

HIGH-PERFORMANCE COMPUTING IN APPLIED MATHEMATICS

MTH 410/510

MPI_SENDRECV_REPLACE function

C:

```
MPI_Sendrecv_replace(buf, count, datatype, dest, send_tag,  
source, recv_tag, comm, status)
```

Fortran:

```
MPI_SENDRECV_REPLACE(buf, count, datatype, dest, send_tag,  
source, recv_tag, comm, status, ierror)
```

```
x = my_rank
```

```
y = x
```

```
if (my_rank == 0) then
```

```
    call MPI_SENDRECV_REPLACE(y,1,MPI_REAL,1,0,1,0,  
& MPI_COMM_WORLD,status, ierror)
```

```
elseif (my_rank == 1) then
```

```
    call MPI_SENDRECV_REPLACE(y,1,MPI_REAL,0,0,0,0,  
& MPI_COMM_WORLD,status, ierror)
```

```
end if
```

```
print*,'my rank is ',my_rank,'my x=',x,'my y=',y
```

MPI_SENDRECV_REPLACE function

Example: Consider the following problem:

Processors 0 and 1 hold the vector x , initialized locally.

We wish to send the first half of x from process 0 into the second half of x in process 1 and send the second half of x from process 1 into the first half of x in process 0.

MPI_SENDRECV_REPLACE function

Example: Consider the following problem:

Processors 0 and 1 hold the vector x , initialized locally.

We wish to send the first half of x from process 0 into the second half of x in process 1 and send the second half of x from process 1 into the first half of x in process 0.

Solution using MPI_SENDRECV_REPLACE function (assume $x(1:10)$)

```
if (my_rank == 0) then
  call MPI_SENDRECV_REPLACE(x,5,MPI_REAL,1,0,1,0,
    & MPI_COMM_WORLD,status, ierr)
elseif (my_rank == 1) then
  call MPI_SENDRECV_REPLACE(x(6),5,MPI_REAL,0,0,0,0,
    & MPI_COMM_WORLD,status, ierr)
```

See [sendreceive.f](#); [sendreceive1.f](#)

Grouping data for communication

In order to use the MPI_Send, MPI_Recv, MPI_Bcast functions the data items must be stored in *contiguous* memory locations.

Grouping data for communication

In order to use the `MPI_Send`, `MPI_Recv`, `MPI_Bcast` functions the data items must be stored in *contiguous* memory locations.

What if we want to send a column of a matrix from one process to another?

In *C* this may be a problem since data is stored in row-major order.

Grouping data for communication

In order to use the `MPI_Send`, `MPI_Recv`, `MPI_Bcast` functions the data items must be stored in *contiguous* memory locations.

What if we want to send a column of a matrix from one process to another?

In *C* this may be a problem since data is stored in row-major order.

What if we want to send a row of a matrix from one process to another?

In *FORTRAN* this may be a problem since data is stored in column-major order.

Grouping data for communication

In order to use the `MPI_Send`, `MPI_Recv`, `MPI_Bcast` functions the data items must be stored in *contiguous* memory locations.

What if we want to send a column of a matrix from one process to another?

In *C* this may be a problem since data is stored in row-major order.

What if we want to send a row of a matrix from one process to another?

In *FORTRAN* this may be a problem since data is stored in column-major order.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Laid-out in C memory storage

$$[1 \ 2 \ 3 \ 4 \ 5 \ 6]$$

Laid-out in FORTRAN storage (also Matlab)

$$[1 \ 4 \ 2 \ 5 \ 3 \ 6]$$

Let

$$x = [0 \ 0 \ 0 \ 0]$$

$$y = [1 \ 1 \ 1 \ 1]$$

and assume that process 0 holds x and process 1 holds y .

Let

$$x = [0 \ 0 \ 0 \ 0]$$

$$y = [1 \ 1 \ 1 \ 1]$$

and assume that process 0 holds x and process 1 holds y .

The challenge:

Let

$$x = [0 \ 0 \ 0 \ 0]$$

$$y = [1 \ 1 \ 1 \ 1]$$

and assume that process 0 holds x and process 1 holds y .

The challenge: interchange data from process 0 to process 1 such that the entries of x are interlaced with the entries of y

$$newx = [0 \ 1 \ 0 \ 1]$$

$$newy = [1 \ 0 \ 1 \ 0]$$

This may be a problem in both *C* and *FORTRAN*.

May use a sequence of send/receive calls

Let

$$x = [0 \ 0 \ 0 \ 0]$$

$$y = [1 \ 1 \ 1 \ 1]$$

and assume that process 0 holds x and process 1 holds y .

The challenge: interchange data from process 0 to process 1 such that the entries of x are interlaced with the entries of y

$$newx = [0 \ 1 \ 0 \ 1]$$

$$newy = [1 \ 0 \ 1 \ 0]$$

This may be a problem in both *C* and *FORTRAN*.

May use a sequence of send/receive calls - not convenient

What if we want to send **a package** (number, char, vector, matrix) ?

MPI provides solutions to these problems by using *derived datatypes* and *MPI_Pack*, *MPI_Unpack* functions.

Grouping data for communication: MPI_Type_vector

C:

```
int MPI_Type_vector(count, block_length, stride, element_type,  
                  new_mpi_t)
```

FORTRAN:

```
MPI_Type_vector(count, block_length, stride, element_type,  
              new_mpi_t, ierr)
```

count - integer, number of elements in the type

block_length - integer, number of entries in each element

stride - integer, number of elements of type *element_type* between successive elements of *new_mpi_t*

element_type - MPI_Datatype type of the elements composing the derived type (e.g., MPI_FLOAT)

new_mpi_t - the MPI type of the new derived data type

After the call to `MPI_Type_vector`, we can't use `new_mpi_t` in communication functions until we call `MPI_Type_commit`.

Example: Create a datatype vector of 5 elements, each separated by 2 memory locations, of type real:

C:
`MPI_Type_vector(5,1,2,MPI_FLOAT,&stride_mpi_t);`
`MPI_Type_commit(&stride_mpi_t);`

FORTRAN:
call `MPI_TYPE_VECTOR(5,1,2,MPI_REAL,stride_mpi_t,ierror)`
call `MPI_TYPE_COMMIT(stride_mpi_t,ierror)`

Example: `newmpitype.f`; `newmpitype1.f`; `send_row.f`; `send_col_to_row.f`

The challenge: Send the lower/upper triangular portion of a square matrix stored in process 0 to process 1.

The challenge: Send the lower/upper triangular portion of a square matrix stored in process 0 to process 1.

Notice that the `MPI_Type_vector` will not work here because the block lengths are different for each row.

The MPI solution: `MPI_Type_indexed`

C:

```
int MPI_Type_indexed(count, block_lengths[ ], displacements[ ],  
                    old_type, new_mpi_t)
```

FORTRAN:

```
MPI_Type_indexed(count, block_lengths[ ], displacements[ ],  
                old_type, new_mpi_t, ierr)
```

The derived type `new_mpi_t` consists of `count` elements (integer) of type `old_type` (e.g., `MPI_REAL`). The i^{th} element consists of `block_lengths[i]` entries, and it is displaced `displacements[i]` units of `old_type` from the beginning (displacement 0) of the `type`.

See `send_triangle.f`

MPI_Type_vector revisited

C:

```
int MPI_Type_vector(count, block_length, stride, element_type,  
                   new_mpi_t)
```

FORTRAN:

```
MPI_Type_vector(count, block_length, stride, element_type,  
               new_mpi_t, ierr)
```

count - integer, number of elements in the type

block_length - integer, number of entries in each element

stride - integer, number of elements of type *element_type* between successive elements of *new_mpi_t*

element_type - MPI_Datatype type of the elements composing the derived type (e.g., MPI_FLOAT)

new_mpi_t - the MPI type of the new derived data type

call MPI_TYPE_COMMIT(stride_mpi_t)

Challenge: Given two matrices \mathbf{A} and \mathbf{B} , of the same size, interchange the elements of \mathbf{A} and \mathbf{B} such that in the new matrix \mathbf{A} the entries of old \mathbf{A} are interlaced with the entries of \mathbf{B} and in the new matrix \mathbf{B} the elements of old \mathbf{B} interlaced with the entries of \mathbf{A}

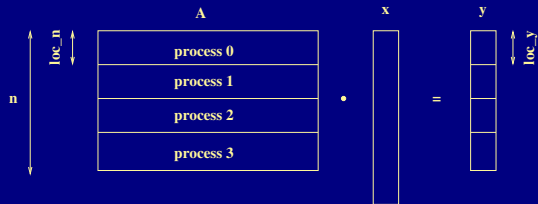
Challenge: Given two matrices **A** and **B**, of the same size, interchange the elements of **A** and **B** such that in the new matrix **A** the entries of old **A** are interlaced with the entries of **B** and in the new matrix **B** the elements of old **B** interlaced with the entries of **A**

Solution:

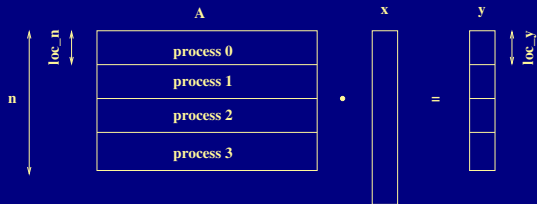
```
...  
    call MPI_TYPE_VECTOR(size, 1, 2, MPI_REAL, stride_vect, ierr)  
    call MPI_TYPE_COMMIT( stride_vect, ierr)  
  
...  
    call MPI_SENDRECV_REPLACE(x,1,stride_vect,1,0,1,0,  
&                             MPI_COMM_WORLD,status, ierr)
```

See redblack.f

Parallel matrix vector product



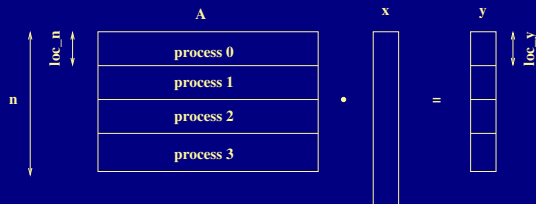
Parallel matrix vector product



Serial algorithm:

```
y=0
for i=1:n
    for j=1:m
        y(i) = y(i) + A(i,j)*x(j)
    end
end
```

Parallel matrix vector product



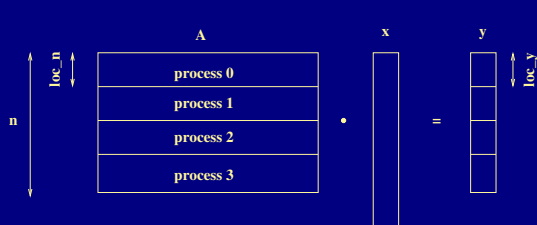
Serial algorithm:

```
y=0
for i=1:n
    for j=1:m
        y(i) = y(i) + A(i,j)*x(j)
    end
end
```

Parallelization:

- Process 0 block-distributes the rows of A among processes
- Process 0 broadcasts x
- Each process computes locally $loc_y = LOC_A * x$
- Process 0 gathers loc_y into y

Parallel matrix vector product



Serial algorithm:

```
y=0
for i=1:n
  for j=1:m
    y(i) = y(i) + A(i,j)*x(j)
  end
end
```

Parallelization:

- Process 0 block-distributes the rows of A among processes
- Process 0 broadcasts x
- Each process computes locally $loc_y = LOC_A * x$
- Process 0 gathers loc_y into y

In C we can directly scatter the rows of the matrix A among processes
In FORTRAN we need to deal with noncontiguous data location

```

....
real A(1:n,1:n), x(n), y(n)
real LOC_A(1:loc_n,1:n), loc_y(loc_n)

call MPI_TYPE_VECTOR(n, loc_n, n, MPI_REAL, stride_vect,
&                    ierror)
call MPI_TYPE_COMMIT( stride_vect, ierror)

....
if (my_rank == 0) then
    do i=0,np-1
        call MPI_SEND(A(i*loc_n+1,1), 1, stride_vect, i,
&                    0,MPI_COMM_WORLD, ierror)
    end do
end if
call MPI_RECV(LOC_A, n*loc_n, MPI_REAL, 0,
&            0, MPI_COMM_WORLD, status, ierror)
...

```

See [pmatvect.f](#)

General derived datatypes: MPI_Type_struct

C syntax:

```
int MPI_Type_struct(count, block_lengths[ ], displacements[ ],  
                  typelist[ ], new_mpi_type)
```

FORTRAN syntax:

```
call MPI_TYPE_STRUCT(count, block_lengths[ ], displacements[ ],  
                    typelist[ ], new_mpi_type, ierr)
```

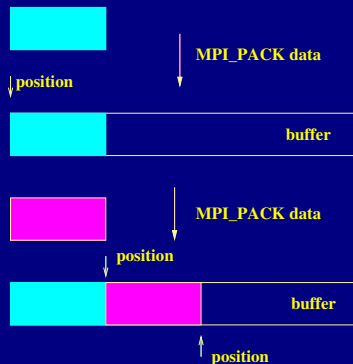
Argument list:

- *count* - the number of blocks of elements in the derived type
- *block_lengths*[] - array containing the number of entries in each element type.
- *displacements*[] - array containing the displacement of each element from the beginning of the message
- *typelist*[] - array containing the datatype of each entry

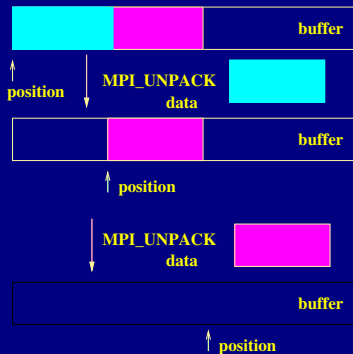
See [get_data3.c](#), [get_data3.f](#)

Packing and unpacking data using MPI_Pack/MPI_Unpack

MPI_PACK



MPI_UNPACK



See `get_data4.f`

Numerical solution to Poisson's equation

One of the most important applications of the iterative methods for linear systems is obtain the numerical solutions of partial differential equations that are discretized using finite differences or finite element methods.

To motivate our study, consider the 2D Poisson's equation on a square: the goal is to find a function

$$u(x, y), \quad 0 \leq x \leq 1, \quad 0 \leq y \leq 1$$

that satisfies the partial differential equation

$$u_{xx}(x, y) + u_{yy}(x, y) = f(x, y), \quad 0 < x, y < 1 \quad (1)$$

with boundary conditions

$$u(x, y) = g(x, y), \quad (x, y) \text{ boundary point}$$

The functions $f(x, y)$ and $g(x, y)$ are given, we must find $u(x, y)$.

For $n > 1$ we define $h = 1/n$ and the mesh

$$(x_j, y_k) = (j * h, k * h), \quad 0 \leq j, k \leq n$$

These are called the *grid points* or *mesh points*. A discrete problem is obtained by approximating the second order derivatives using the central difference formula:

$$G''(x) \approx G(x + h) - 2G(x) + G(x - h) \quad (2)$$

which is a second order accurate approximation (error of order h^2). When (2) is used to approximate u_{xx} and u_{yy} at each *interior* grid point we obtain a linear system of equations of dimension $(n - 1)^2$

$$4u_{j,k} - u_{j+1,k} - u_{j-1,k} - u_{j,k-1} - u_{j,k+1} = -h^2 f_{j,k}, \quad 1 \leq j, k \leq n - 1 \quad (3)$$

Poisson's equation in 1D

The matrix of the linear system and the stencil of the central difference discretization in 1D are

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix},$$

