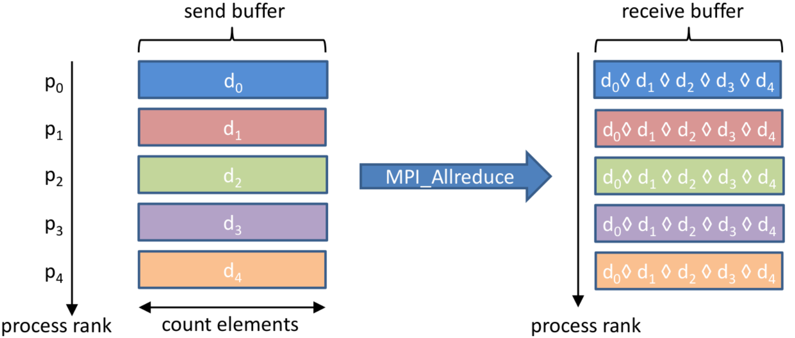
Collective Communication: All Reduce

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

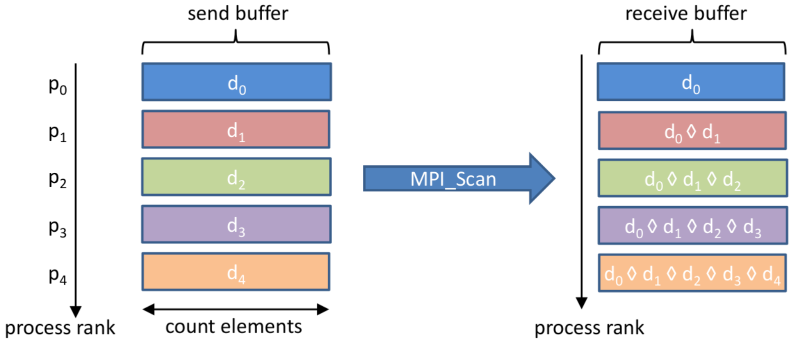
Similar to the reduce operation, but the result is available on every process.



Collective Communication: Scan

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

A scan performs a partial reduction of data, every process has a distinct result



Mat vec

Matrix-vector multiplication

Master : Coordinates the work of others

Slave : does a bit of work

Task : compute A . b

A : double precision (m x n) matrix

b : double precision (n x 1) column matrix

Master algorithm

```
1. Broadcast b to each slave
2. Send 1 row of A to each slave
3. while ( not all m results received ) {
     Receive result from any slave s
     if ( not all m rows sent )
          Send new row to slave s
     else
          Send termination message to s
   }
4. continue
```

Slave algorithm

```
1. Broadcast b (in fact receive b)
2. do {
     Receive message m
      if( m != termination )
          compute result
          send result to master
   } while( m != termination )
3. slave terminates
```

Mat Vec Code: Part-1

```
int main( int argc, char** argv ) {
    int rows = 100, cols = 100;     // dimensions of a
    double **a;
    double *b, *c;
    int master = 0;                // rank of master
    int myid;                      // rank of this process
    int numprocs;                  // number of processes
    // allocate memory for a, b and c
    a = (double**)malloc(rows * sizeof(double*));
    for( int i = 0; i < rows; i++ )
        a[i]=(double*)malloc(cols * sizeof(double));
    b = (double*)malloc(cols * sizeof(double));
    c = (double*)malloc(rows * sizeof(double));
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myid );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    if( myid == master )
        // execute master code
    else
        // execute slave code
    MPI_Finalize();
}
```

Mat Vec Code: Part-2

```
// initialize a and b
for(int j=0;j<cols;j++) {b[j]=1.0; for(int i=0;i<rows;i++) a[i][j]=i;}
// broadcast b to each slave
MPI_Bcast( b, cols, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD );
// send row of a to each slave, tag = row number
int numsent = 0;
for( int i = 0; (i < numprocs-1) && (i < rows); i++ ) {
     MPI_Send(a[i], cols, MPI_DOUBLE_PRECISION, i+1,i,MPI_COMM_WORLD);
     numsent++;
}
for( int i = 0; i < rows; i++ ) {
    MPI_Status status; double ans; int sender;
    MPI_Recv( &ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status );
    c[status.MPI_TAG] = ans;
    sender = status.MPI_SOURCE;
    if ( numsent < rows ) { // send more work if any
        MPI_Send( a[numsent], cols, MPI_DOUBLE_PRECISION,
                  sender, numsent, MPI_COMM_WORLD );
        numsent++;
    } else  // send termination message
        MPI_Send( MPI_BOTTOM, 0, MPI_DOUBLE_PRECISION, sender,
                  rows, MPI_COMM_WORLD );
}
```

Mat Vec Code: Part-3

```
// broadcast b to each slave (receive here)
MPI_Bcast( b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD );
// send row of a to each slave, tag = row number
if( myid <= rows ) {
    double* buffer=(double*)malloc(cols*sizeof(double));
    while (true) {
        MPI_Status status;
        MPI_Recv( buffer, cols, MPI_DOUBLE_PRECISION, master,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        if( status.MPI_TAG != rows ) { // not a termination message
            double ans = 0.0;
            for(int i=0; i < cols; i++)
                ans += buffer[i]*b[i];
            MPI_Send( &ans, 1, MPI_DOUBLE_PRECISION, master,
                      status.MPI_TAG, MPI_COMM_WORLD );
        } else
            break;
    }
} // more processes than rows => no work for some nodes
```

Non blocking Communications

Idea:

- Do something useful while waiting for communications to finish
- Try to overlap communications and computations

How?

- Replace blocking communication by non-blocking variants

`MPI_Send(...)` → `MPI_Isend(..., MPI_Request *request)`

`MPI_Recv(...)` → `MPI_Irecv(..., ~~status~~, MPI_Request *request)`

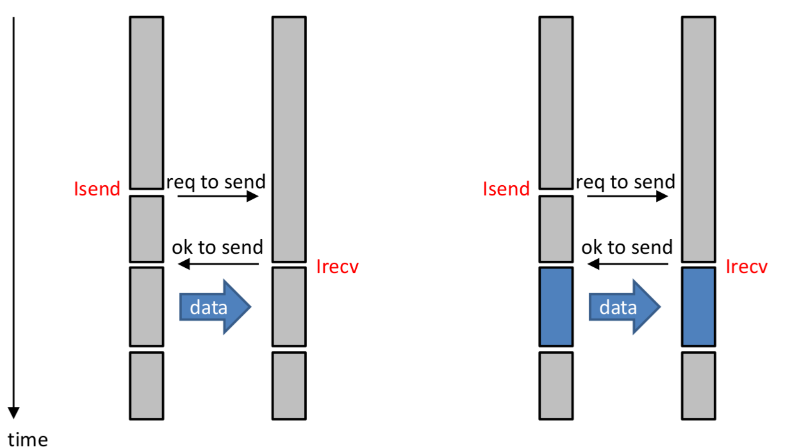
- I = intermediate functions
- `MPI_Isend` and `MPI_Irecv` routines return immediately
- Need polling routines to verify progress
  - **request** handle is used to identify communications
  - **status** field moved to polling routines (see further)

Non blocking Communications

Asynchronous progress

- = ability to progress communications while performing calculations
- Depends on hardware
  - Gigabit Ethernet = very limited
  - Infiniband = much more possibilities
- Depends on MPI implementation
  - Multithreaded implementations of MPI (e.g. Open MPI)
  - Daemon for asynchronous progress (e.g. LAM MPI)
- Depends on protocol
  - Eager protocol
  - Handshake protocol
- Still the subject of ongoing research

Non blocking Communications



a) network interface supports overlapping computations and communications

b) network interface has no such support

Non blocking Communications

Polling / waiting routines

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

`request:` handle to identify communication

`status:` status information (cfr. '*normal*' MPI_Recv)

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )
```

Returns immediately. Sets `flag = true` if communication has completed

```
int MPI_Waitany( int count, MPI_Request *array_of_requests,
                 int *index, MPI_Status *status )
```

Waits for exactly one communication to complete

If more than one communication has completed, it picks a random one

`index` returns the index of completed communication

```
int MPI_Testany( int count, MPI_Request *array_of_requests,
                 int *index, int *flag, MPI_Status *status )
```

Returns immediately. Sets `flag = true` if at least one communication completed

If more than one communication has completed, it picks a random one

`index` returns the index of completed communication

If `flag = false`, `index` returns MPI_UNDEFINED

Non blocking Communications

```
if ( rank != 0 ) {       // client code
    while ( true ) {    // generate requests and send to the server
        generate_request( data, &size );
        MPI_Send( data, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD );
    }
} else {                 // server code (rank == 0)
    MPI_Request *reqList = new MPI_Request[nProc];
    for ( int i = 0; i < nProc - 1; i++ )
        MPI_Irecv( buffer[i].data, MAX_LEN, MPI_CHAR, i+1, tag,
                   MPI_COMM_WORLD, &reqList[i] );
    while ( true ) {             // main consumer loop
        MPI_Status status;
        int reqIndex, recvSize;

        MPI_Waitany( nProc-1, reqList, &reqIndex, &status );
        MPI_Get_count ( &status, MPI_CHAR, &recvSize );
        do_service( buffer[reqIndex].data, recvSize );
        MPI_Irecv( buffer[reqIndex].data, MAX_LEN, MPI_CHAR,
                   status.MPI_SOURCE, tag, MPI_COMM_WORLD,
                   &reqList[reqIndex] );
    }
}
```

Non blocking Communications

```
int MPI_Waitall( int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses )
```

Waits for **all** communications to complete

```
int MPI_Testall ( int count, MPI_Request *array_of_requests,
                  int *flag, MPI_Status *array_of_statuses )
```

Returns immediately. Sets `flag = true` if all communications have completed

```
int MPI_Waitsome ( int incount, MPI_Request * array_of_requests,
                   int *outcount, int *array_of_indices,
                   MPI_Status *array_of_statuses )
```

Waits for **at least one** communications to complete

`outcount` contains the number of communications that have completed

Completed requests are set to `MPI_REQUEST_NULL`

```
int MPI_Testsome ( int incount, MPI_Request * array_of_requests,
                   int *outcount, int *array_of_indices,
                   MPI_Status *array_of_statuses )
```

Same as Waitsome, but returns immediately.

flag field no longer needed, returns **outcount = 0** if no completed communications

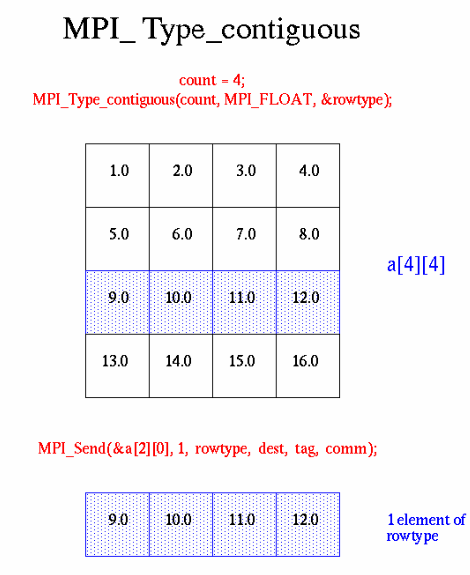
Non blocking Commun.: Exchange from Neighbors

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

main(int argc, char *argv[])  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];   // required variable for non-blocking calls
MPI_Status stats[4];   // required variable for Waitall routine

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

// determine left and right neighbors
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;
```

Non blocking Commun.: Exchange from Neighbors

```
// post non-blocking receives and sends for neighbors
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

   // do some work while sends/receives progress in background

// wait for all non-blocking operations to complete
MPI_Waitall(4, reqs, stats);

   // continue - do more work

MPI_Finalize();
}
```

MPI Datatypes: Contiguous Derived



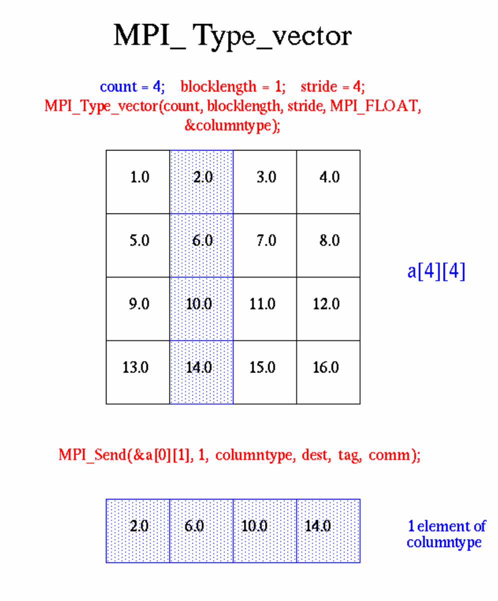
MPI Datatypes: Contiguous Derived

```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);
```

MPI Datatypes: Contiguous Derived

```
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();
}
```

MPI Datatypes: Vector



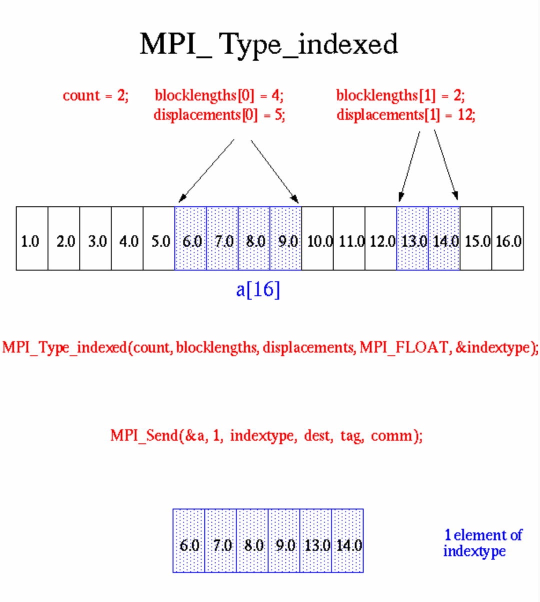
MPI Datatypes: Vector

```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);
```

MPI Datatypes: Vector

```
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();
}
```

MPI Datatypes: Indexed



MPI Datatypes: Indexed

```
#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;
```

MPI Datatypes: Indexed

```
// 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();
}
```