)
      enddo
      call MPI_ALLREDUCE(local,tmp,1,MPI_DOUBLE_PRECISION,
     >                   MPI_SUM, MPI_COMM_WORLD, ierr)
      global = tmp
      return
      end
```



```
switch(rank) {
    case 0:
           MPI_BCAST(buf1,count,type,0,comm);
           MPI_BCAST(buf2,count,type,1,comm);
           break;
    case 1:
           MPI_BCAST(buf2,count,type,1,comm);
           MPI_BCAST(buf1,count,type,0,comm);
           break;
}
```

Assume that comm={0,1}.
The calls do not specify the same root.
!!! Collective communications must be executed in the same order at all members of the communication group.

```
switch(rank) {
   case 0:
        MPI_BCAST(buf1,count,type,0,comm0);
        MPI_BCAST(buf2,count,type,2,comm2);
        break;
   case 1:
        MPI_BCAST(buf1,count,type,1,comm1);
        MPI_BCAST(buf2,count,type,0,comm0);
        break;
   case 2:
        MPI_BCAST(buf1,count,type,2,comm2);
        MPI_BCAST(buf2,count,type,1,comm1);
        break;
}
```

Say, comm0={0,1}, comm1={1,2} and comm2={2,0}.
If the broadcast is a synchronizing operation, the code will deadlock.
Reason: there is a cyclic dependency:

BCAST in comm2 ⟶ BCAST in comm0
BCAST in comm0 ⟶ BCAST in comm1
BCAST in comm1 ⟶ BCAST in comm2

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS DCAMM
Erroneous examples

```
switch(rank) {
   case 0:
        MPI_BCAST(buf1,count,type,0,comm);
        MPI_SEND(buf2,count,type,1,tag,comm);
        break;
   case 1:
        MPI_RECV(buf2,count,type,0,tag,comm);
        MPI_BCAST(buf1,count,type,0,comm);
        break;
}
```

The program may deadlock because MPI_BCAST on P0 may block till PE1 executes the matching MPI_BCAST. However, PE1 waits to receive data and will never execute BCAST.

```
switch(rank) {
   case 0:
        MPI_BCAST(buf1,count,type,0,comm);
        MPI_SEND(buf2,count,type,1,tag,comm);
        break;
   case 1:
        MPI_RECV(buf2,count,type,MPI_ANY_SOURCE,tag,comm);
        MPI_BCAST(buf1,count,type,0,comm);
        MPI_RECV(buf2,count,type,MPI_ANY_SOURCE,tag,comm);
        break;
   case 2:
        MPI_SEND(buf2,count,type,1,tag,comm);
        MPI_BCAST(buf1,count,type,0,comm);
        break;
}
```

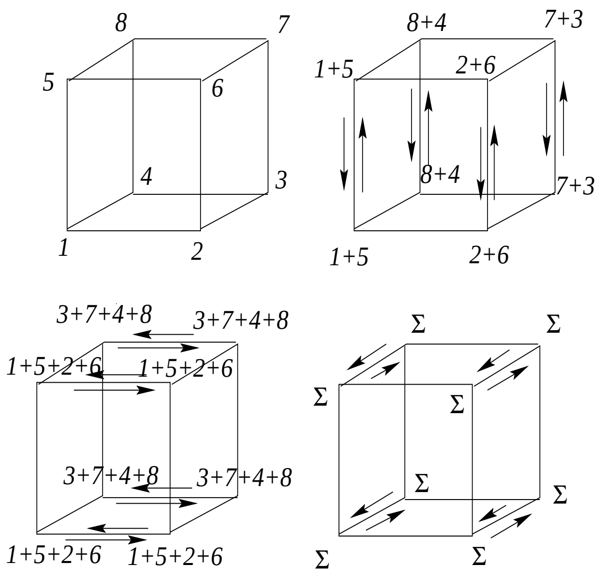

A correct but nondeterministic code. There are two possible scenarios:

	Processes	
0	1	2
	Scenario 1	
	RECV	⟵ SEND
BCAST	BCAST	BCAST
SEND ⟶	RECV	
	Scenario 2	
BCAST		
SEND ⟶	RECV	
	BCAST	
	RECV	⟵ SEND
		BCAST

Timing MPI Programs

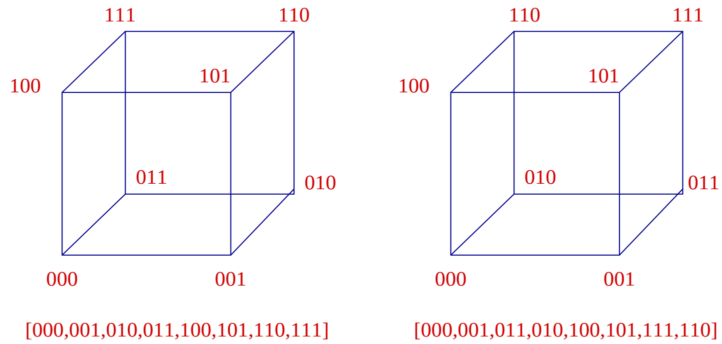
```
MPI_WTIME( )
DOUBLE PRECISION MPI_WTIME( )
```

MPI_WTIME returns a floating-point number of seconds representing *elapsed wall-clock* time since some arbitrary point of time in the past. This point is guaranteed not to change during the lifetime of the process. Thus, a time interval can be measured by calling this routine at the beginning and end of the program segment has to be measured and subtracting the values returned.



$\Sigma = 1+2+3+4+5+6+7+8$

Computing a scalar product on a 3-D hypercube



(a) Standard numbering (b) Gray code ordering

Theorem *Any* $m_1 \times m_2 \ldots \times m_n$ *mesh in the* n*-dimensional space* R^n*, where* $m_i = 2^{r_i}$ *can be mapped onto a* d*-cube where* $d = r_1 + r_2 + \cdots r_n$*, with the proximity property preserved. The mapping of the grid points is the cross product* $G_1 \times G_2 \times \cdots \times G_n$ *where* G_i*,* $i = 1, \ldots n$ *is any one-dimensional Gray-code mapping of the* m_i *points in the* i*th coordinate direction.*