MPI: Numerical Integration, P2P and Collective Communication

Kameswararao Anupindi

Department of Mechanical Engineering
Indian Institute of Technology Madras (IITM)

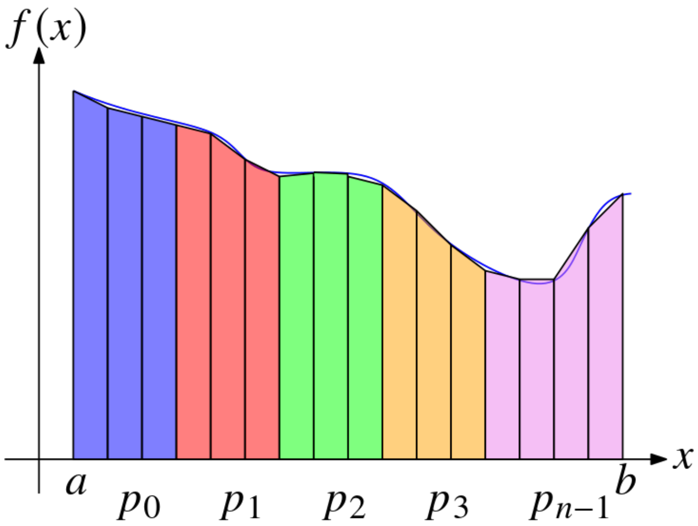
March, 2024

A few potential pitfalls of MPI_Send/MPI_Recv

Process A	Process B
x	MPI_Recv
MPI_Send	x

- Non-matching tags
- Rank of the destination process is **the same** as that of the source.

The Trapezoidal Rule approximation



$$\int_a^b f(x)dx = \frac{h}{2}\left[f(x_0) + f(x_n) + 2\left(f(x_1) + f(x_2)\ldots + f(x_{n-1})\right)\right] \tag{1}$$

The Trapezoidal Rule using MPI in C

```
/* MPI parallel version of trapezoidal rule */
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<mpi.h>

#define PI 3.14159265358

double func(double x)
{
  return (1.0 + sin(x));
}

double trapezoidal_rule(double la, double lb, double ln, double h)
{
  double total;
  double x;
  int i;

  total = (func(la) + func(lb))/2.0;
  for(i = 1; i <= ln-1; i++) /* sharing the work, use only local_n */
    {
      x = la + i*h;
      total += func(x);
    }
  total = total * h;

  return total;                  /* total for each thread, private */
}
```

The Trapezoidal Rule using MPI in C contd...

```c
int main(int argc, char* argv[])
{
  double a, b, final_result, la, lb, lsum, h;
  int myid, nprocs, proc;
  int n, ln;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);   /* myrank of the process */
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs); /* size of the communicator */

  n = 1024;                       /* number of trapezoids.. */
  a = 0.0;
  b = PI;                         /* hard-coded.. */
  final_result = 0.0;

  h = (b-a)/n;
  ln = n/nprocs;                  /* nprocs evenly divides number of trapezoids */

  la = a + myid*ln*h;
  lb = la + ln*h;
  lsum = trapezoidal_rule(la, lb, ln, h); /* every process calls this function... */
```

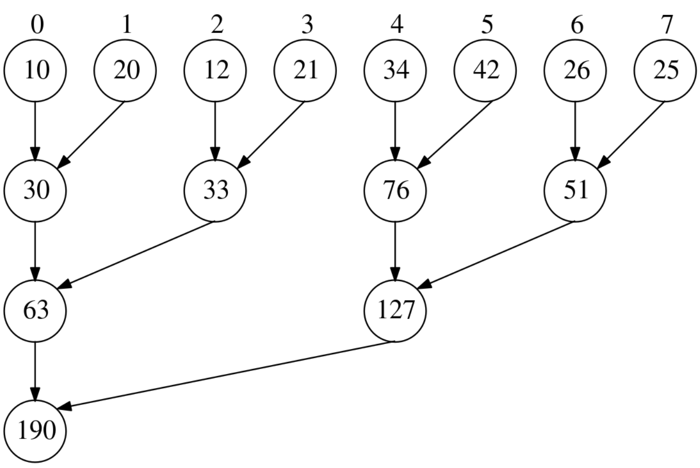

The Trapezoidal Rule using MPI in C contd...

```c
  if (myid != 0)
    {
      MPI_Send(&lsum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
  else                            /* process 0 */
    {
      final_result = lsum;
      for (proc = 1; proc < nprocs; proc++)
        {
          MPI_Recv(&lsum, 1, MPI_DOUBLE, proc, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          final_result += lsum;
        }
    }

  if (myid == 0)                  /* output is only printed by process 0 */
    {
      printf("\n The area under the curve (1+sin(x)) between 0 to PI is equal to %lf \n\n", final_result);
    }

  MPI_Finalize();
  return 0;
}
```

The Trapezoidal Rule - Enhancements - Dealing with input and output

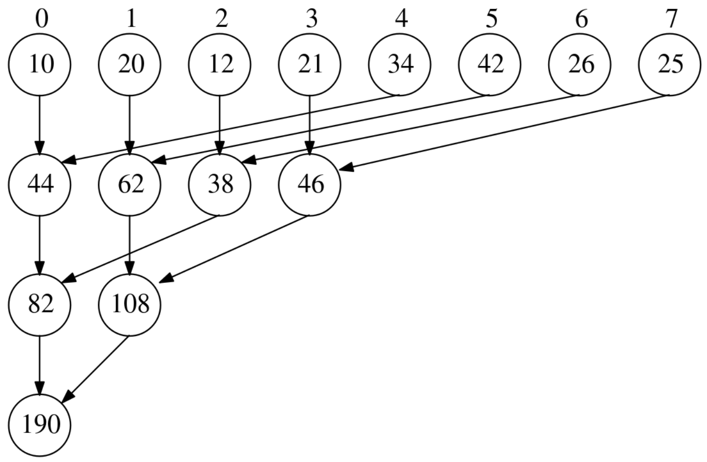
```
if (myid == 0)
  {
    printf("\n Enter the lower limit, upper limit and n");
    scanf(&a, &b, &n);

    for (proc = 1, proc<nprocs; proc++)
      {
        MPI_Send(&a, ....);
        MPI_Send(&b, ....);
        MPI_Send(&n, ....);
      }
  }
else
  {
    MPI_Recv(&a, 1, ...);
    MPI_Recv(&b, 1, ...);
    MPI_Recv(&n, 1, ...);
  }



  return 0;
}
```


The Trapezoidal Rule - Enhancements - Calculating global sum



- Original sum: 7 receives and adds
- Tree sum: 3 receives and adds
- if nprocs = 1024, tree sum would do only 10 receives and adds

The Trapezoidal Rule - Calculating global sum - another way



- Several possibilities exist
- A method works best for small trees, and another for large trees!
- A method may work best for system A, and another for system B.
- MPI provides a **global sum** that works the best in the form of **Collective Communication**.

Collective Communication - MPI_Reduce

```
int MPI_Reduce(
               void* input_data_p,
               void* output_data_p,
               int count,
               MPI_Datatype datatype,
               MPI_Op operator,
               int root,
               MPI_Comm communicator);
```

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
TYPE(*), DIMENSION(:), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(:) :: recvbuf
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

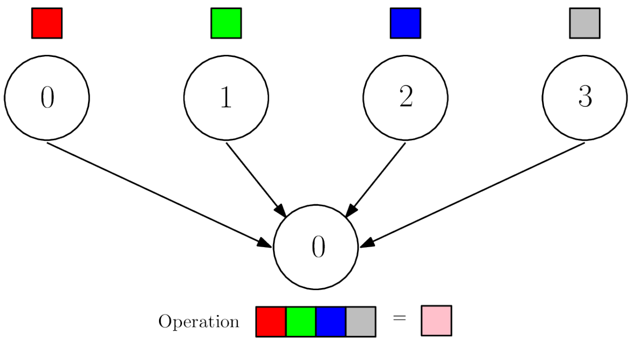
Collective Communication - MPI_Reduce

```
MPI_Reduce(&lsum, &final_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
call MPI_Reduce(lsum, final_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD, mpierror);
```

MPI_MAX	MPI_LOR
MPI_MIN	MPI_BAND
MPI_SUM	MPI_BOR
MPI_PROD	MPI_MAXLOC
MPI_LAND	MPI_MINLOC

Collective communication: Reduce




```
MPI_Reduce(
          void *send_buffer,
          void *receive_buffer,
          int count,
          MPI_Datatype datatype,
          MPI_Op operator,
          int root,
          MPI_Comm communicator)
```

Difference between Collective and P2P communications

- All the processes must call the same MPI Collective Communication (CC)
- The arguments passed by each process to MPI CC must be *compatible*
- All processes must supply an output_data_p, although this is needed only on *root*
- While P2P are matched using *communicator* and *tags*, MPI CC are matched solely on the basis of *communicator* and order of calling.

Multiple CC calls

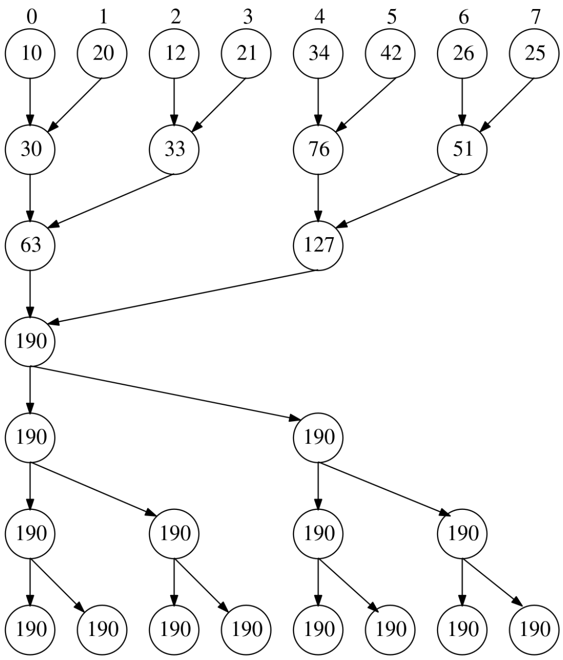
```
Process 0:
  a = 1, b = 0, c = 2, d = 0;
  dest_process = 0;

  MPI_Reduce(&a, &b, ..., 0, comm);
  MPI_Reduce(&c, &d, ..., 0, comm);

Process 1:
  a = 1, b = 0, c = 2, d = 0;
  dest_process = 0;

  MPI_Reduce(&c, &d, ..., 0, comm);
  MPI_Reduce(&a, &b, ..., 0, comm);

Process 2:
  a = 1, b = 0, c = 2, d = 0;
  dest_process = 0;

  MPI_Reduce(&a, &b, ..., 0, comm);
  MPI_Reduce(&c, &d, ..., 0, comm);
```

Reduction on the same variable

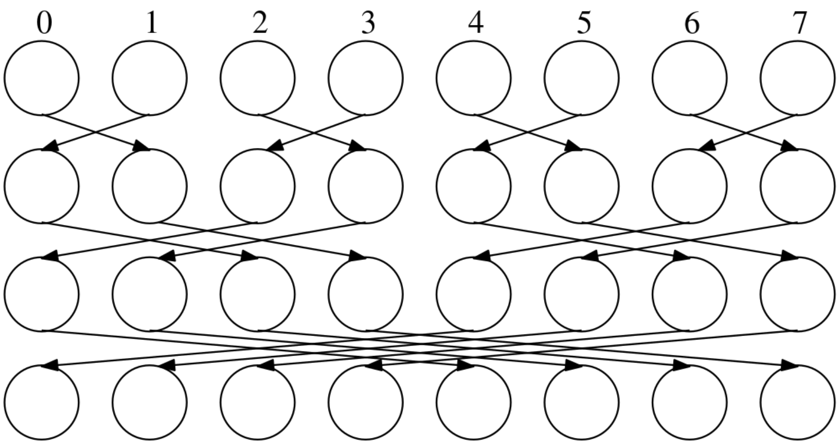
```
MPI_Reduce(&x, &x, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

- Illegal in MPI
- Produces unpredictable result.

MPI_Allreduce: Tree and Reverse-tree



MPI_Allreduce: Butterfly

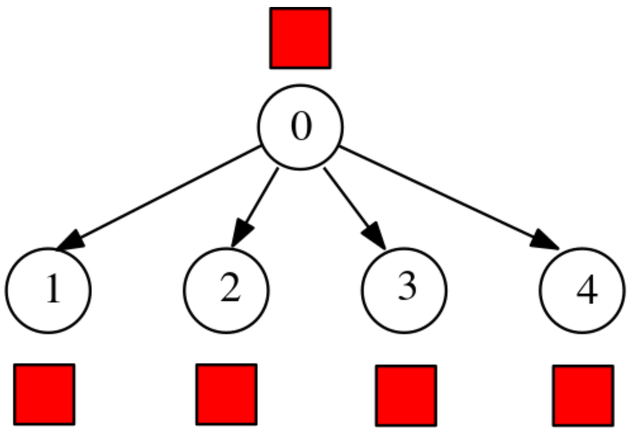


MPI_Allreduce function prototype

```
int MPI_Allreduce(
                void* input_data_p,
                void* output_data_p,
                int count,
                MPI_Datatype datatype,
                MPI_Op operator,
                MPI_Comm communicator);
```

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierror)
TYPE(*), DIMENSION(:), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(:) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

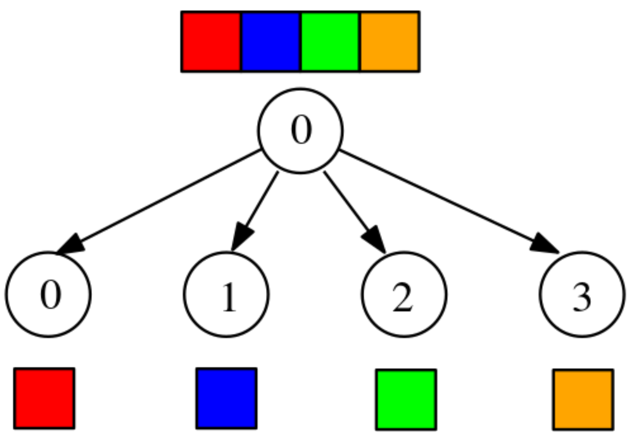
Collective communication: Broadcast




```
MPI_Bcast(
          void *data,
          int count,
          MPI_Datatype datatype,
          int root,
          MPI_Comm communicator)
```

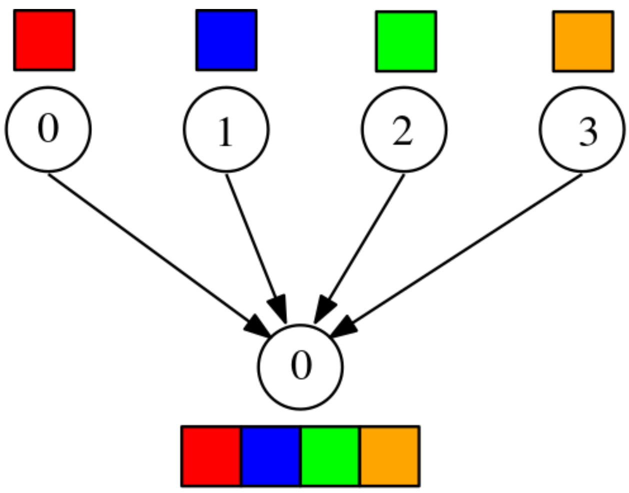
- Use a tree-structured communication instead!
- data_p is an input argument on root (send_proc) and output on the other processes.

Collective communication: Scatter




```
MPI_Scatter(
          void *send_data,
          int send_count,
          MPI_Datatype datatype,
          void *receive_data,
          int receive_count,
          MPI_Datatype datatype,
          int root,
          MPI_Comm communicator)
```

Collective communication: Gather




```
MPI_Gather(
          void *send_data,
          int send_count,
          MPI_Datatype datatype,
          void *receive_data,
          int receive_count,
          MPI_Datatype datatype,
          int root,
          MPI_Comm communicator)
```


Broadcast example program

```
if (myid == 0)
  buf = 327;


MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (myid == 0)
  printf("\n Broadcasted values on processors are:\n");

printf("\t (%d, %d)\n", myid, buf);
```

Gather example program

```
int send_buf, *recv_buf;
if (myid == 0)
  {
    recv_buf = (int *)malloc(size*sizeof(int));
  }


send_buf = 100+myid*myid;


MPI_Gather(&send_buf, 1, MPI_INT, recv_buf, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (myid == 0)
  {
    printf("\n Received values on host process are:\n");
    for(i=0; i<size; i++)
      printf("\t %d", recv_buf[i]);
    printf("\n");
  }

if (myid == 0)
  free(recv_buf);
```


Scatter example program

```
int *send_buf, recv_buf;
if (myid == 0)
  {
    send_buf = (int *)malloc(size*sizeof(int));
    for(i=0; i<size; i++)
      send_buf[i] = 100+i*5+i;
  }


MPI_Scatter(send_buf, 1, MPI_INT, &recv_buf, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (myid == 0)
  printf("\n Received values on processors are:\n");


printf("\t (%d, %d)", myid, recv_buf);

if (myid == 0)
  free(send_buf);
```

Matrix - Vector Multiplication using block decomposition

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- Row block-decomposition
- Consider

$$\begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T$$

  has block-decomposition as well
- How to arrange that each process has access to all components of $[x]$?

MPI_Allgather

```
int MPI_Allgather(
                  const void* send_buf_p,
                  int send_count,
                  MPI_Datatype send_type,
                  void* recv_buf_p,
                  int recv_count,
                  MPI_Datatype recv_type,
                  MPI_Comm communicator);
```

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Motivation for MPI Derived Datatypes

```
double x[1000];

if (myid == 0)
  for(i = 0; i < 1000; i++)
    MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else
  for(i = 0; i < 1000; i++)
    MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);
```

```
double x[1000];

if (myid == 0)
  MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else
  MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```


MPI Data consolidation

- The **count** argument
- MPI Derived Datatypes
- MPI_Pack and MPI_Unpack

MPI Derived Datatypes

- Collection of data items in memory by storing both **type** and **relative memory locations**

Variable	Address
a	24
b	40
c	48

```
{(MPI_DOUBLE,0), (MPI_DOUBLE,16), (MPI_DOUBLE,24)}
```


Building MPI Derived Datatypes

```
int MPI_Type_create_struct(
             int             count,
             int             array_of_blocklengths[],
             MPI_Aint        array_of_displacements[],
             MPI_Datatype    array_of_types[],
             MPI_Datatype*   new_type_p);

int array_of_blocklengths[3] = {1, 1, 1};

array_of_blocklengths[0] = 5;

array_of_displacements[] = {0, 16, 24};
```

Building MPI Derived Datatypes contd...

```
int MPI_Get_address(
    void*       location_p,
    MPI_Aint*   address_p);

MPI_Aint a_addr, b_addr, c_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;

MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr;

MPI_Get_address(&c, &c_addr);
array_of_displacements[2] = c_addr - a_addr;
```

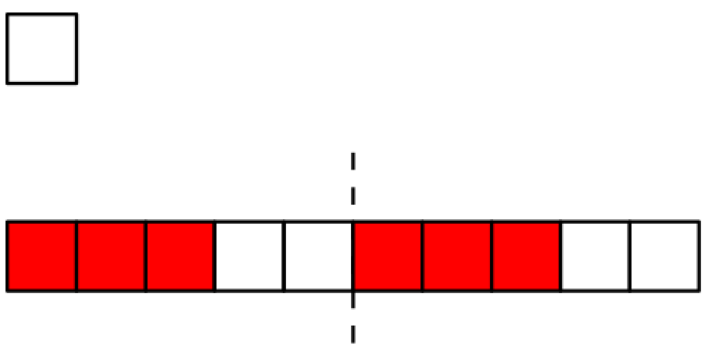

Building MPI Derived Datatypes contd...

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT);
....
....
MPI_Datatype new_mpi_t;
...
MPI_Type_create_struct(3, array_of_blocklengths,
   array_of_displacements, array_of_types, &new_mpi_t);
...
int MPI_Type_commit(MPI_Datatype* new_mpi_type_p);
...
...
MPI_Bcast(&a, 1, new_mpi_t, 0, comm);
...
...
int MPI_Type_free(MPI_Datatype* old_mpi_type_p);
```

Other Derived Data Types: Contiguous data

```
int MPI_Type_contiguous(
        int               count,
        MPI_Datatype      oldtype,
        MPI_Datatype*      newtype);

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierror)
```


Other Derived Data Types: Vector data



- blocklength =
- count =
- stride =

Where would such a pattern of values that has blocks and gaps is needed?

Other Derived Data Types: Vector data

```
int MPI_Type_vector(
        int             count,
        int             blocklength,
        int             stride,
        MPI_Datatype    oldtype,
        MPI_Datatype*    newtype);

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype, ierror)
```

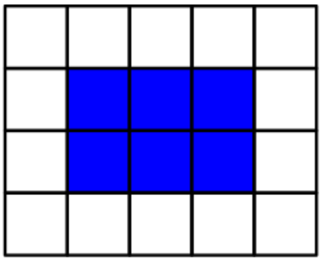

Representing 2D arrays in C

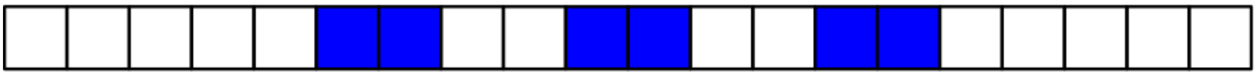
$$A = \begin{bmatrix} a[0][3] & \cdots & \cdots & a[3][3] \\ a[0][2] & \cdots & \cdots & \cdots \\ a[0][1] & a[1][1] & \cdots & \cdots \\ a[0][0] & a[1][0] & a[2][0] & a[3][0] \end{bmatrix}$$

$$A = \begin{bmatrix} 4 & \cdots & \cdots & 16 \\ 3 & \cdots & \cdots & \cdots \\ 2 & 6 & \cdots & \cdots \\ 1 & 5 & 9 & 13 \end{bmatrix}$$

1	2	3	4	5											16

Extracting a sub-array in C





- blocklength =
- count =
- stride =

Sending a sub-array

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

MPI_Send(&x[1][1], 1, &newtype, ...)
```

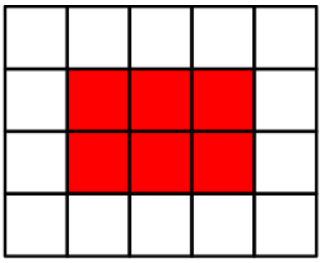
Representing 2D arrays in FORTRAN

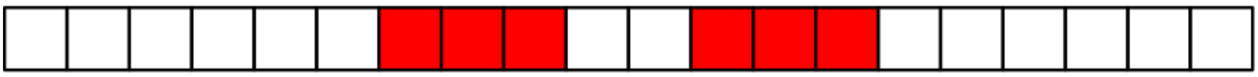
$$A = \begin{bmatrix} a[1][4] & \cdots & \cdots & a[4][4] \\ a[1][3] & \cdots & \cdots & \cdots \\ a[1][2] & a[2][2] & \cdots & \cdots \\ a[1][1] & a[2][1] & a[3][1] & a[4][1] \end{bmatrix}$$

$$A = \begin{bmatrix} 13 & \cdots & \cdots & 16 \\ 9 & \cdots & \cdots & \cdots \\ 5 & 6 & \cdots & \cdots \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

1	2	3	4	5											16

Extracting a sub-array in FORTRAN





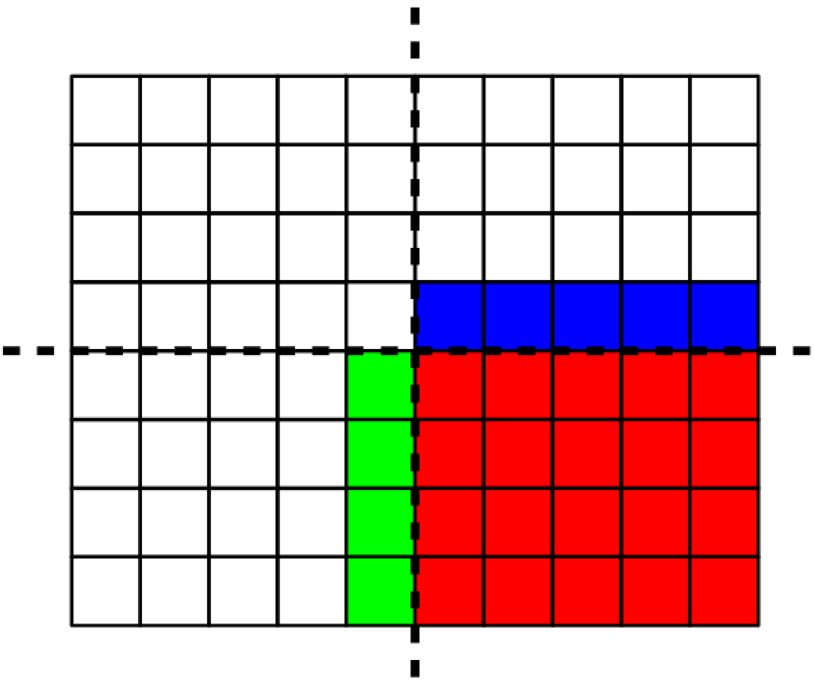
- blocklength =
- count =
- stride =

Sending a sub-array

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

MPI_Send(&x[2][2], 1, &newtype, ...)
```

Remember to commit the new type!

2D or 3D Jacobi/GS



Bubble Sort

```
void Bubble_sort(int a[], int n,)
{
int list_length, i, temp;

for (list_length = n; list_length >= 2; list_length--)
  for (i = 0; i < list_length-1; i++)
    if (a[i] > a[i+1])
      {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
      }
}
```


Odd-Even Transposition Sort

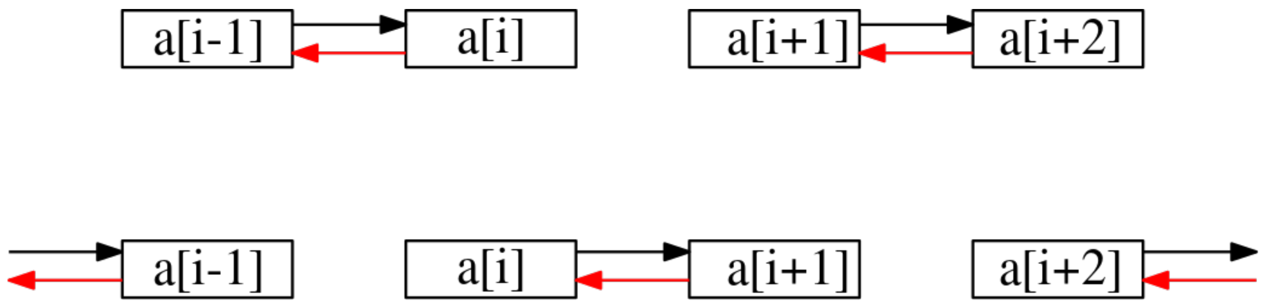
```
for (pass = 0; pass < n; pass++)
    if (pass%2 == 0) {
      for (i = 1; i < n; i += 2)
          if (a[i-1] > a[i]) {
              temp = a[i];
              a[i] = a[i-1];
              a[i-1] = temp;
              }
    }else {
      for (i = 1; i < n-1; i += 2)
          if (a[i] > a[i+1]) {
              temp = a[i];
              a[i] = a[i+1];
              a[i+1] = temp;
              }
    }
```

Odd-Even Transposition Sort contd...

```
Even-pass: (a[0], a[1]), (a[2], a[3]), (a[4], a[5]), ...

Odd-pass:         (a[1], a[2]), (a[3], a[4]), (a[5], a[6]), ...
```

```
Given list:  5, 9, 4, 3
Even-pass:  (5, 9),  (4, 3) -> 5, 9, 3, 4
Odd-pass:   5, (9, 3), 4    -> 5, 3, 9, 4
Even-pass:  (5, 3), (9, 4)  -> 3, 5, 4, 9
Odd-pass:   3, (5, 4), 9    -> 3, 4, 5, 9
```


Parallel Odd-Even Transposition sort for $n = p$



Parallel Odd-Even Transposition sort for $n >> p$

	Process 0	Process 1	Process 2	Process 3
Given	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After local sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Parallel Odd-Even Transposition Sort – Algorithm

```
Sort local elements;
for (pass = 0; pass < comm_sz; pass++) {
    partner = compute_partner(pass, my_rank);
    if (I am active) {
        Send my elements to partner;
        Receive elements from partner;
        if (my_rank < partner)
          Keep smaller elements;
        else
          Keep larger elements;
   }
}
```

Safety in MPI programs

```
MPI_Send(my_elements, n/p, MPI INT, partner, 0, comm);
MPI_Recv(temp_elements, n/p, MPI INT, partner, 0, comm, &status);
```

- A program that relies on MPI-provided buffering is **unsafe**
- How can we tell if a program is unsafe?
- How to modify the communication to make it safe?

How can we tell if a program is safe?

```
MPI_Send --> MPI_Ssend
```

```
MPI_Ssend (
          void*          message_buffer_p,
          int            message_size,
          MPI_Datatype   message_type,
          int            dest_process,
          int            tag,
          MPI_Comm       communicator);
```

How to modify the communication to make it safe?

```
MPI_Send(msg, size, MPI_INT, (myid+1)%p, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (myid+p-1)%p, 0, comm, &status);
```

```
if (myid % 2 == 0){
  MPI_Send(msg, size, MPI_INT, (myid+1)%p, 0, comm);
  MPI_Recv(new_msg, size, MPI_INT, (myid+p-1)%p, 0, comm, &status);
}
else{
  MPI_Recv(new_msg, size, MPI_INT, (myid+p-1)%p, 0, comm, &status);
  MPI_Send(msg, size, MPI_INT, (myid+1)%p, 0, comm);
}
```


MPI alternative to manual scheduling

```
int MPI_Sendrecv(
          void*          send_buf_p,
          int            send_buf_size,
          MPI_Datatype   send_buf_type,
          int            dest,
          int            send_tag,
          void*          recv_buf_p,
          int            recv_buf_size,
          MPI_Datatype   recv_buf_type,
          int            source,
          int            recv_tag,
          MPI_Comm       communicator,
          MPI_Status*    status_p);
```

For the same send/recv buffers

```
int MPI_Sendrecv_replace(
          void*         buf_p,
          int           buf_size,
          MPI_Datatype  buf_type,
          int           dest,
          int           send_tag,
          int           source,
          int           recv_tag,
          MPI_Comm      communicator,
          MPI_Status*   status_p);
```