Message Passing Interface (MPI)

Zhengxiong Hou

Fall, 2022

MPI Routines

- (1) Environment Management Routines
- (2) Point to Point Communication Routines
- *(3) Collective Communication Routines*
- (4) Derived Data Types
- (5) Process Group and Communicator Management Routines
- (6) Virtual Topologies

Collective Communication Routines

- **Types of Collective Operations:**
  - **Data Movement/communication** - broadcast, scatter/gather, all to all.
    （1-N; N-1; N-N)
  - **Collective Computation (reductions)** - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.
  - **Synchronization** - processes wait until all members of the group have reached the synchronization point.

MPI_Bcast
MPI_Gather
MPI_Gatherv
MPI_Allgather
MPI_Allgatherv
MPI_Scatter
MPI_Scatterv
MPI_Alltoall
MPI_Alltoallv

MPI_Reduce
MPI_Allreduce
MPI_Reduce_scatter
MPI_Scan

MPI_Barrier

Scope:

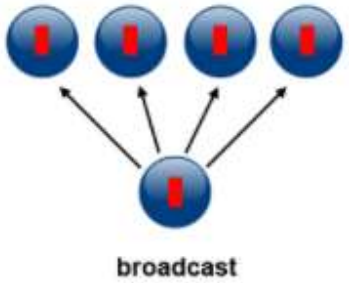
- Collective communication routines must involve **all** processes within the scope of a communicator.
  - All processes are by default, members in the communicator MPI_COMM_WORLD.
  - Additional communicators can be defined by the programmer. See the Group and Communicator Management Routines section for details.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

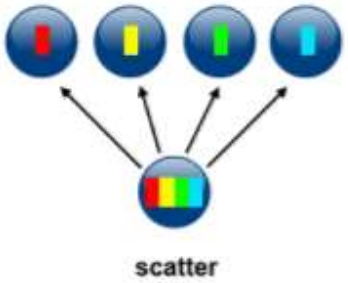
Programming Considerations and Restrictions:

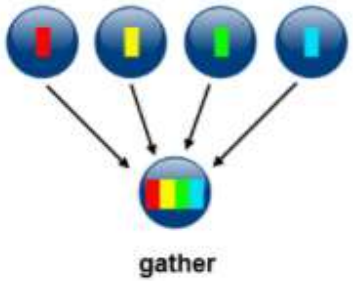
- Collective communication routines do not take message tag arguments
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators
- Can only be used with MPI predefined datatypes - not with MPI Derived Data Types.

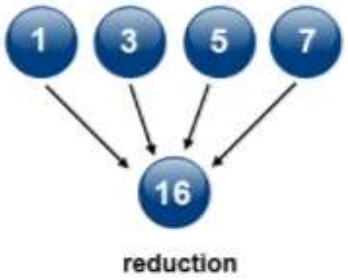
Collective Communication Routines

Collective Communication Routines			
MPI_Allgather	MPI_Allgatherv	MPI_Allreduce	MPI_Alltoall
MPI_Alltoallv	MPI_Barrier	MPI_Bcast	MPI_Gather
MPI_Gatherv	MPI_Op_create	MPI_Op_free	MPI_Reduce
MPI_Reduce_scatter	MPI_Scan	MPI_Scatter	MPI_Scatterv







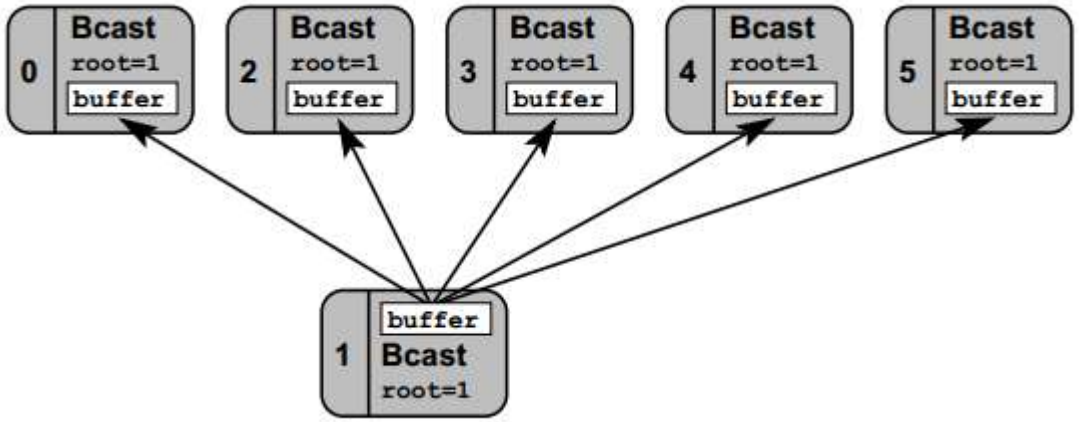


MPI_Bcast

```
MPI_Bcast (&buffer,count,datatype,root,comm)
MPI_BCAST (buffer,count,datatype,root,comm,ierr)
```

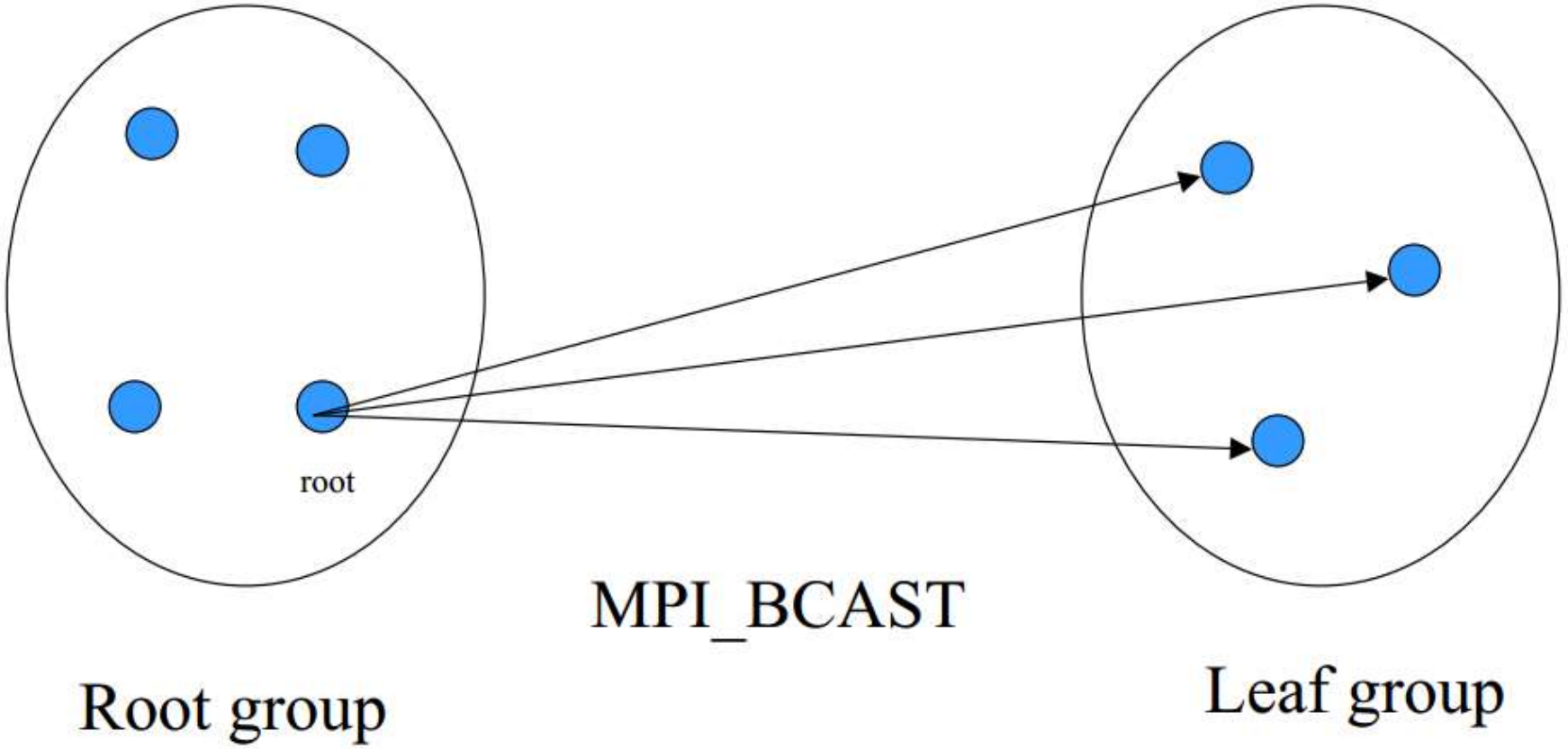
- Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

```
<type> buf(*)
integer :: count, datatype, root, comm, ierror
call MPI_Bcast(buffer,     ! send/receive buffer
               count,      ! message length
               datatype,   ! MPI data type
               root,       ! rank of root process
               comm,       ! communicator
               ierror)     ! return value
```



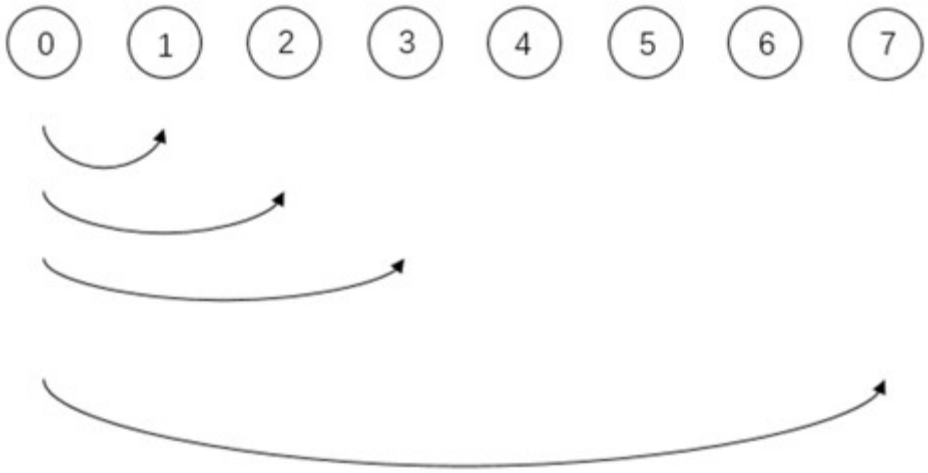

```
int MPI_Bcast(
        void*         data_p          /* in/out */,
        int           count           /* in     */,
        MPI_Datatype  datatype        /* in     */,
        int           source_proc     /* in     */,
        MPI_Comm      comm            /* in     */);
```

MPI_Bcast



Implementation Algorithms of MPI_Bcast

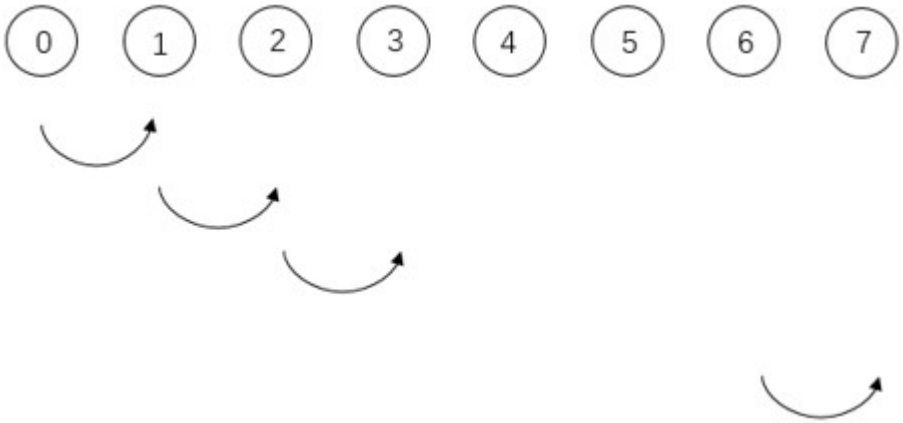
- Flat tree broadcast algorithm



  - Performance model: $(p-1) \times \alpha + m \times \beta$

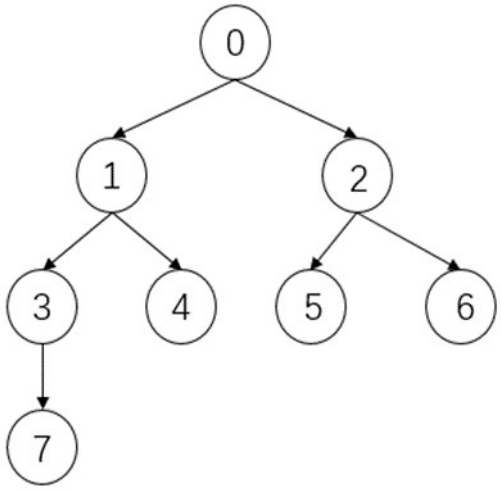
p: number of MPI processes
α: latency
m: $size\ of\ data$
β: 1/bandwidth

- **Linear tree broadcast algorithm/chain algorithm**



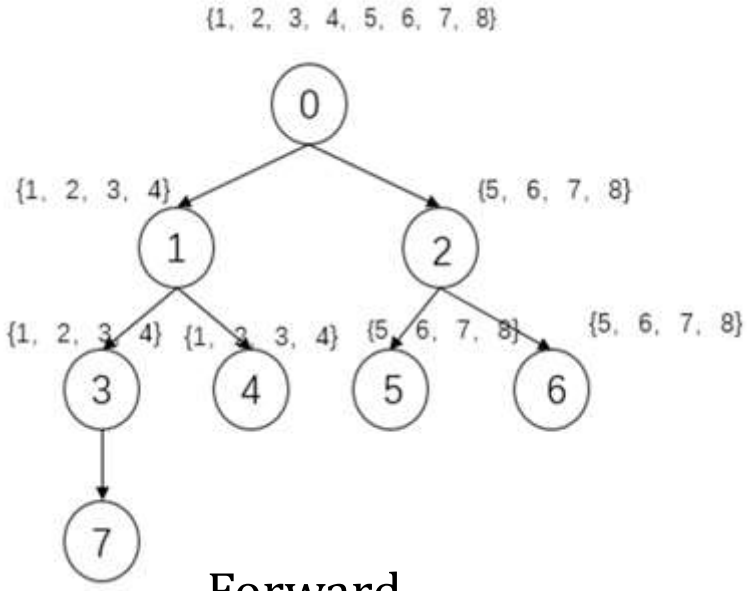
- Performance model: $(p-1) \times \alpha + m \times \beta$

- **Binary tree broadcast algorithm**

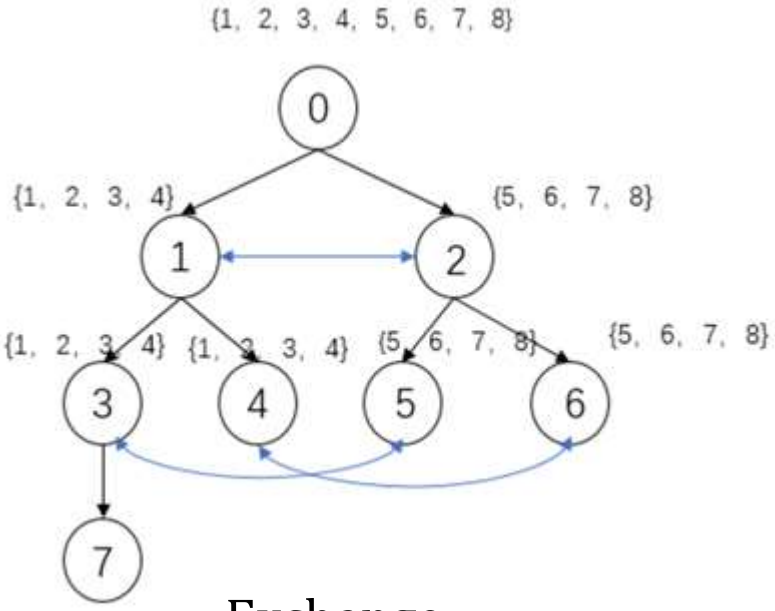


- Performance model: $2 \times (log_2(\mathrm{p}+1)-1) \times (\alpha + \mathrm{m} \times \beta)$

- **Split-binary tree broadcast algorithm**



Forward

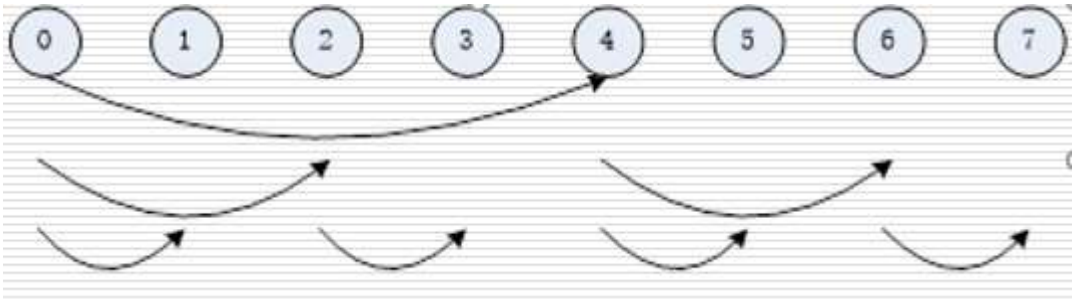


Exchange

- Performance model:

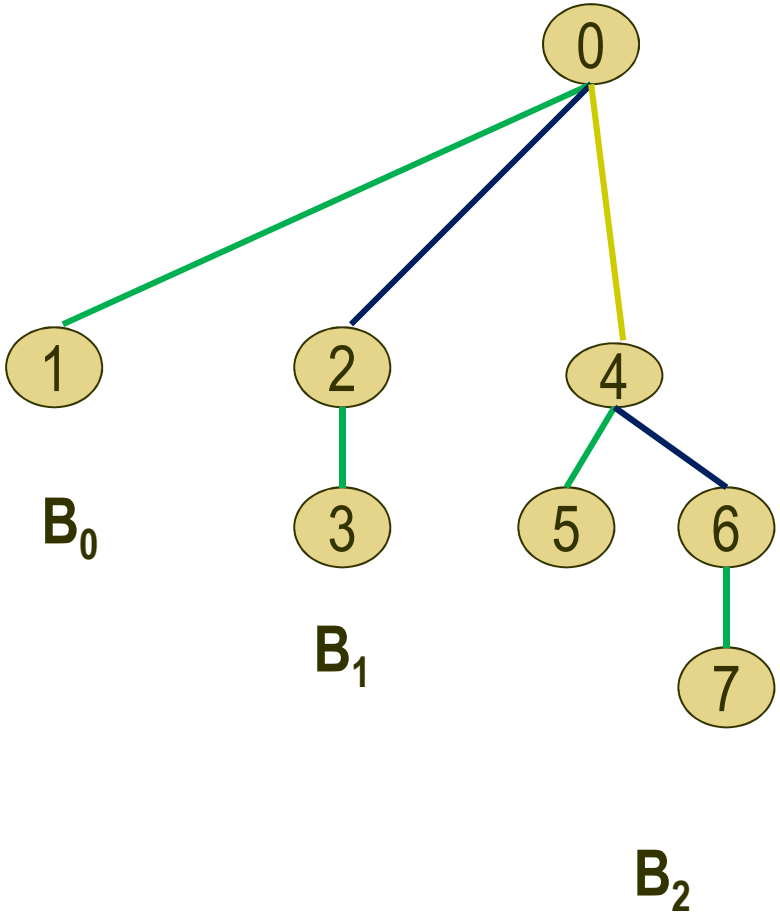
$$2 \times (log_2(\mathrm{p}+1) - 2) \times (\alpha + \mathrm{m} \times \beta) + \frac{m}{2}\beta$$

- **Binomial tree broadcast algorithm**



Performance model: $log_2(\mathrm{p}) \times (\alpha + \mathrm{m} \times \beta)$

- **Binomial tree broadcast algorithm for 8 processes**



Step1:0->4

Step2:0->2;4->6

Step3:0->1;2->3;4->5;6->7

```
1   void Get_input(
2         int       my_rank   /* in  */,
3         int       comm_sz   /* in  */,
4         double*   a_p       /* out */,
5         double*   b_p       /* out */,
6         int*      n_p       /* out */) {
7
8      if (my_rank == 0) {
9         printf("Enter a, b, and n\n");
10        scanf("%lf %lf %d", a_p, b_p, n_p);
11     }
12     MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13     MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14     MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15  }  /* Get_input */
```

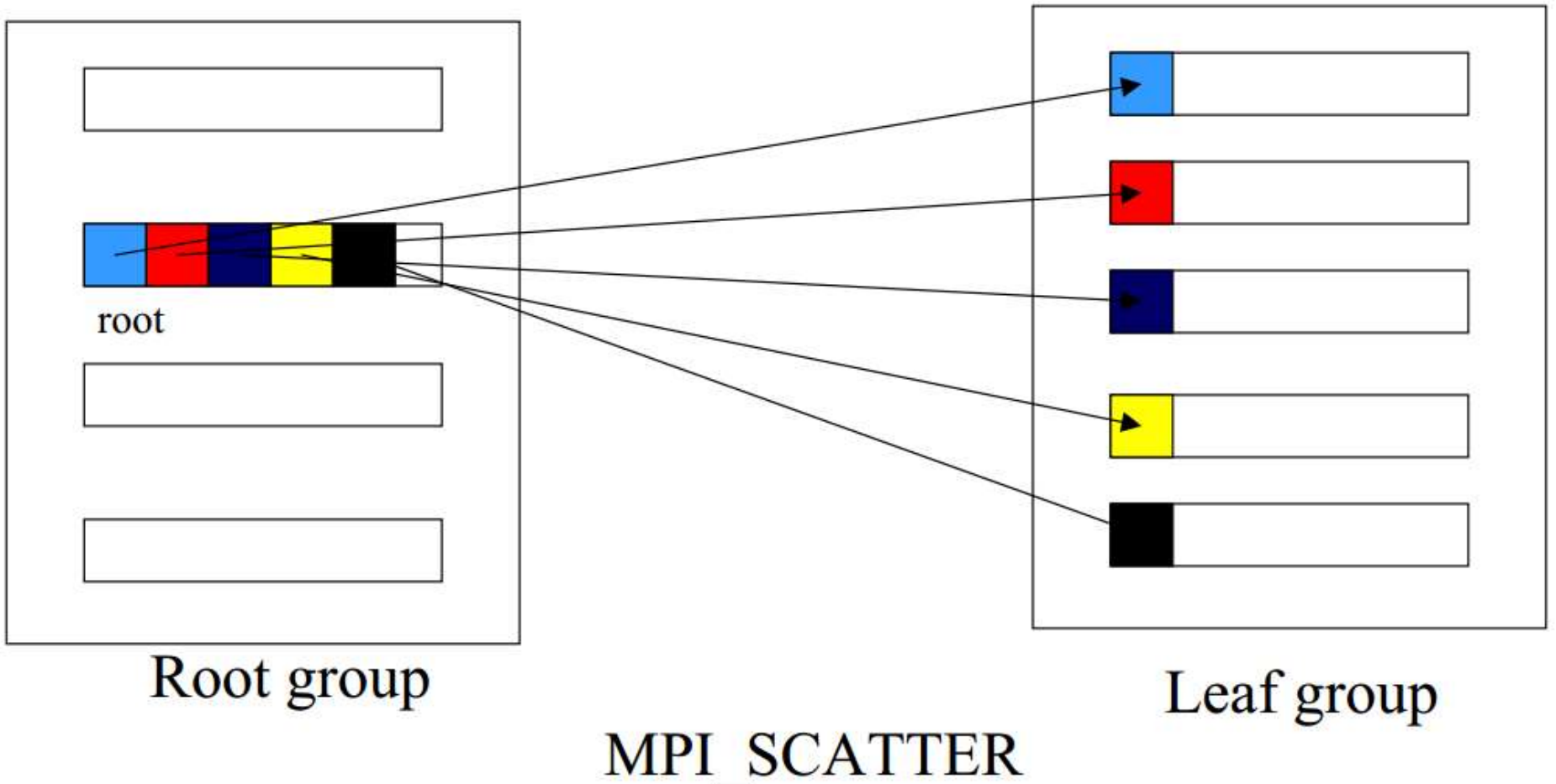
- A version of Get input that uses MPI Bcast

MPI_Scatter

- Data movement operation. Distributes distinct messages from a single source task to each task in the group.

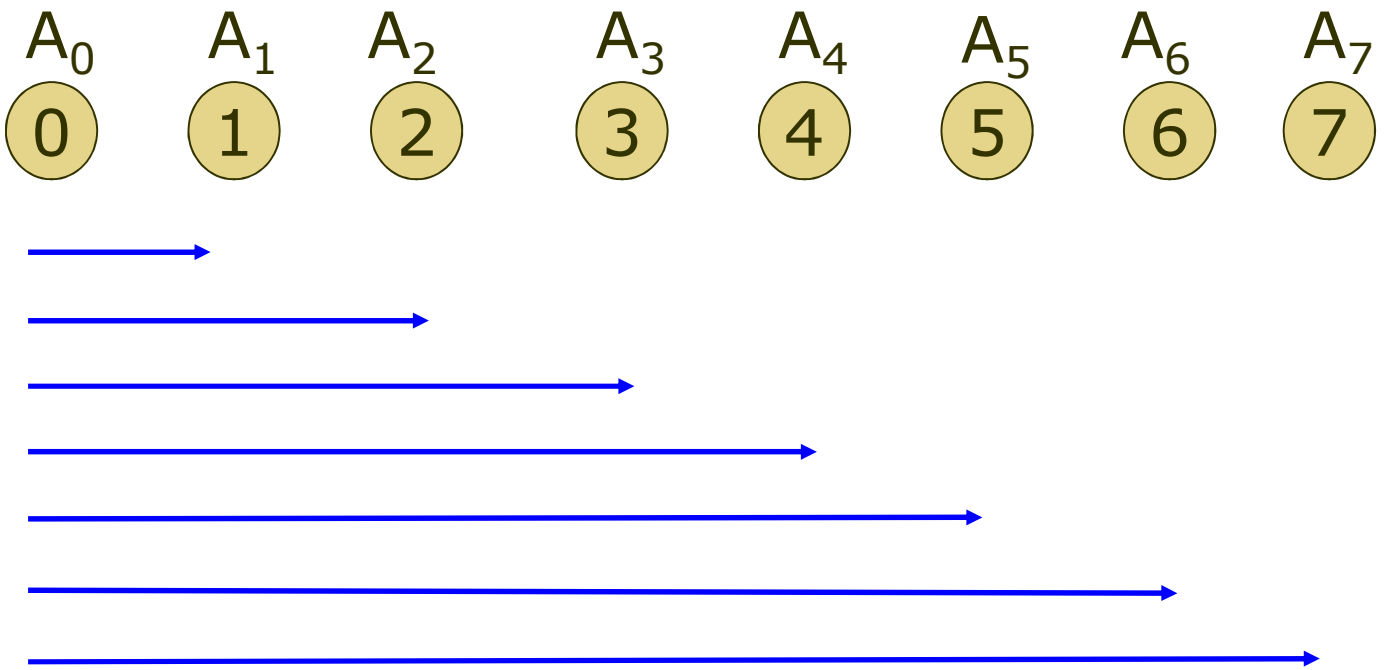
```
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,
       recvcnt,recvtype,root,comm)
MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf,
       recvcnt,recvtype,root,comm,ierr)
```

```
int MPI_Scatter(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            src_proc     /* in  */,
      MPI_Comm       comm         /* in  */);
```

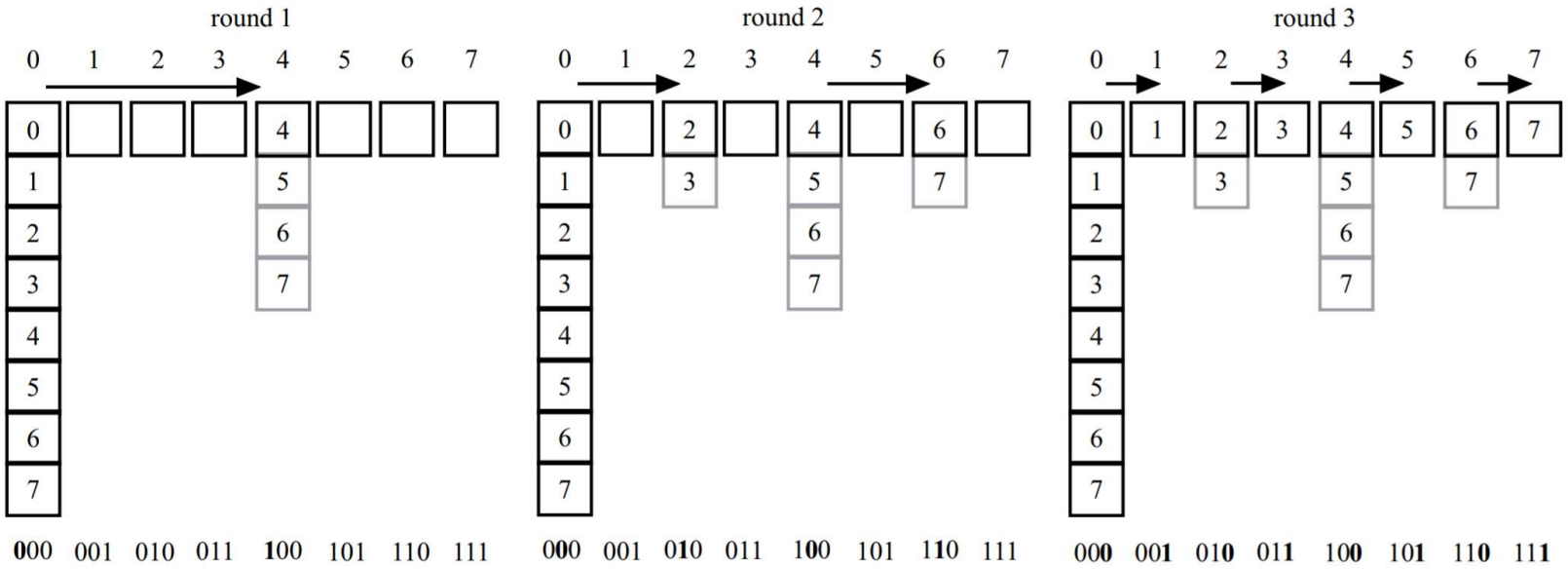


MPI_SCATTER

- **Linear algorithm for 8 processes**



- **Binomial tree algorithm for 8 processes**



Performance model: $[\log p]\alpha + \frac{p-1}{p} n\beta$

```
1  void Read_vector(
2        double    local_a[]   /* out */,
3        int       local_n     /* in  */,
4        int       n           /* in  */,
5        char      vec_name[]  /* in  */,
6        int       my_rank     /* in  */,
7        MPI_Comm  comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 }  /* Read_vector */
```

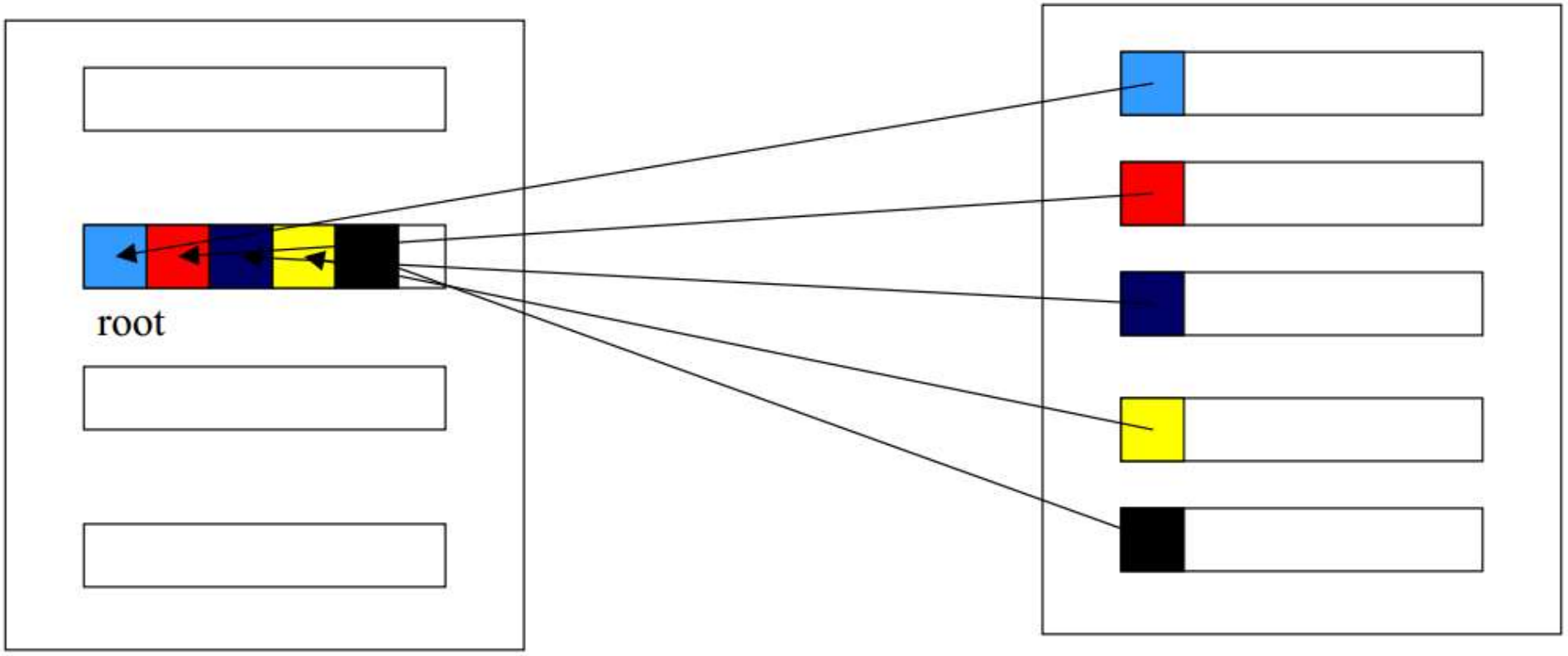
- A function for reading and distributing a vector

MPI_Gather

- Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

```
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,
      recvcount,recvtype,root,comm)
MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf,
      recvcount,recvtype,root,comm,ierr)
```

```
int MPI_Gather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            dest_proc    /* in  */,
      MPI_Comm       comm         /* in  */);
```

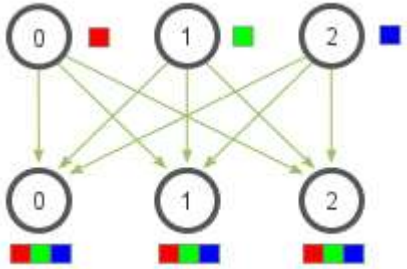


MPI_GATHER

MPI_Allgather

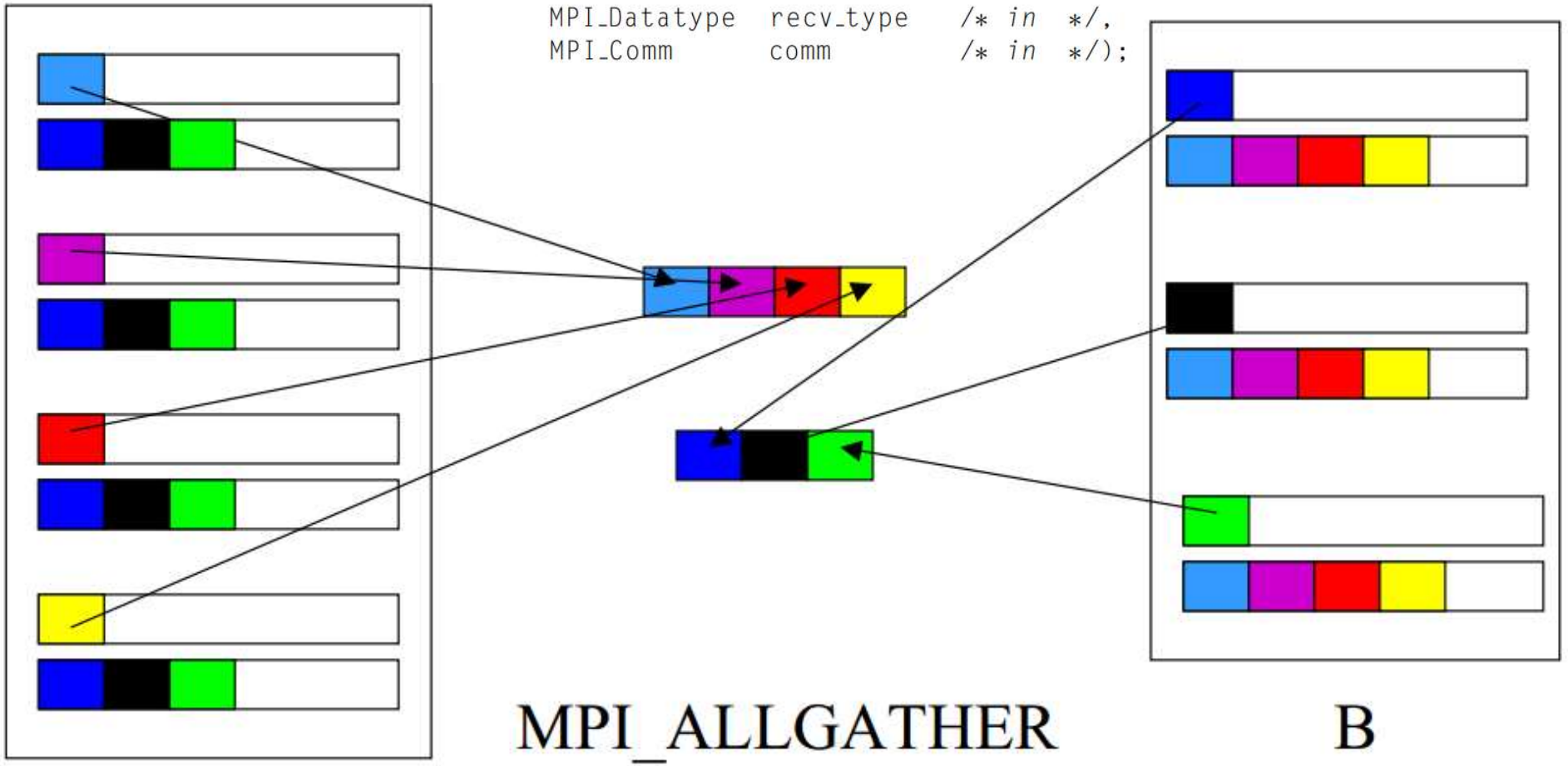
```
MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf,
       recvcount,recvtype,comm)
MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf,
       recvcount,recvtype,comm,info)
```

- Data movement operation. Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.



- All arguments are meaningful at every process
- Data from *sbuf* at all processes in group A is concatenated in rank order and the result is stored at *rbuf* of every process in group B and vice-versa
- Send arguments in A must be consistent with receive arguments in B, and viceversa

```
int MPI_Allgather(
      void*          send_buf_p   /* in */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```



MPI_ALLGATHER

A

B

- An MPI matrix-vector multiplication function

```
1  void Mat_vect_mult(
2        double  A[]  /* in  */,
3        double  x[]  /* in  */,
4        double  y[]  /* out */,
5        int     m    /* in  */,
6        int     n    /* in  */) {
7     int i, j;
8
9     for (i = 0; i < m; i++) {
10       y[i] = 0.0;
11       for (j = 0; j < n; j++)
12          y[i] += A[i*n+j]*x[j];
13    }
14 }  /* Mat_vect_mult */
```

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

MPI_Reduce

```
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```

- Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.

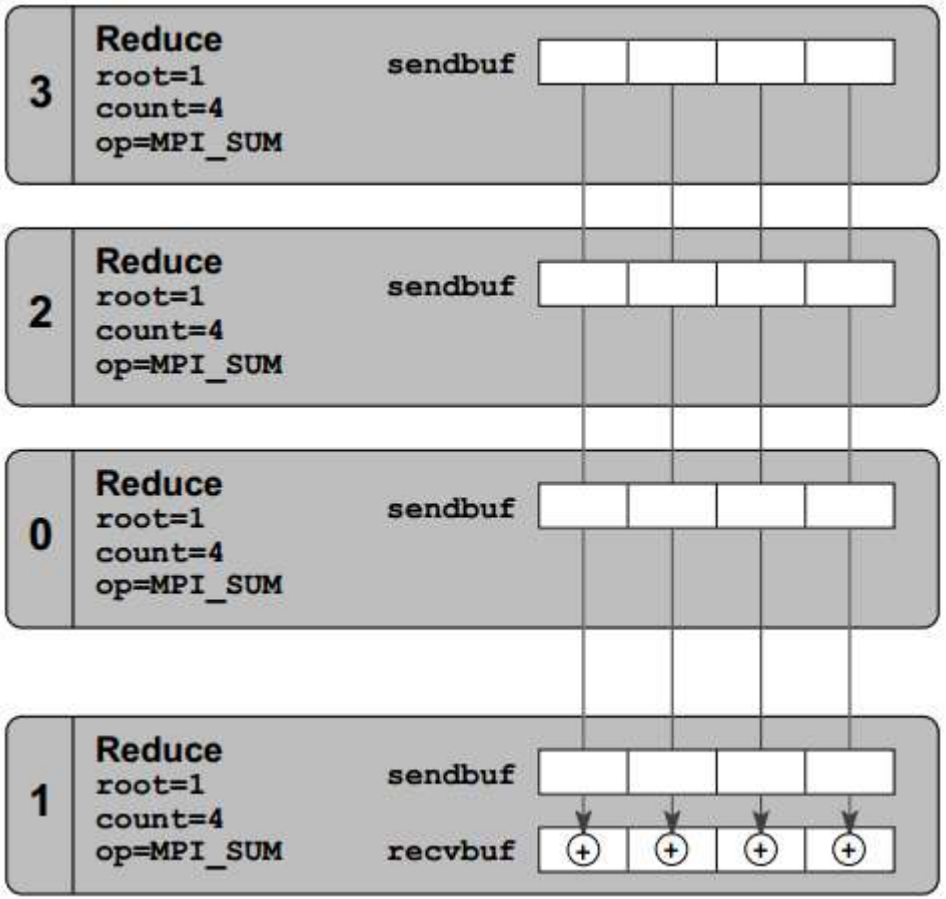
```
<type> sendbuf(*), recvbuf(*)
integer :: count, datatype, op, root, comm, ierror
call MPI_Reduce(sendbuf,     ! send buffer
                recvbuf,     ! receive buffer
                count,       ! number of elements
                datatype,    ! MPI data type
                op,          ! MPI reduction operator
                root,        ! root rank
                comm,        ! communicator
                ierror)      ! return value
```

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

- The predefined MPI reduction operations appear below. Users can also define their own reduction functions by using the MPI_Op_create routine.

MPI Reduction Operation		C Data Types	Fortran Data Type
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex
MPI_LAND	logical AND	integer	logical
MPI_BAND	bit-wise AND	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	logical OR	integer	logical
MPI_BOR	bit-wise OR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer	logical
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double	real, complex,double precision
MPI_MINLOC	min value and location	float, double and long double	real, complex, double precision

- A reduction on an array of length count (a sum in this example) is performed by MPI_Reduce().
- Every process must provide a send buffer. The receive buffer argument is only used on the root process.
- The local copy on root can be prevented by specifying MPI_IN_PLACE instead of a send buffer address.



MPI_Allreduce

- Collective computation operation + data movement. Applies a reduction operation and places the result in all tasks in the group. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.

```
MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)
MPI_ALLREDUCE (sendbuf,recvbuf,count,datatype,op,comm,ierr)
```

```
int MPI_Allreduce(
        void*         input_data_p   /* in  */,
        void*         output_data_p  /* out */,
        int           count          /* in  */,
        MPI_Datatype  datatype       /* in  */,
        MPI_Op        operator       /* in  */,
        MPI_Comm      comm           /* in  */);
```

MPI_Reduce_scatter

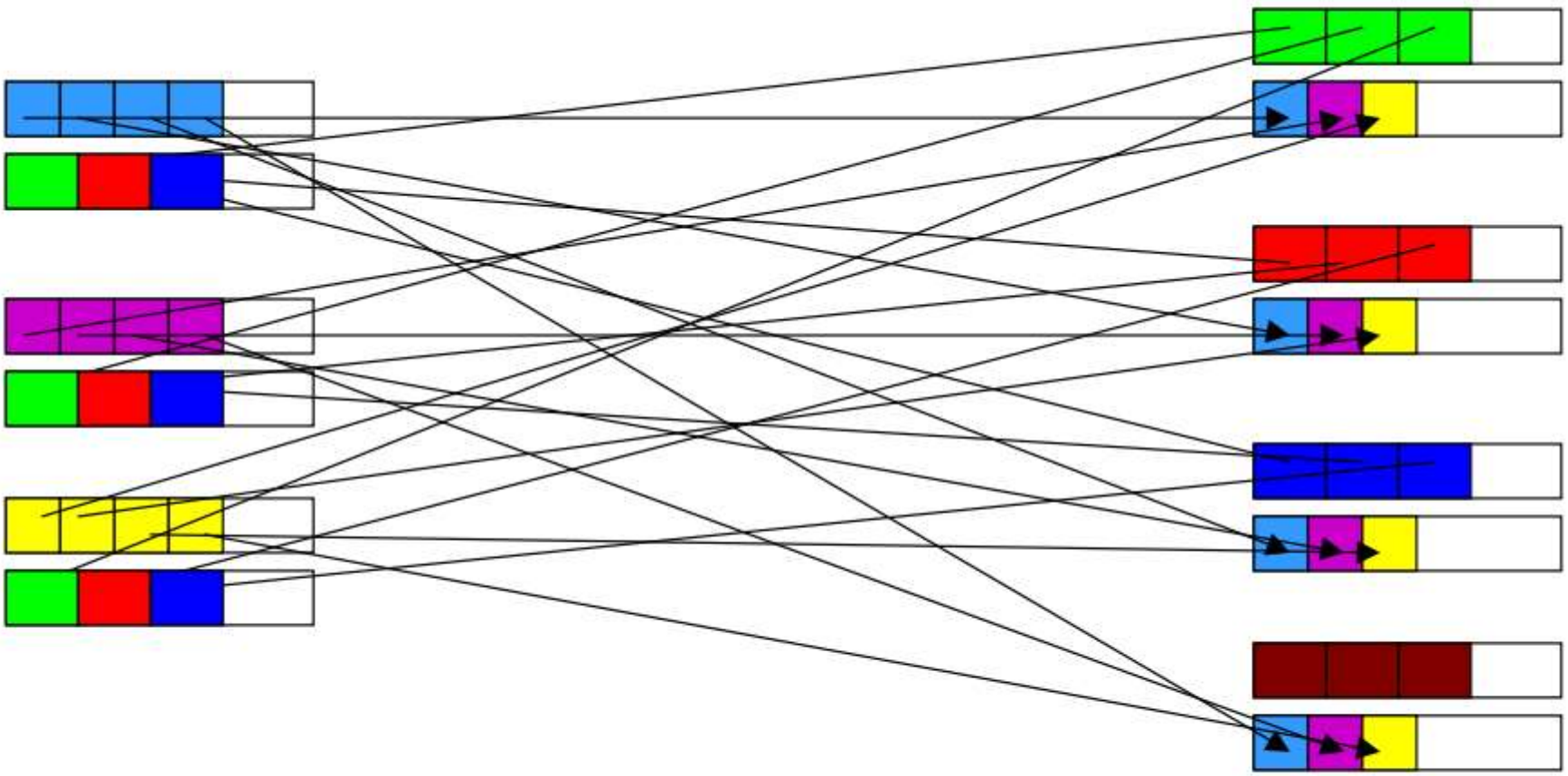
- Collective computation operation + data movement. First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks. This is equivalent to an MPI_Reduce followed by an MPI_Scatter operation

```
MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype,
        op,comm)
MPI_REDUCE_SCATTER (sendbuf,recvbuf,recvcount,datatype,
        op,comm,ierr)
```

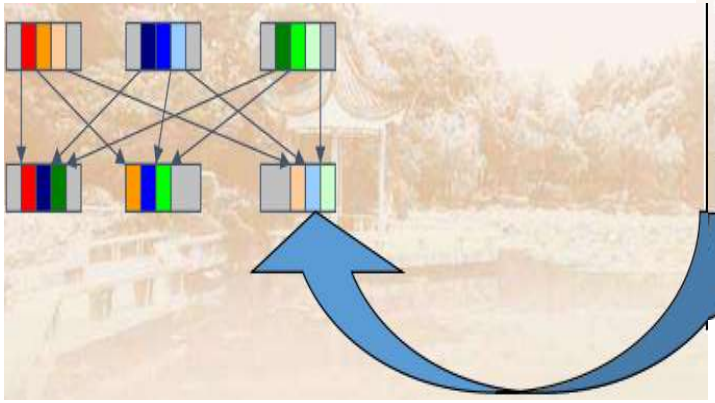
MPI_Alltoall

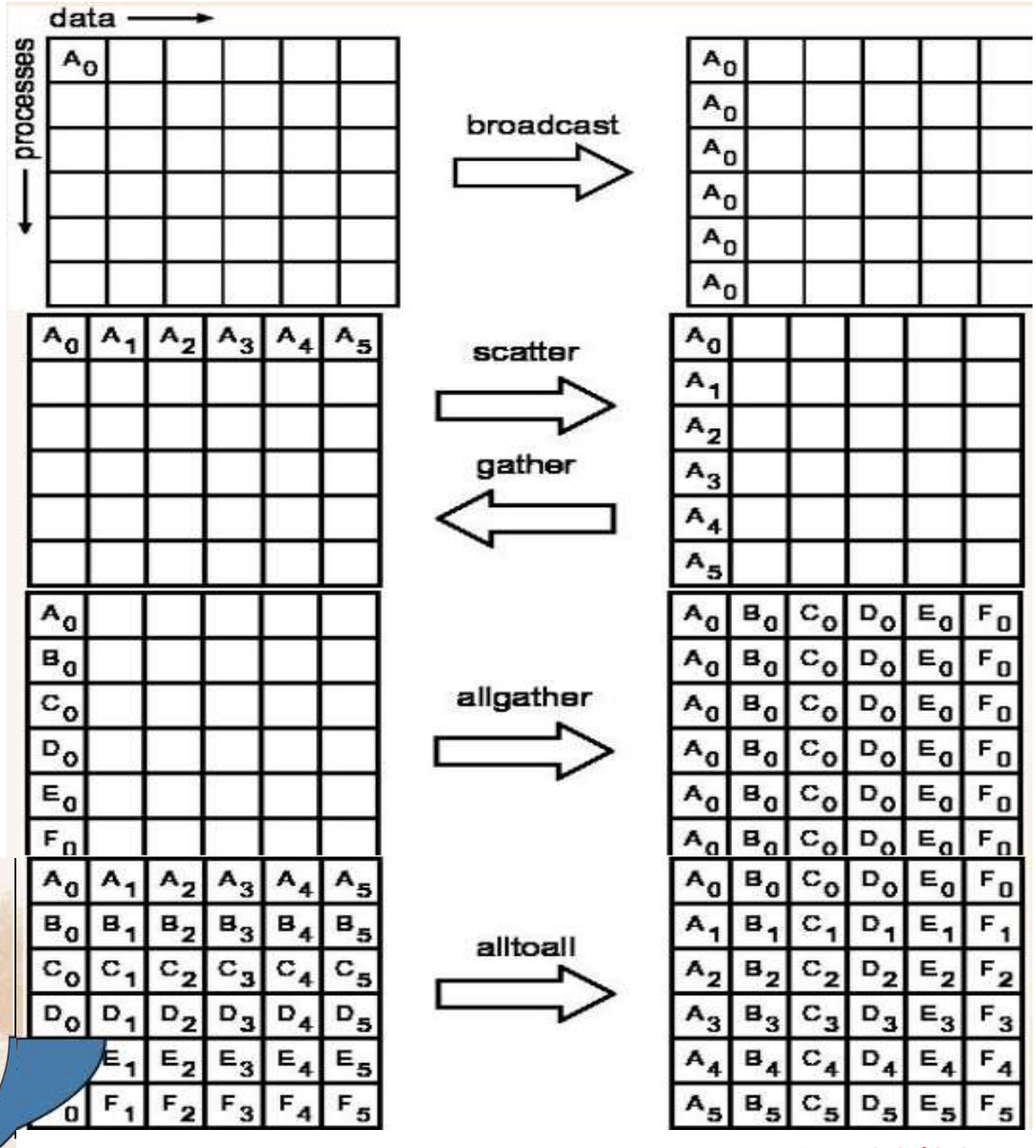
- Data movement operation. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.
- Result is as if each process in group A scatters its *sbuff* to each process in group B and each process in group B scatters its *sbuff* to each process in group A
- Data is gathered in *rbuff* in rank order according to the rank in the group providing the data
- Each process in group A sends the same amount of data to group B and vice-versa

```
MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf,
       recvcnt,recvtype,comm)
MPI_ALLTOALL (sendbuf,sendcount,sendtype,recvbuf,
       recvcnt,recvtype,comm,ierr)
```



MPI_ALLTOALL




data
processes
broadcast
scatter
gather
allgather
alltoall

MPI_Scan

- Performs a scan operation with respect to a reduction operation across a task group.

```
MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)
MPI_SCAN (sendbuf,recvbuf,count,datatype,op,comm,ierr)
```

- **MPI_Barrier**

_Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.

```
MPI_Barrier (comm)
MPI_BARRIER (comm,ierr)
```

Examples: Collective Communications

-Perform a scatter operation on the rows of an array

C Language - Collective Communications Example

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {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}  };
float recvbuf[SIZE];

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

if (numtasks == SIZE) {
  // define source task and elements to send/receive, then perform collective scatter
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Fortran - Collective Communications Example

```
program scatter
include 'mpif.h'

integer SIZE
parameter(SIZE=4)
integer numtasks, rank, sendcount, recvcount, source, ierr
real*4 sendbuf(SIZE,SIZE), recvbuf(SIZE)

! Fortran stores this array in column major order, so the
! scatter will actually scatter columns, not rows.
data sendbuf /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 /

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

if (numtasks .eq. SIZE) then
   ! define source task and elements to send/receive, then perform collective scatter
   source = 1
   sendcount = SIZE
   recvcount = SIZE
   call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf, recvcount, MPI_REAL, &
                    source, MPI_COMM_WORLD, ierr)

   print *, 'rank= ',rank,' Results: ',recvbuf

else
   print *, 'Must specify',SIZE,' processors.  Terminating.'
endif

call MPI_FINALIZE(ierr)

end
```

- Sample program output:

```
rank= 0 Results: 1.000000 2.000000 3.000000 4.000000
rank= 1 Results: 5.000000 6.000000 7.000000 8.000000
rank= 2 Results: 9.000000 10.000000 11.000000 12.000000
rank= 3 Results: 13.000000 14.000000 15.000000 16.000000
```

Three types of collective

Rooted:

- MPI_Gather and MPI_Gatherv
- MPI_Reduce
- MPI_Scatter and MPI_Scatterv
- MPI_Bcast

All-to-all:

- MPI_Allgather and MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv, and MPI_Alltoallw
- MPI_Allreduce, MPI_Reduce_scatter

Other:

- MPI_Scan, MPI_Exscan, and MPI_Barrier

Performance Issues

Operation	Cost (small N)	Cost (large N)
MPI_BARRIER	$t_s \log P$	$t_s \log P$
MPI_BCAST	$\log P(t_s + t_w N)$	$2(t_s \log P + t_w N)$
MPI_SCATTER	$t_s \log P + t_w N$	$t_s \log P + t_w N$
MPI_GATHER	$t_s \log P + t_w N$	$t_s \log P + t_w N$
MPI_REDUCE	$\log P(t_s + (t_w + t_{op})N)$	$t_s 2 \log P + (t_w 2 + t_{op})N$
MPI_ALLREDUCE	$\log P(t_s + (t_w + t_{op})N)$	$t_s 2 \log P + (t_w 2 + t_{op})N$

- Communication costs associated with various MPI global operations when implemented using hypercube communication algorithms on the idealized multicomputer architecture. The term t_{op} represents the cost of a single reduction operation.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
      int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
      int source, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

```
int MPI_Get_count(MPI_Status *status, MPI_datatype type,
      int *count)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
      MPI_Status *status)
int MPI_Probe(int source, int tag, MPI_Comm comm,
      MPI_Status *status)
```

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype)
      MPI_Datatype *newtype)
int MPI_Type_vector(int count, int blocklen, int stride,
      MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_indexed(int count, int *blocklens, int *indices,
      MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *datatype)
int MPI_Type_free(MPI_Datatype *datatype)
```

```
int MPI_Barrier(MPI_Comm comm)
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
      int root, MPI_Comm comm)
int MPI_Gather(void *inbuf, int incnt, MPI_Datatype intype,
      void *outbuf, int outcnt, MPI_Datatype outtype,
      int root, MPI_Comm comm)
int MPI_Scatter(void *inbuf, int incnt, MPI_Datatype intype,
      void *outbuf, int outcnt, MPI_Datatype outtype,
      int root, MPI_Comm comm)
int MPI_Reduce(void *inbuf, void *outbuf, int count,
      MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
int MPI_Allreduce(void *inbuf, void *outbuf, int count,
      MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

```
int MPI_Comm_dup(MPI_Comm comm, MPI_comm *newcomm)
int MPI_Comm_split(MPI_Comm comm, int color, int key,
      MPI_comm *newcomm)
int MPI_Comm_free(MPI_Comm *comm)
int MPI_Intercomm_create(MPI_Comm comm, int leader1,
      MPI_Comm peer, int leader2, int tag, MPI_Comm *inter)
```

MPI quick reference: C language binding

```
MPI_SEND(BUF, ICOUNT, ITYPE, IDEST, ITAG, ICOMM, IERR)
        <type> BUF(*)
MPI_RECV(BUF, ICOUNT, ITYPE, ISOURCE, ITAG, ICOMM, ISTATUS,
       IERR)
        <type> BUF(*)
MPI_COMM_SIZE(ICOMM, ISIZE, IERR)
MPI_COMM_RANK(ICOMM, IRANK, IERR)
MPI_INIT(IERR)
MPI_FINALIZE(IERR)
```

```
MPI_GET_COUNT(ISTATUS, ITYPE, ICOUNT, IERR)
MPI_IPROBE(ISOURCE, ITAG, ICOMM, FLAG, ISTATUS, IERR)
        LOGICAL FLAG
MPI_PROBE(ISOURCE, ITAG, ICOMM, ISTATUS, IERR)
```

```
MPI_TYPE_CONTIGUOUS(ICOUNT, IOLDTYPE, INEWTYPE, IERR)
MPI_TYPE_VECTOR(ICOUNT, IBLOCKLEN, ISTRIDE, IOLDTYPE,
       INEWTYPE, IERR)
MPI_TYPE_INDEXED(ICOUNT, IBLOCKLENS, INDICES, IOLDTYPE,
        INEWTYPE, IERR)
        INTEGER IBLOCKLENS(*), INDICES(*)
MPI_TYPE_COMMIT(ITYPE, IERR)
MPI_TYPE_FREE(ITYPE, IERR)
```

```
MPI_BARRIER(ICOMM, IERR)
MPI_BCAST(BUF, ICOUNT, ITYPE, IROOT, ICOMM, IERR)
        <type> BUF(*)
MPI_GATHER(INBUF, INCNT, INTYPE, OUTBUF, IOUTCNT, IOUTTYPE,
        IROOT, ICOMM, IERR)
        <type> INBUF(*), OUTBUF(*)
MPI_SCATTER(INBUF, INCNT, INTYPE, OUTBUF, IOUTCNT, IOUTTYPE,
        IROOT, ICOMM, IERR)
        <type> INBUF(*), OUTBUF(*)
MPI_REDUCE(INBUF, OUTBUF, ICOUNT, ITYPE, IOP, IROOT, ICOMM,
        IERR)
        <type> INBUF(*), OUTBUF(*)
MPI_ALLREDUCE(INBUF, OUTBUF, ICOUNT, ITYPE, IOP, ICOMM, IERR)
        <type> INBUF(*), OUTBUF(*)
```

```
MPI_COMM_DUP(ICOMM, INEWCOMM, IERR)
MPI_COMM_SPLIT(ICOMM, ICOLOR, IKEY, INEWCOMM, IERR)
MPI_COMM_FREE(ICOMM, IERR)
MPI_INTERCOMM_CREATE(ICOMM, ILEADER1, IPEER, ILEADER2,
        ITAG, INTERCOMM, IERR)
```

MPI quick reference: Fortran language binding.

(4) Derived Data Types

- MPI predefines its primitive data types:

C Data Types		Fortran Data Types
MPI_CHAR	MPI_C_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_FLOAT_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_DOUBLE_COMPLEX	MPI_INTEGER1
MPI_INT	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER2
MPI_LONG	MPI_C_BOOL	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_LOGICAL	MPI_REAL
MPI_LONG_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT8_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT16_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_INT32_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_INT64_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT8_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT16_T	MPI_LOGICAL
MPI_DOUBLE	MPI_UINT32_T	MPI_BYTE
MPI_LONG_DOUBLE	MPI_UINT64_T	MPI_PACKED
	MPI_BYTE	
	MPI_PACKED	

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.
- Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Derived Data Type Routines

Derived Types Routines			
MPI_Type_commit	MPI_Type_contiguous	MPI_Type_extent*	MPI_Type_free
MPI_Type_hindexed*	MPI_Type_hvector*	MPI_Type_indexed	MPI_Type_lb
MPI_Type_size	MPI_Type_struct*	MPI_Type_ub*	MPI_Type_vector

- **MPI_Type_contiguous**

_The simplest constructor. Produces a new data type by making count copies of an existing data type.

```
MPI_Type_contiguous (count,oldtype,&newtype)
MPI_TYPE_CONTIGUOUS (count,oldtype,newtype,ierr)
```

MPI_Type_vector
MPI_Type_hvector

- Similar to contiguous, but allows for regular gaps (stride) in the displacements. MPI_Type_hvector is identical to MPI_Type_vector except that stride is specified in bytes.

```
MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)
MPI_TYPE_VECTOR (count,blocklength,stride,oldtype,newtype,ierr)
```

MPI_Type_indexed
MPI_Type_hindexed

- An array of displacements of the input data type is provided as the map for the new data type. MPI_Type_hindexed is identical to MPI_Type_indexed except that offsets are specified in bytes.

```
MPI_Type_indexed (count,blocklens[],offsets[],old_type,&newtype)
MPI_TYPE_INDEXED (count,blocklens(),offsets(),old_type,newtype,ierr)
```

MPI_Type_struct

- The new data type is formed according to completely defined map of the component data types.
  NOTE: This function is deprecated in MPI-2.0 and replaced by MPI_Type_create_struct in MPI-3.0

```
MPI_Type_struct (count,blocklens[],offsets[],old_types,&newtype)
MPI_TYPE_STRUCT (count,blocklens(),offsets(),old_types,newtype,ierr)
```

MPI_Type_extent

- Returns the size in bytes of the specified data type. Useful for the MPI subroutines that require specification of offsets in bytes.
  NOTE: This function is deprecated in MPI-2.0 and replaced by MPI_Type_get_extent in MPI-3.0

```
MPI_Type_extent (datatype,&extent)
MPI_TYPE_EXTENT (datatype,extent,ierr)
```

MPI_Type_commit

- Commits new datatype to the system. Required for all user constructed (derived) datatypes.

```
MPI_Type_commit (&datatype)
MPI_TYPE_COMMIT (datatype,ierr)
```

MPI_Type_free

- Deallocates the specified datatype object. Use of this routine is especially important to prevent memory exhaustion if many datatype objects are created, as in a loop.

```
MPI_Type_free (&datatype)
MPI_TYPE_FREE (datatype,ierr)
```

Examples: Contiguous Derived Data Type

- Create a data type representing a row of an array and distribute a different row to all processes.

C Language - Contiguous Derived Data Type Example

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {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};
float b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;   // required variable

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

// create contiguous derived data type
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
   // task 0 sends one element of rowtype to all tasks
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
      }

   // all tasks receive rowtype data from task 0
   MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

// free datatype when done using it
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

Fortran - Contiguous Derived Data Type Example

```
1  program contiguous
2  include 'mpif.h'
3
4  integer SIZE
5  parameter(SIZE=4)
6  integer numtasks, rank, source, dest, tag, i,  ierr
7  real*4 a(0:SIZE-1,0:SIZE-1), b(0:SIZE-1)
8  integer stat(MPI_STATUS_SIZE)
9  integer columntype   ! required variable
10 tag = 1
11
12 ! Fortran stores this array in column major order
13 data a  /1.0, 2.0, 3.0, 4.0, &
14         5.0, 6.0, 7.0, 8.0, &
15         9.0, 10.0, 11.0, 12.0, &
16         13.0, 14.0, 15.0, 16.0 /
17
18 call MPI_INIT(ierr)
19 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
20 call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
21
22 ! create contiguous derived data type
23 call MPI_TYPE_CONTIGUOUS(SIZE, MPI_REAL, columntype, ierr)
24 call MPI_TYPE_COMMIT(columntype, ierr)
25
26 if (numtasks .eq. SIZE) then
27    ! task 0 sends one element of columntype to all tasks
28    if (rank .eq. 0) then
29       do i=0, numtasks-1
30       call MPI_SEND(a(0,i), 1, columntype, i, tag, MPI_COMM_WORLD,ierr)
31       end do
32    endif
33
34    ! all tasks receive columntype data from task 0
35    source = 0
36    call MPI_RECV(b, SIZE, MPI_REAL, source, tag, MPI_COMM_WORLD, stat, ierr)
37    print *, 'rank= ',rank,' b= ',b
38 else
39    print *, 'Must specify',SIZE,' processors.  Terminating.'
40 endif
41
42 ! free datatype when done using it
43 call MPI_TYPE_FREE(columntype, ierr)
44 call MPI_FINALIZE(ierr)
45
46 end
```

- Sample program output:

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

Examples: Vector Derived Data Type

- Create a data type representing a column of an array and distribute different columns to all processes.

C Language - Vector Derived Data Type Example

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {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};
float b[SIZE];

MPI_Status stat;
MPI_Datatype columntype;   // required variable


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

// create vector derived data type
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);

if (numtasks == SIZE) {
   // task 0 sends one element of columntype to all tasks
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
         MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
      }

   // all tasks receive columntype data from task 0
   MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

// free datatype when done using it
MPI_Type_free(&columntype);
MPI_Finalize();
}
```

Fortran - Vector Derived Data Type Example

```
program vector
include 'mpif.h'

integer SIZE
parameter(SIZE=4)
integer numtasks, rank, source, dest, tag, i,  ierr
real*4 a(0:SIZE-1,0:SIZE-1), b(0:SIZE-1)
integer stat(MPI_STATUS_SIZE)
integer rowtype    ! required variable
tag = 1

! Fortran stores this array in column major order
data a  /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 /

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

! create vector derived data type
call MPI_TYPE_VECTOR(SIZE, 1, SIZE, MPI_REAL, rowtype, ierr)
call MPI_TYPE_COMMIT(rowtype, ierr)

if (numtasks .eq. SIZE) then
   ! task 0 sends one element of rowtype to all tasks
   if (rank .eq. 0) then
      do i=0, numtasks-1
      call MPI_SEND(a(i,0), 1, rowtype, i, tag, MPI_COMM_WORLD, ierr)
      end do
   endif

   ! all tasks receive rowtype data from task 0
   source = 0
   call MPI_RECV(b, SIZE, MPI_REAL, source, tag, MPI_COMM_WORLD, stat, ierr)
   print *, 'rank= ',rank,' b= ',b
else
   print *, 'Must specify',SIZE,' processors.  Terminating.'
endif

! free datatype when done using it
call MPI_TYPE_FREE(rowtype, ierr)
call MPI_FINALIZE(ierr)

end
```

- Sample program output:

```
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
```

Examples: Indexed Derived Data Type

- Create a datatype by extracting variable portions of an array and distribute to all tasks.

C Language - Indexed Derived Data Type Example

```c
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] =
  {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};
float b[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;   // required variable

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

blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;

// create indexed derived data type
MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
MPI_Type_commit(&indextype);

if (rank == 0) {
  for (i=0; i<numtasks; i++)
    // task 0 sends one element of indextype to all tasks
      MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

// all tasks receive indextype data from task 0
MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
       rank,b[0],b[1],b[2],b[3],b[4],b[5]);

// free datatype when done using it
MPI_Type_free(&indextype);
MPI_Finalize();
}
```

Fortran - Indexed Derived Data Type Example

```
1   program indexed
2   include 'mpif.h'
3
4   integer NELEMENTS
5   parameter(NELEMENTS=6)
6   integer numtasks, rank, source, dest, tag, i,  ierr
7   integer blocklengths(0:1), displacements(0:1)
8   real*4 a(0:15), b(0:NELEMENTS-1)
9   integer stat(MPI_STATUS_SIZE)
10  integer indextype   ! required variable
11  tag = 1
12
13  data a  /1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, &
14           9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0 /
15
16  call MPI_INIT(ierr)
17  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
18  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
19
20  blocklengths(0) = 4
21  blocklengths(1) = 2
22  displacements(0) = 5
23  displacements(1) = 12
24
25  ! create indexed derived data type
26  call MPI_TYPE_INDEXED(2, blocklengths, displacements, MPI_REAL, &
27                        indextype, ierr)
28  call MPI_TYPE_COMMIT(indextype, ierr)
29
30  if (rank .eq. 0) then
31     ! task 0 sends one element of indextype to all tasks
32     do i=0, numtasks-1
33     call MPI_SEND(a, 1, indextype, i, tag, MPI_COMM_WORLD, ierr)
34     end do
35  endif
36
37  ! all tasks receive indextype data from task 0
38  source = 0
39  call MPI_RECV(b, NELEMENTS, MPI_REAL, source, tag, MPI_COMM_WORLD, &
40                stat, ierr)
41  print *, 'rank= ',rank,' b= ',b
42
43  ! free datatype when done using it
44  call MPI_TYPE_FREE(indextype, ierr)
45  call MPI_FINALIZE(ierr)
46
47  end
```

Sample program output:

```
rank= 0   b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 1   b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 2   b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 3   b= 6.0 7.0 8.0 9.0 13.0 14.0
```

Examples: Struct Derived Data Type

- Create a data type that represents a particle and distribute an array of such particles to all processes.

C Language - Struct Derived Data Type Example

```c
#include "mpi.h"
#include <stdio.h>
#define NELEM 25

main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;

typedef struct {
  float x, y, z;
  float velocity;
  int  n, type;
  }           Particle;
Particle     p[NELEM], particles[NELEM];
MPI_Datatype particletype, oldtypes[2];   // required variables
int          blockcounts[2];

// MPI_Aint type used to be consistent with syntax of
// MPI_Type_extent routine
MPI_Aint    offsets[2], extent;

MPI_Status stat;

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

// setup description of the 4 MPI_FLOAT fields x, y, z, velocity
offsets[0] = 0;
oldtypes[0] = MPI_FLOAT;
blockcounts[0] = 4;

// setup description of the 2 MPI_INT fields n, type
// need to first figure offset by getting size of MPI_FLOAT
MPI_Type_extent(MPI_FLOAT, &extent);
offsets[1] = 4 * extent;
oldtypes[1] = MPI_INT;
blockcounts[1] = 2;
```

```
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  }
```

Fortan - Struct Derived Data Type Example

```
1  program struct
2  include 'mpif.h'
3
4  integer NELEM
5  parameter(NELEM=25)
6  integer numtasks, rank, source, dest, tag, i,  ierr
7  integer stat(MPI_STATUS_SIZE)
8
9  type Particle
10 sequence
11 real*4 x, y, z, velocity
12 integer n, type
13 end type Particle
14
15 type (Particle) p(NELEM), particles(NELEM)
16 integer particletype, oldtypes(0:1)    ! required variables
17 integer blockcounts(0:1), offsets(0:1), extent
18 tag = 1
19
20 call MPI_INIT(ierr)
21 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
22 call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
23
24 ! setup description of the 4 MPI_REAL fields x, y, z, velocity
25 offsets(0) = 0
26 oldtypes(0) = MPI_REAL
27 blockcounts(0) = 4
28
29 ! setup description of the 2 MPI_INTEGER fields n, type
30 ! need to first figure offset by getting size of MPI_REAL
31 call MPI_TYPE_EXTENT(MPI_REAL, extent, ierr)
32 offsets(1) = 4 * extent
33 oldtypes(1) = MPI_INTEGER
34 blockcounts(1) = 2
35
```

```
36  ! define structured type and commit it
37  call MPI_TYPE_STRUCT(2, blockcounts, offsets, oldtypes, &
38                       particletype, ierr)
39  call MPI_TYPE_COMMIT(particletype, ierr)
40
41  ! task 0 initializes the particle array and then sends it to each task
42  if (rank .eq. 0) then
43     do i=0, NELEM-1
44     particles(i) = Particle ( 1.0*i, -1.0*i, 1.0*i, 0.25, i, mod(i,2) )
45     end do
46
47     do i=0, numtasks-1
48     call MPI_SEND(particles, NELEM, particletype, i, tag, &
49                   MPI_COMM_WORLD, ierr)
50     end do
51  endif
52
53  ! all tasks receive particletype data
54  source = 0
55  call MPI_RECV(p, NELEM, particletype, source, tag, &
56                MPI_COMM_WORLD, stat, ierr)
57
58  print *, 'rank= ',rank,' p(3)= ',p(3)
59
60  ! free datatype when done using it
61  call MPI_TYPE_FREE(particletype, ierr)
62  call MPI_FINALIZE(ierr)
63  end
```

Sample program output:

```
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
```

(5) Group and Communicator Management Routines

Process Group Routines			
MPI_Group_compare	MPI_Group_difference	MPI_Group_excl	MPI_Group_free
MPI_Group_incl	MPI_Group_intersection	MPI_Group_range_excl	MPI_Group_range_incl
MPI_Group_rank	MPI_Group_size	MPI_Group_translate_ranks	MPI_Group_union
Communicators Routines			
MPI_Comm_compare	MPI_Comm_create	MPI_Comm_dup	MPI_Comm_free
MPI_Comm_group	MPI_Comm_rank	MPI_Comm_remote_group	MPI_Comm_remote_size
MPI_Comm_size	MPI_Comm_split	MPI_Comm_test_inter	MPI_Intercomm_create
MPI_Intercomm_merge			

- **Groups vs. Communicators:**
  - A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.

Groups vs. Communicators:

- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles". For example, the handle for the communicator that comprises all tasks is MPI_COMM_WORLD.
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

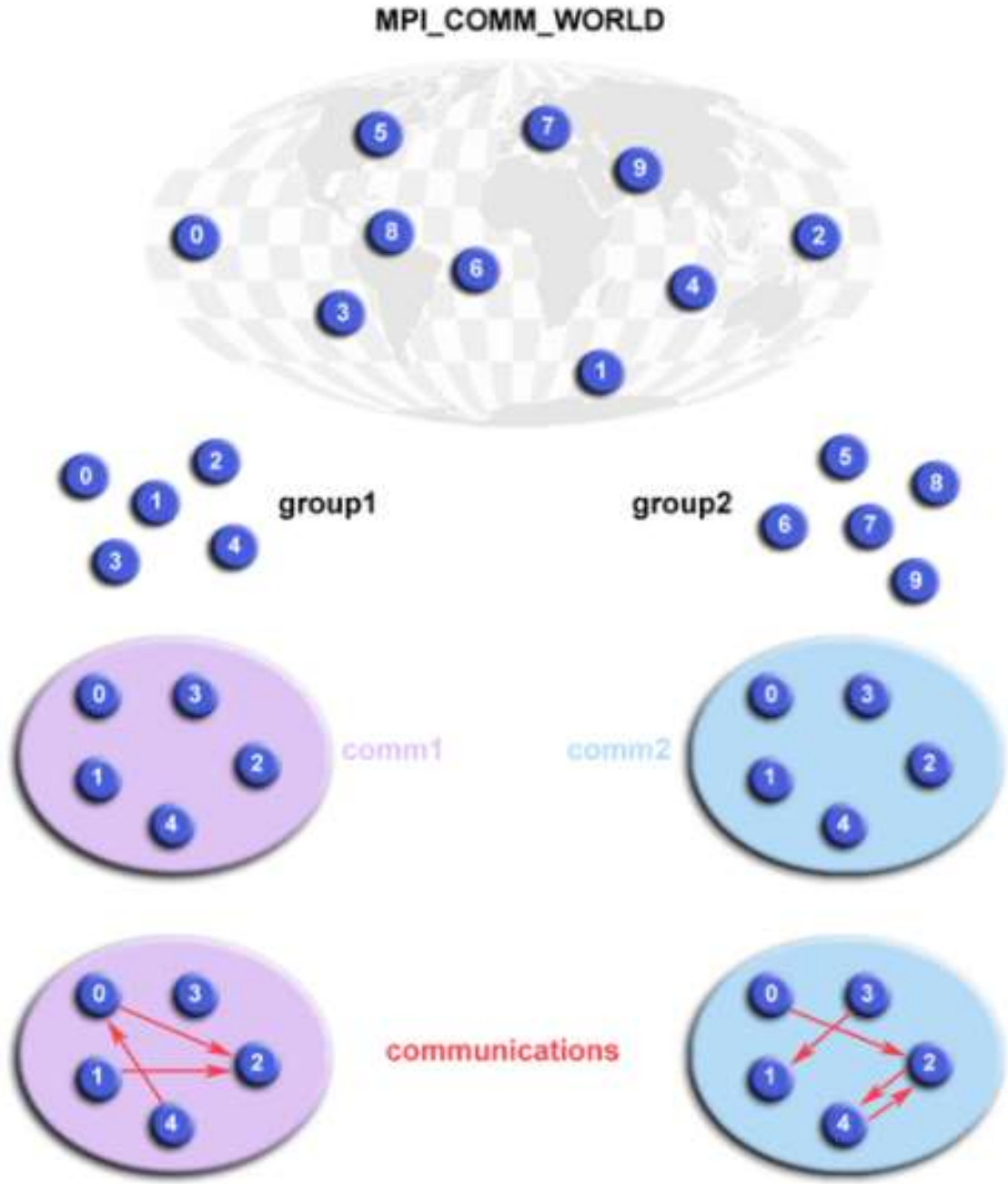
Primary Purposes of Group and Communicator Objects:

- Allow you to organize tasks, based upon function, into task groups.
- Enable Collective Communications operations across a subset of related tasks.
- Provide basis for implementing user defined virtual topologies
- Provide for safe communications

Programming Considerations and Restrictions:

- Groups/communicators are dynamic - they can be created and destroyed during program execution.
- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.
- MPI provides over 40 routines related to groups, communicators, and virtual topologies.

- Typical usage:Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  - Form new group as a subset of global group using MPI_Group_incl
  - Create new communicator for new group using MPI_Comm_create
  - Determine new rank in new communicator using MPI_Comm_rank
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free


MPI_COMM_WORLD
group1
group2
comm1
comm2
communications

Group and Communicator Management Routines

- Create two different process groups for separate collective communications exchange. Requires creating new communicators also.

C Language - Group and Communicator Example

```c
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

main(int argc, char *argv[])  {
int         rank, new_rank, sendbuf, recvbuf, numtasks,
            ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group   orig_group, new_group;    // required variables
MPI_Comm    new_comm;    // required variable

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

if (numtasks != NPROCS) {
  printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
  MPI_Finalize();
  exit(0);
  }

sendbuf = rank;

// extract the original group handle
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

//  divide tasks into two distinct groups based upon rank
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

// create new new communicator and then perform collective communications
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

// get rank in new group
MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

MPI_Finalize();
}
```

Fortran - Group and Communicator Example

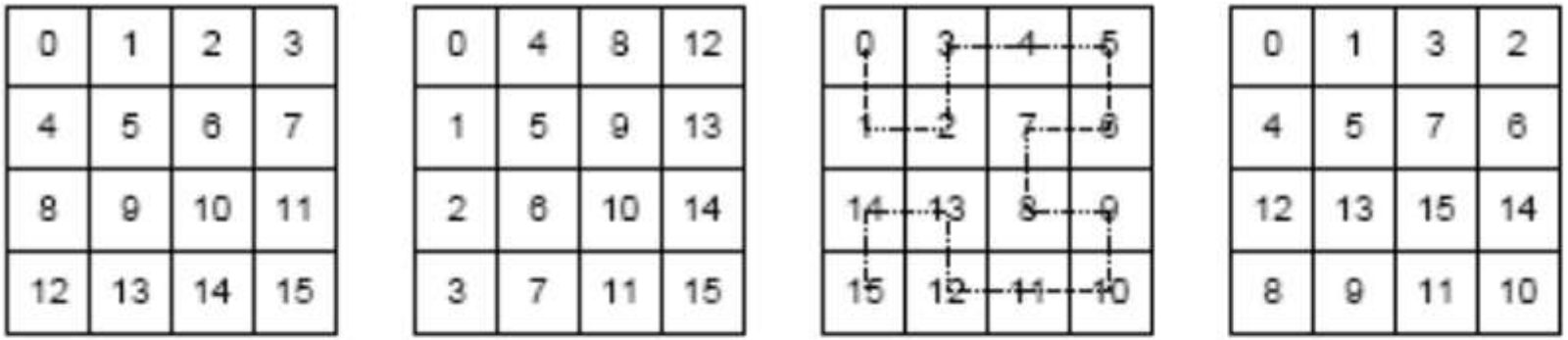
```
1   program group
2   include 'mpif.h'
3
4   integer NPROCS
5   parameter(NPROCS=8)
6   integer rank, new_rank, sendbuf, recvbuf, numtasks
7   integer ranks1(4), ranks2(4), ierr
8   integer orig_group, new_group, new_comm    ! required variables
9   data ranks1 /0, 1, 2, 3/, ranks2 /4, 5, 6, 7/
10
11  call MPI_INIT(ierr)
12  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
13  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
14
15  if (numtasks .ne. NPROCS) then
16    print *, 'Must specify NPROCS= ',NPROCS,' Terminating.'
17    call MPI_FINALIZE(ierr)
18    stop
19  endif
20
21  sendbuf = rank
22
23  ! extract the original group handle
24  call MPI_COMM_GROUP(MPI_COMM_WORLD, orig_group, ierr)
25
26  ! divide tasks into two distinct groups based upon rank
27  if (rank .lt. NPROCS/2) then
28     call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks1, new_group, ierr)
29  else
30     call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks2, new_group, ierr)
31  endif
32
33  ! create new new communicator and then perform collective communications
34  call MPI_COMM_CREATE(MPI_COMM_WORLD, new_group, new_comm, ierr)
35  call MPI_ALLREDUCE(sendbuf, recvbuf, 1, MPI_INTEGER, MPI_SUM, new_comm, ierr)
36
37  ! get rank in new group
38  call MPI_GROUP_RANK(new_group, new_rank, ierr)
39  print *, 'rank= ',rank,' newrank= ',new_rank,' recvbuf= ', recvbuf
40
41  call MPI_FINALIZE(ierr)
42  end
```

Sample program output:

```
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
```

(6) Virtual Topologies

- **What Are They?**
  - In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".
  - The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
  - MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
  - Virtual topologies are built upon MPI communicators and groups.
  - Must be "programmed" by the application developer.



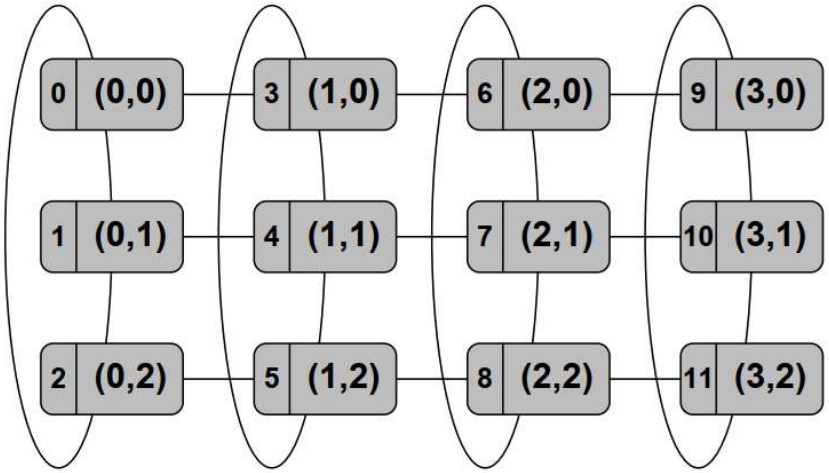
- Different ways to map a set of processes to a two-dimensional grid
  - (a) and (b) show a row- and column-wise mapping of these processes
  - (c) shows a mapping that follows a space-filling curve (dotted line)
  - (d) shows a mapping in which neighboring processes are directly connected in a hypercube

Examples:

- A simplified mapping of processes into a Cartesian virtual topology appears below:

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)

- Two-dimensional Cartesian topology: 12 processes form a 3×4 grid, which is periodic in the second dimension but not in the first. The mapping between MPI ranks and Cartesian coordinates is shown.



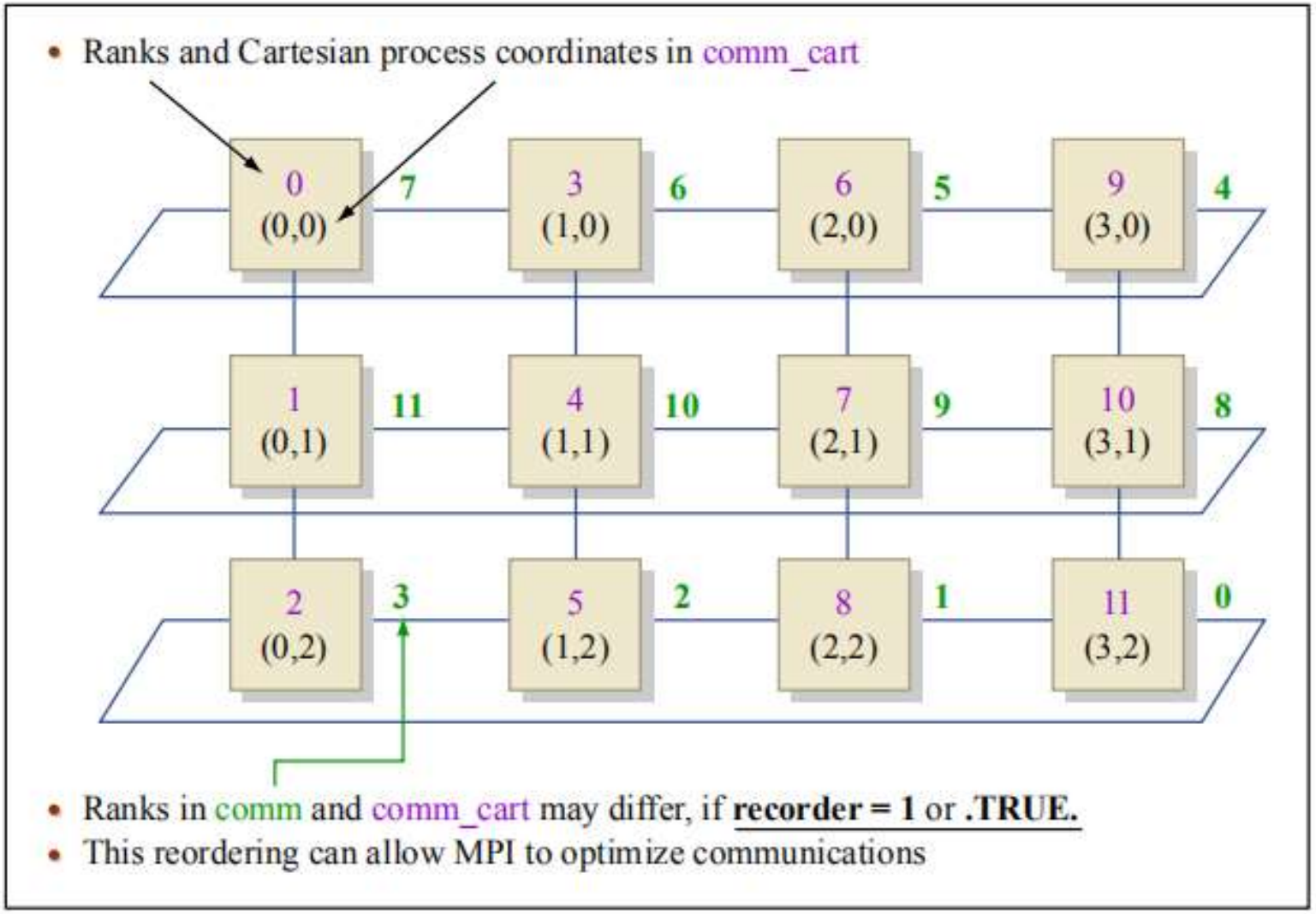


Figure by MIT OpenCourseWare.

Why Use Them?

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.
- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

Virtual Topology Routines

Virtual Topology Routines			
MPI_Cart_coords	MPI_Cart_create	MPI_Cart_get	MPI_Cart_map
MPI_Cart_rank	MPI_Cart_shift	MPI_Cart_sub	MPI_Cartdim_get
MPI_Dims_create	MPI_Graph_create	MPI_Graph_get	MPI_Graph_map
MPI_Graph_neighbors	MPI_Graph_neighbors_count	MPI_Graphdims_get	MPI_Topo_test
Miscellaneous Routines			
MPI_Address*	MPI_Attr_delete*	MPI_Attr_get*	MPI_Attr_put*
MPI_Keyval_create*	MPI_Keyval_free*	MPI_Pack	MPI_Pack_size
MPI_Pcontrol	MPI_Unpack		

Cartesian rank/coordinate functions

- MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank); out of range values get shifted (periodic topos)
- MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

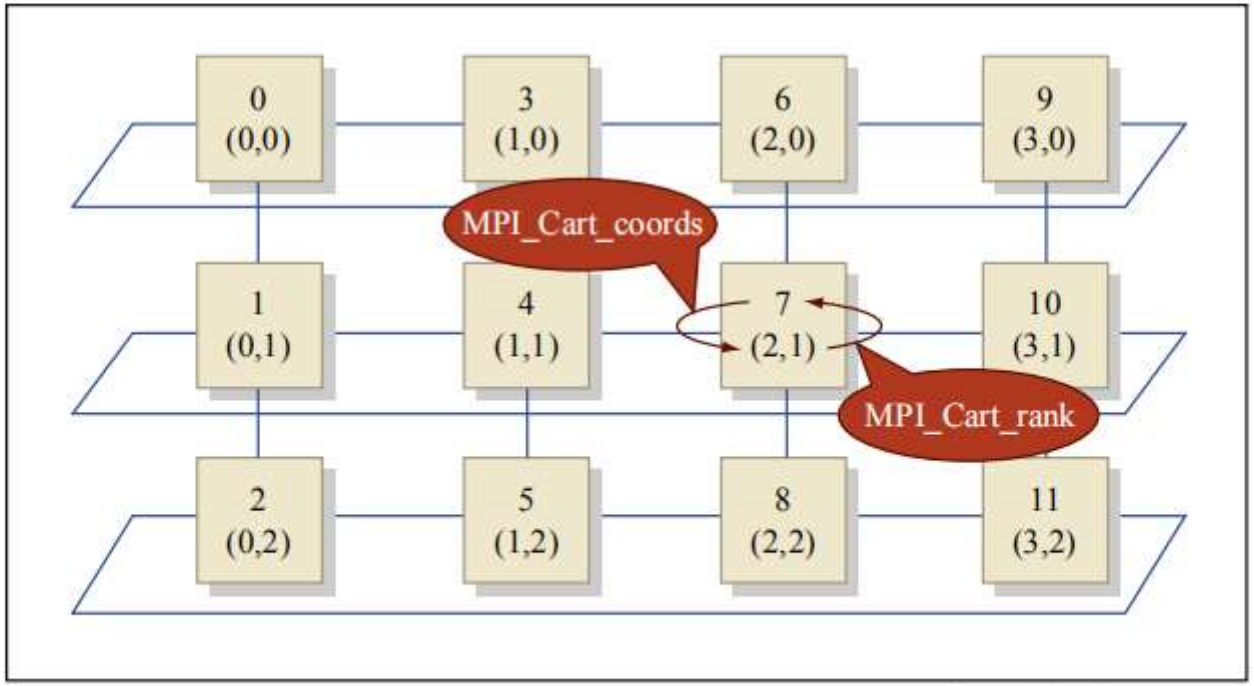


Figure by MIT OpenCourseWare.

Cartesian shift

- MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
  - MPI_PROC_NULL for shifts at non-periodic boundaries

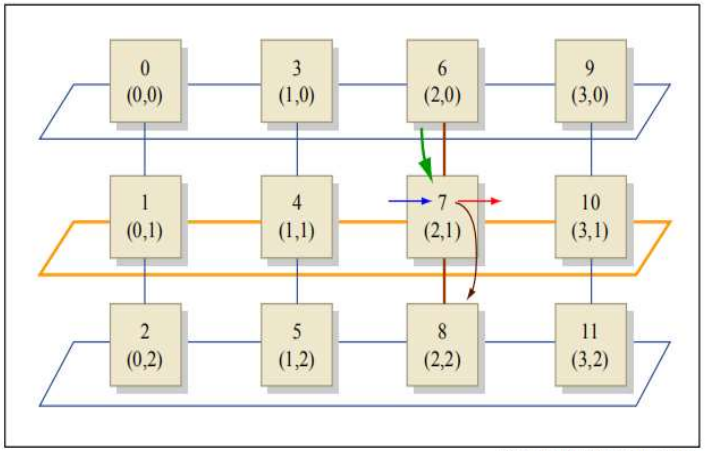


Figure by MIT OpenCourseWare.

Cartesian subspaces

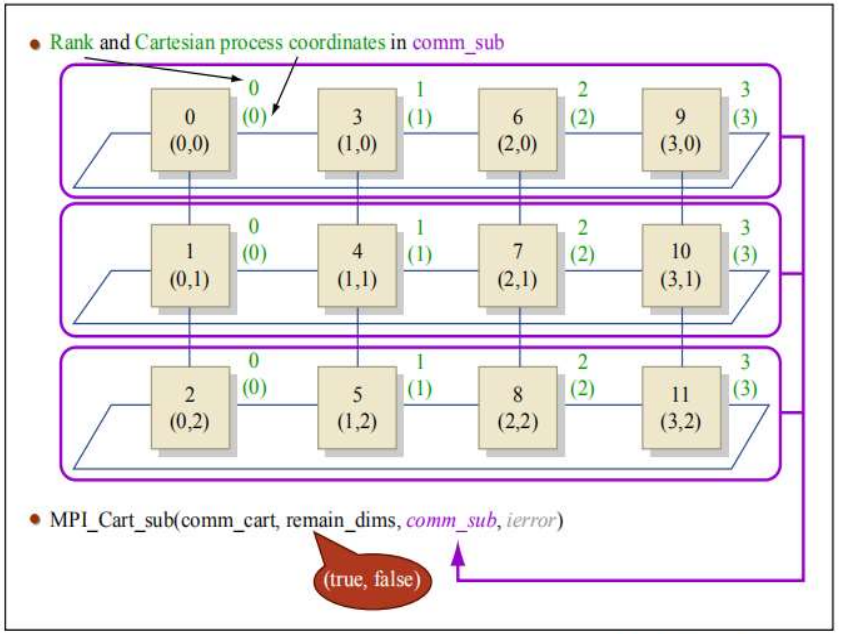


Figure by MIT OpenCourseWare.

- Create a 4 x 4 Cartesian topology from 16 processors and have each process exchange its rank with four neighbors.

C Language - Cartesian Virtual Topology Example

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

main(int argc, char *argv[])  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   nbrs[4], dims[2]={4,4},
   periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;   // required variable

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

if (numtasks == SIZE) {
   // create cartesian virtual topology, get rank, coordinates, neighbor ranks
   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
   MPI_Comm_rank(cartcomm, &rank);
   MPI_Cart_coords(cartcomm, rank, 2, coords);
   MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
   MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

   printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
          rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
          nbrs[RIGHT]);

   outbuf = rank;

   // exchange data (rank) with 4 neighbors
   for (i=0; i<4; i++) {
      dest = nbrs[i];
      source = nbrs[i];
      MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD, &reqs[i]);
      MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &reqs[i+4]);
      }

   MPI_Waitall(8, reqs, stats);

   printf("rank= %d                  inbuf(u,d,l,r)= %d %d %d %d\n",
          rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);  }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Fortran - Cartesian Virtual Topology Example

```
program cartesian
include 'mpif.h'

integer SIZE, UP, DOWN, LEFT, RIGHT
parameter(SIZE=16)
parameter(UP=1)
parameter(DOWN=2)
parameter(LEFT=3)
parameter(RIGHT=4)
integer numtasks, rank, source, dest, outbuf, i, tag, ierr, &
        inbuf(4), nbrs(4), dims(2), coords(2), periods(2), reorder
integer stats(MPI_STATUS_SIZE, 8), reqs(8)
integer cartcomm   ! required variable
data inbuf /MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL/, &
     dims /4,4/, tag /1/, periods /0,0/, reorder /0/

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

if (numtasks .eq. SIZE) then
   ! create cartesian virtual topology, get rank, coordinates, neighbor ranks
   call MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, periods, reorder, &
                        cartcomm, ierr)
   call MPI_COMM_RANK(cartcomm, rank, ierr)
   call MPI_CART_COORDS(cartcomm, rank, 2, coords, ierr)
   call MPI_CART_SHIFT(cartcomm, 0, 1, nbrs(UP), nbrs(DOWN), ierr)
   call MPI_CART_SHIFT(cartcomm, 1, 1, nbrs(LEFT), nbrs(RIGHT), ierr)

   write(*,20) rank,coords(1),coords(2),nbrs(UP),nbrs(DOWN), &
               nbrs(LEFT),nbrs(RIGHT)

   ! exchange data (rank) with 4 neighbors
   outbuf = rank
   do i=1,4
      dest = nbrs(i)
      source = nbrs(i)
      call MPI_ISEND(outbuf, 1, MPI_INTEGER, dest, tag, &
                     MPI_COMM_WORLD, reqs(i), ierr)
      call MPI_IRECV(inbuf(i), 1, MPI_INTEGER, source, tag, &
                     MPI_COMM_WORLD, reqs(i+4), ierr)
   enddo

   call MPI_WAITALL(8, reqs, stats, ierr)

   write(*,30) rank,inbuf

else
  print *, 'Must specify',SIZE,' processors.  Terminating.'
endif
```

```
50
51      call MPI_FINALIZE(ierr)
52
53      20 format('rank= ',I3,' coords= ',I2,I2, &
54                ' neighbors(u,d,l,r)= ',I3,I3,I3,I3 )
55      30 format('rank= ',I3,'                    ', &
56                ' inbuf(u,d,l,r)= ',I3,I3,I3,I3 )
57
58      end
```

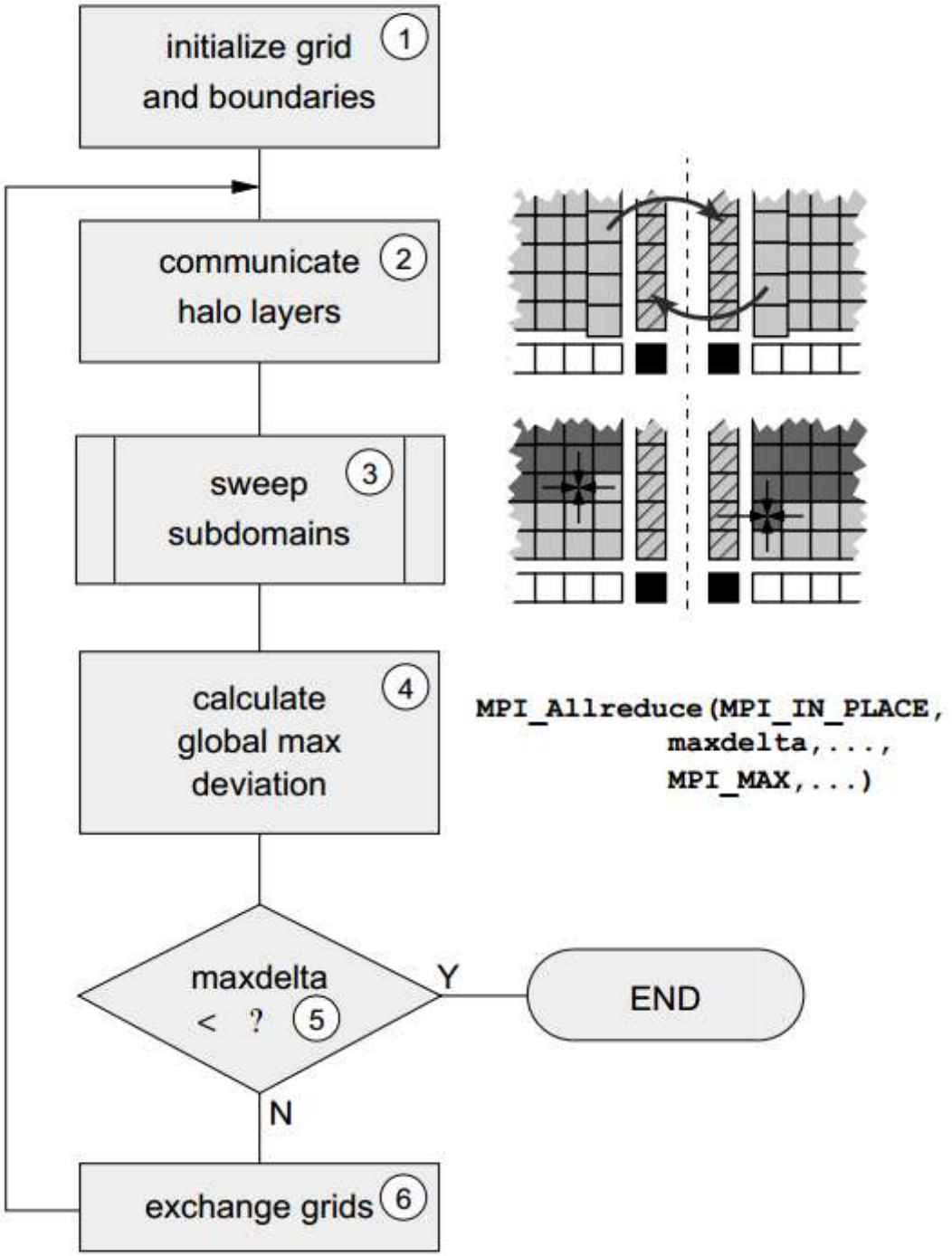
Sample program output: (partial)

```
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
```

- **Example-Jacobi**

Example-Jacobi

- Flowchart for distributed-memory parallelization of the Jacobi algorithm. Hatched cells are ghost layers, dark cells are already updated in the $T = 1$ grid, and light-colored cells denote $T = 0$ data. White cells are real boundaries of the overall grid, whereas black cells are unused.



- First the required parameters are read by **rank zero** from standard input (line 10 in the following listing): problem size (spat_dim), possible presets for number of processes (proc_dim), and periodicity (pbc_check), each for all dimensions.
- Although many MPI implementations have options to allow the standard input of rank zero to be seen by all processes, a portable MPI program cannot rely on this feature, and must broadcast the data (lines 14–16).

```
1  logical, dimension(1:3) :: pbc_check
2  integer, dimension(1:3) :: spat_dim, proc_dim
3
4  call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
5  call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)
6
7  if(myid.eq.0) then
8    write(*,*) ' spat_dim , proc_dim, PBC ? '
9    do i=1,3
10     read(*,*) spat_dim(i), proc_dim(i), pbc_check(i)
11   enddo
12 endif
13
14 call MPI_Bcast(spat_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
15 call MPI_Bcast(proc_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
16 call MPI_Bcast(pbc_check, 3, MPI_LOGICAL, 0, MPI_COMM_WORLD, ierr)
```

- After that, the Cartesian topology
  can be set up using MPI_Dims_create() and MPI_Cart_create
- Since rank reordering is allowed (line 6), the process rank must be obtained again using MPI_Comm_rank() (line 12). Moreover, the new Cartesian communicator GRID_COMM_WORLD may be of smaller size than MPI_COMM_WORLD. The “surplus” processes then receive a communicator value of MPI_COMM_NULL, and are sent into a barrier to wait for the whole parallel program to complete (line 10)

```
1  call MPI_Dims_create(numprocs, 3, proc_dim, ierr)
2
3  if(myid.eq.0) write(*,'(a,3(i3,x))') 'Grid: ', &
4      (proc_dim(i),i=1,3)
5
6  l_reorder = .true.
7  call MPI_Cart_create(MPI_COMM_WORLD, 3, proc_dim, pbc_check, &
8      l_reorder, GRID_COMM_WORLD, ierr)
9
10 if(GRID_COMM_WORLD .eq. MPI_COMM_NULL) goto 999
11
12 call MPI_Comm_rank(GRID_COMM_WORLD, myid_grid, ierr)
13 call MPI_Comm_size(GRID_COMM_WORLD, nump_grid, ierr)
```

- Now that the topology has been created, the local subdomains can be set up, including memory allocation:

```
1  integer, dimension(1:3) :: loca_dim, mycoord
2
3  call MPI_Cart_coords(GRID_COMM_WORLD, myid_grid, 3,
4      mycoord,ierr)
5
6  do i=1,3
7    loca_dim(i) = spat_dim(i)/proc_dim(i)
8    if(mycoord(i) < mod(spat_dim(i),proc_dim(i))) then
9      local_dim(i) = loca_dim(i)+1
10   endif
11 enddo
12
13 iStart = 0 ; iEnd = loca_dim(3)+1
14 jStart = 0 ; jEnd = loca_dim(2)+1
15 kStart = 0 ; kEnd = loca_dim(1)+1
16
17 allocate(phi(iStart:iEnd, jStart:jEnd, kStart:kEnd,0:1))
```

- Array mycoord is used to store a process' Cartesian coordinates as acquired from MPI_Cart_coords() in line 3. Array loca_dim holds the extensions of a process' subdomain in the three dimensions. These numbers are calculated in lines 6–11. Memory allocation takes place in line 17, allowing for an additional layer in all directions, which is used for fixed boundaries or halo as needed. For brevity, we are omitting the initialization of the array and its outer grid boundaries here

- We use two intermediate buffers per process, one for sending and one for receiving. Since the amount of halo data can be different along different Cartesian directions, the size of the intermediate buffer must be chosen to accommodate the largest possible halo:

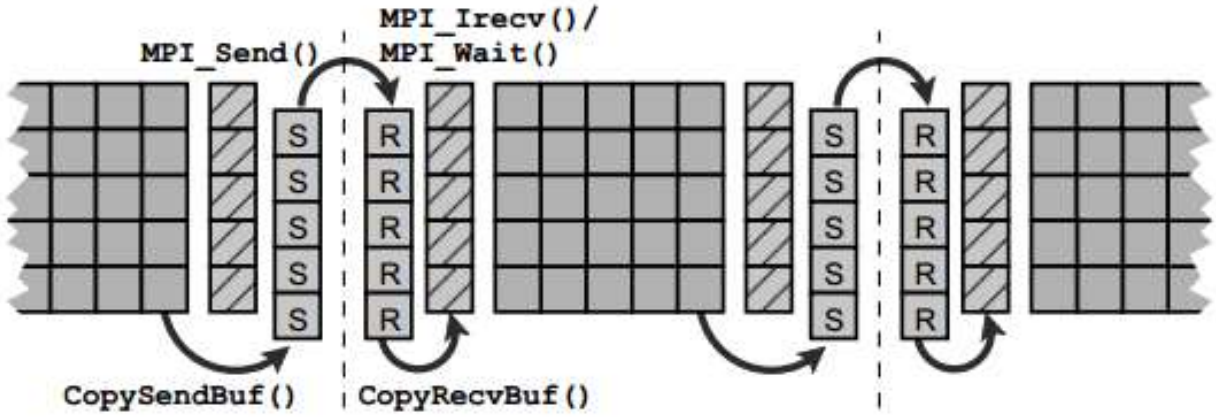
```
integer, dimension(1:3) :: totmsgsize

! j-k plane
totmsgsize(3) = loca_dim(1)*loca_dim(2)
MaxBufLen=max(MaxBufLen,totmsgsize(3))
! i-k plane
totmsgsize(2) = loca_dim(1)*loca_dim(3)
MaxBufLen=max(MaxBufLen,totmsgsize(2))
! i-j plane
totmsgsize(1) = loca_dim(2)*loca_dim(3)
MaxBufLen=max(MaxBufLen,totmsgsize(1))

allocate(fieldSend(1:MaxBufLen))
allocate(fieldRecv(1:MaxBufLen))
```

- At the same time, the halo sizes for the three directions are stored in the integer array totmsgsize.

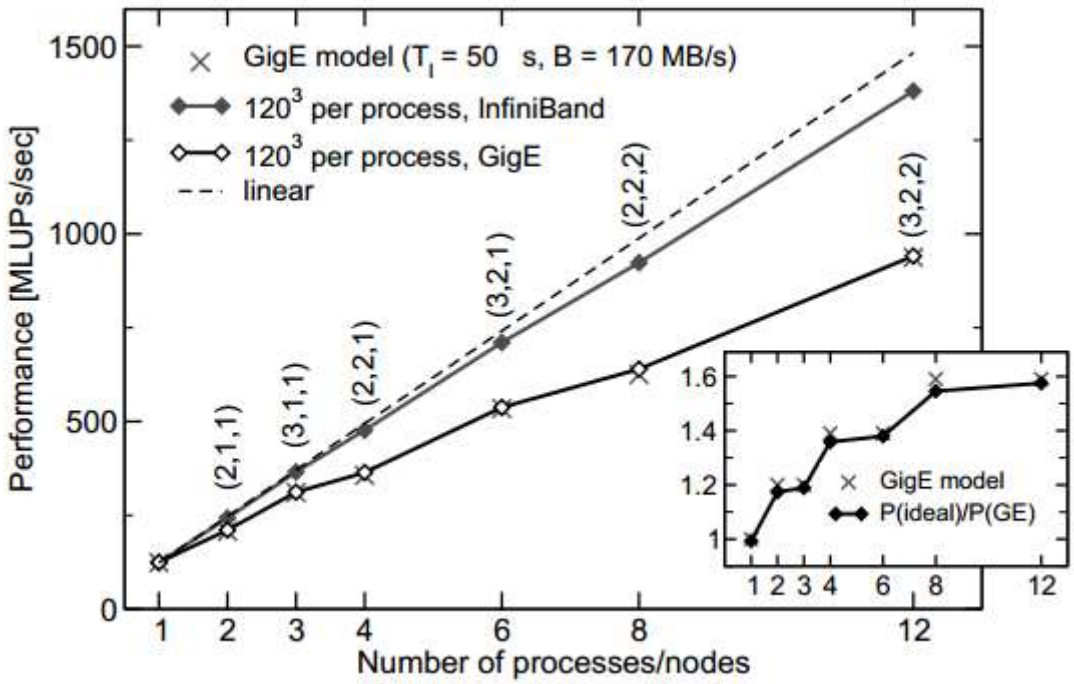
- Now we can start implementing the main iteration loop, whose length is the maximum number of iterations (sweeps), ITERMAX:

```
t0=0 ; t1=1
tag = 0
do iter = 1, ITERMAX
  do disp = -1, 1, 2
    do dir = 1, 3

      call MPI_Cart_shift(GRID_COMM_WORLD, (dir-1), &
                          disp, source, dest, ierr)

      if(source /= MPI_PROC_NULL) then
        call MPI_Irecv(fieldRecv(1), totmsgsize(dir), &
             MPI_DOUBLE_PRECISION, source, &
             tag, GRID_COMM_WORLD, req(1), ierr)
      endif   ! source exists

      if(dest /= MPI_PROC_NULL) then
        call CopySendBuf(phi(iStart, jStart, kStart, t0), &
                         iStart, iEnd, jStart, jEnd, kStart, kEnd, &
                         disp, dir, fieldSend, MaxBufLen)

        call MPI_Send(fieldSend(1), totmsgsize(dir), &
                      MPI_DOUBLE_PRECISION, dest, tag, &
                      GRID_COMM_WORLD, ierr)
      endif   ! destination exists

      if(source /= MPI_PROC_NULL) then
        call MPI_Wait(req, status, ierr)

        call CopyRecvBuf(phi(iStart, jStart, kStart, t0), &
                         iStart, iEnd, jStart, jEnd, kStart, kEnd, &
                         disp, dir, fieldRecv, MaxBufLen)
      endif   ! source exists

    enddo   ! dir
  enddo   ! disp

  call Jacobi_sweep(loca_dim(1), loca_dim(2), loca_dim(3), &
                    phi(iStart, jStart, kStart, 0), t0, t1, &
                    maxdelta)

  call MPI_Allreduce(MPI_IN_PLACE, maxdelta, 1, &
                     MPI_DOUBLE_PRECISION, &
                     MPI_MAX, 0, GRID_COMM_WORLD, ierr)
  if(maxdelta<eps) exit
  tmp=t0; t0=t1; t1=tmp
enddo   ! iter

999 continue
```

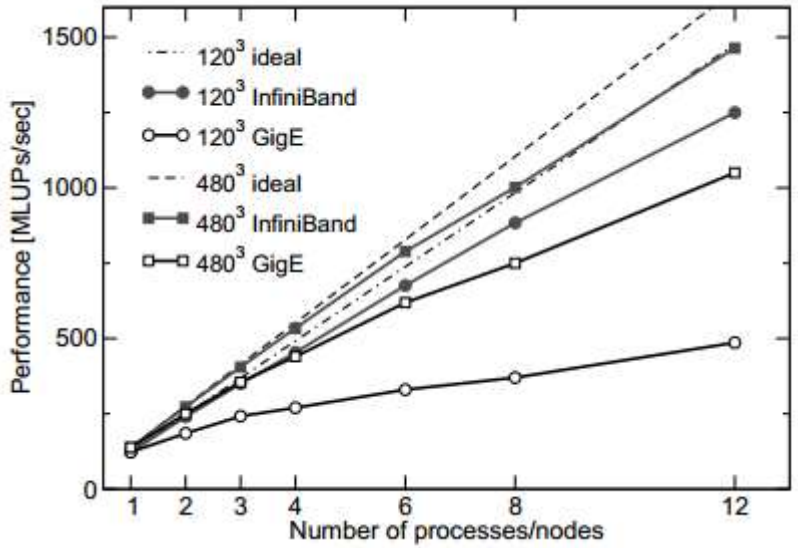
- Halos are exchanged in six steps, i.e., separately per positive and negative Cartesian direction. This is parameterized by the loop variables disp and dir. In line 7, MPI_Cart_shift() is used to determine the communication neighbors along the current direction (source and dest).
- If a subdomain is located at a grid boundary, and periodic boundary conditions are not in place, the neighbor will be reported to have rank MPI_PROC_NULL. MPI calls using this rank as source or destination will return immediately. However, as the copying of halo data to and from the intermediate buffers should be avoided for efficiency in this case, we also mask out any MPI calls, keeping overhead to a minimum (lines 10, 16, and 26).



- Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled “R” (“S”) belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity



- Main panel: **Weak scaling** of the MPI-parallel 3D Jacobi code with problem size 1203 per process on InfiniBand vs. Gigabit Ethernet networks. Only one process per node was used. The domain decomposition topology (number of processes in each Cartesian direction) is indicated. The weak scaling performance model (crosses) can reproduce the GigE data well. Inset: Ratio between ideally scaled performance and Gigabit Ethernet performance vs. process count. (Single-socket cluster based on Intel Xeon 3070 at 2.66 GHz, Intel MPI 3.2.)



- **Strong scaling** of the MPI-parallel 3D Jacobi code with problem size 1203 (circles) and 4803 (squares) on IB (filled symbols) vs. GigE (open symbols) networks. Only one process per node was used. (Same system and MPI topology as in the previous Figure.)

References and More Information

- Blaise Barney, Livermore Computing.
- MPI Standard documents: http://www.mpi-forum.org/docs/
- "Using MPI", Gropp, Lusk and Skjellum. MIT Press, 1994.
- MPI Tutorials: www.mcs.anl.gov/research/projects/mpi/tutorial
- Livermore Computing specific information:
  - Linux Clusters Overview tutorial computing.llnl.gov/tutorials/linux_clusters
  - Using the Sequoia/Vulcan BG/Q Systems tutorial computing.llnl.gov/tutorials/bgq
- "A User's Guide to MPI", Peter S. Pacheco. Department of Mathematics, University of San Francisco.

Thank you!