Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

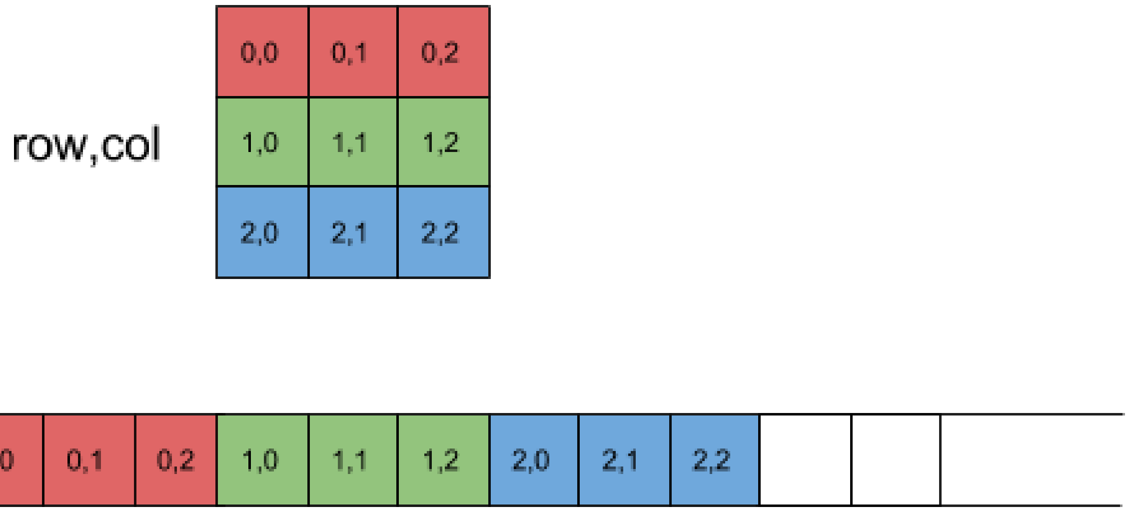
Recap

- How to measure performance:
  - A barrier at the beginning of the application, to be sure everyone starts at the same time
  - Report the maximum runtime across the ranks
  - Execute your application multiple times and report the distribution of timings
- Strong vs. weak scaling
- Amdahl’s Law and Gustafson’s Law

Q&A

Gather/Broadcast on matrices

int matrix[3][3];

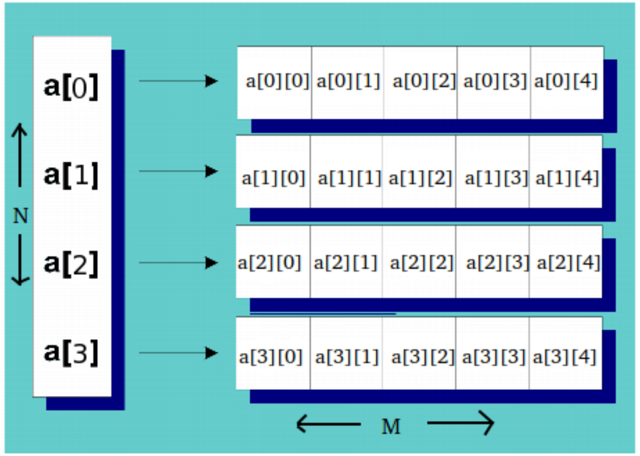



```
MPI_Reduce(sendbuf, recvbuf, num_rows*num_cols,
                MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Gather/Broadcast on dynamically allocated matrices

```
int** a;
a = (int**) malloc(sizeof(int*)*num_rows);
for(int i = 0; i < num_rows; i++){
        a[i] = (int*) malloc(sizeof(int)*num_cols);
}
```




```
MPI_Reduce(a, recvbuf, num_rows*num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
MPI_Reduce(a[0], recvbuf, num_rows*num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

WRONG!

CORRECT

```
for(int i = 0; i < num_rows; i++){
    MPI_Reduce(a[i], recvbuf[i], num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Gather/Broadcast on dynamically allocated matrices

```
int* a;
a = (int*) malloc(sizeof(int)*num_rows*num_cols);
...
...
// a[i][j]
a[i * num_cols + j] = ....
```

CORRECT

```
MPI_Reduce(a, recvbuf, num_rows*num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

Questions?

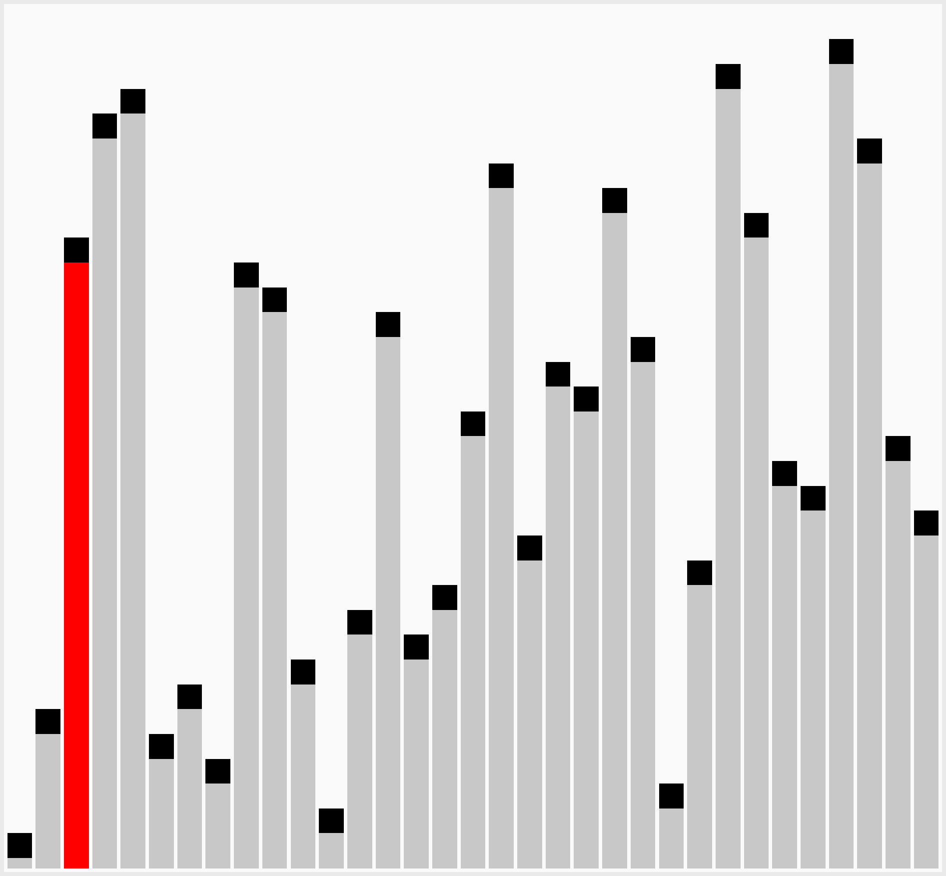
A Parallel Sorting Algorithm

Sorting

- n keys and p = comm_sz processes.
- n/p keys assigned to each process.
- No restrictions on which keys are assigned to which processes.
- When the algorithm terminates:
  - The keys assigned to each process should be sorted in (say) increasing order.
  - If $0 \le q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.
  - E.g.:

Process			
0	**1**	**2**	**3**
1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Serial bubble sort



Serial bubble sort

```
void Bubble_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    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;
            }

}  /* Bubble_sort */
```

Inerently sequential, not many opportunities for parallelization

Odd-even transposition sort

- A sequence of phases.
- Even phases, compare swaps:

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots$$

- Odd phases, compare swaps:

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots .$$

Example

Start: 5, 9, 4, 3

Even phase: compare-swap (5,9) and (4,3)
getting the list 5, 9, 3, 4

Odd phase: compare-swap (9,3)
getting the list 5, 3, 9, 4

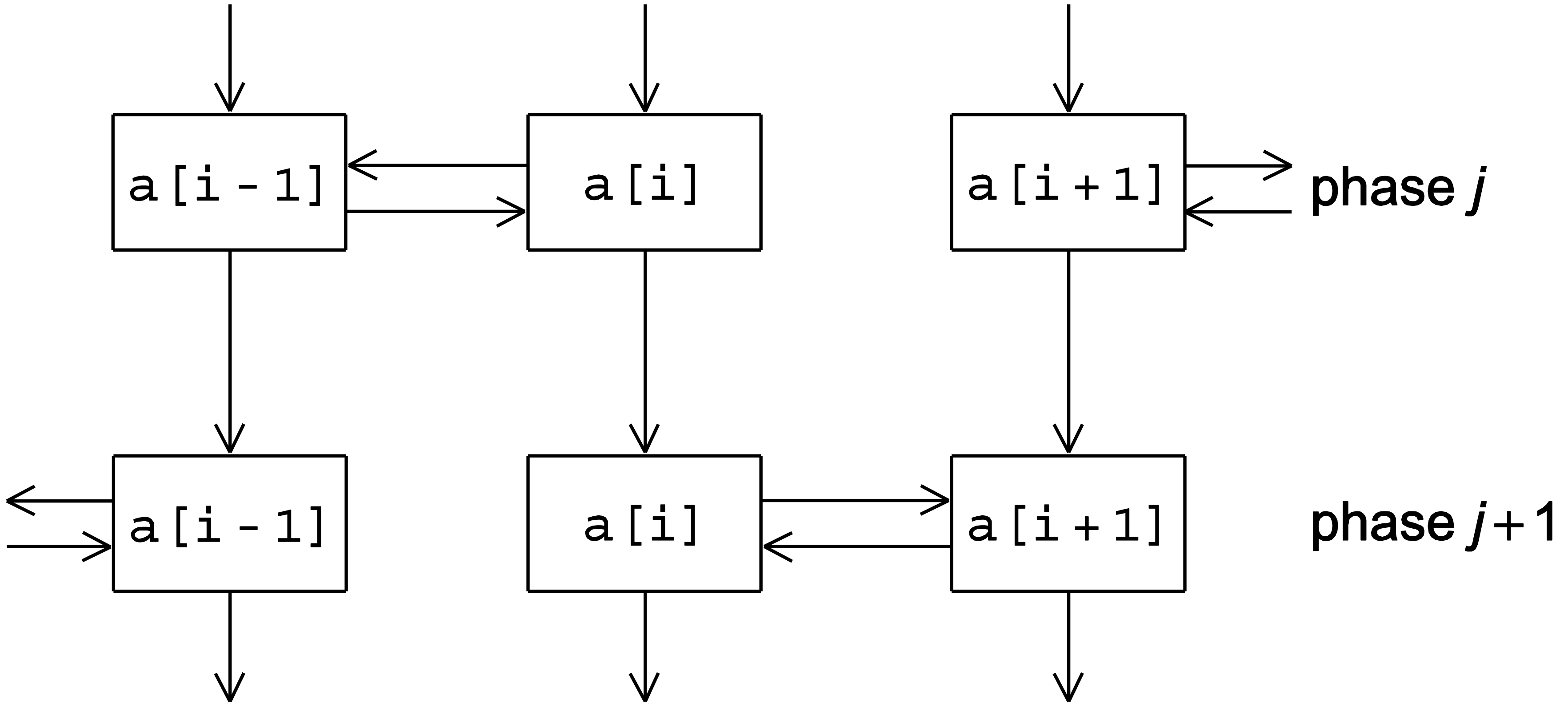
Even phase: compare-swap (5,3) and (9,4)
getting the list 3, 5, 4, 9

Odd phase: compare-swap (5,4)
getting the list 3, 4, 5, 9

Serial odd-even transposition sort

```
void Odd_even_sort(
      int  a[]  /* in/out  */,
      int  n    /* in      */) {
   int phase, i, temp;

   for (phase = 0; phase < n; phase++)
      if (phase % 2 == 0) { /* Even phase */
         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 { /* Odd phase */
         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_sort */
```

Communications among tasks in odd-even sort



Tasks determining a[i] are labeled with a[i].

Pseudo-code

- If number of elements to sort is equal to number of processes, each one has an element and communicates with the left/right neighbor depending on the phase being odd/even
- If n >> p:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

Parallel odd-even transposition sort

Time	Process			
	0	1	2	3
Start	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

Compute_partner

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)        /* Odd rank */
      partner = my_rank - 1;
   else                         /* Even rank */
      partner = my_rank + 1;
else                         /* Odd phase */
   if (my_rank % 2 != 0)        /* Odd rank */
      partner = my_rank + 1;
   else                         /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

Pseudo-code

- If number of elements to sort is equal to number of processes, each one has an element and communicates with the left/right neighbor depending on the phase being odd/even
- If n >> p:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

Parallel odd-even transposition sort

```
void Merge_low(
      int   my_keys[],     /* in/out     */
      int   recv_keys[],   /* in         */
      int   temp_keys[],   /* scratch    */
      int   local_n        /* = n/p, in */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

Questions?

Be careful with send/recv order

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
      0, comm, MPI_STATUS_IGNORE.
```

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
} else {
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```

Be careful with send/recv order

- An alternative to scheduling the communications ourselves (and to using Isend/Irecv/Wait).
- Carries out a blocking send and a receive in a single call.
- The dest and the source can be the same or different.
- Especially useful because MPI schedules the communications so that the program won't hang or crash.

MPI_Sendrecv

Another alternative is using MPI_Sendrecv

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

Example 1: Sum Between Vectors

Example: Sum Between Vectors

$$
\begin{aligned}
\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= \mathbf{z}
\end{aligned}
$$

Compute a vector sum.

Serial implementation of vector addition

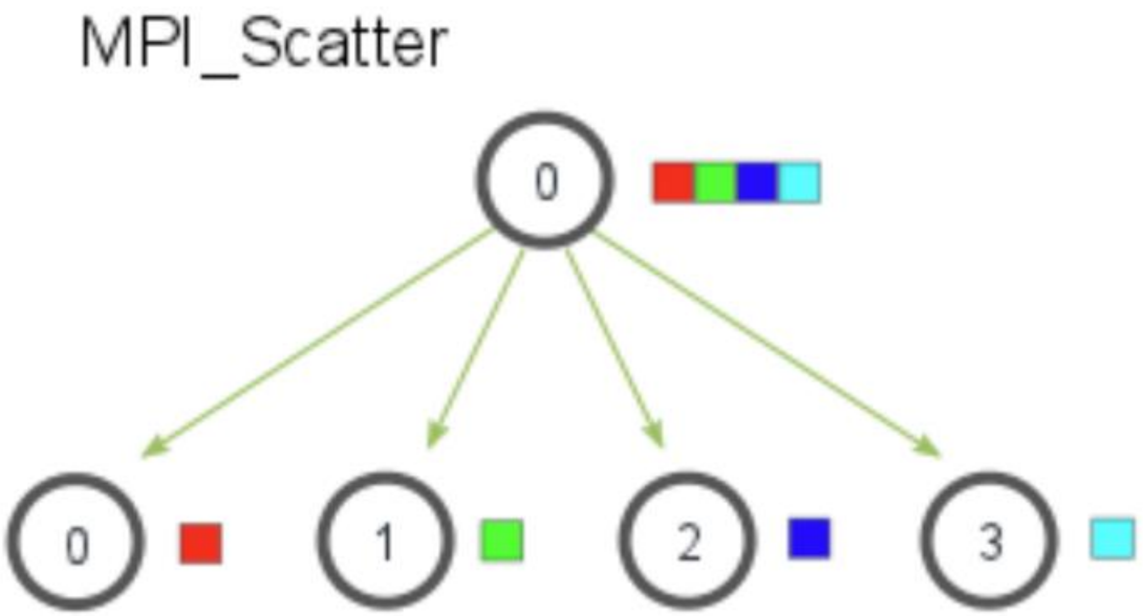
```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

Parallel implementation of vector addition

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.



ATTENTION:
Different from
MPI_Bcast

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

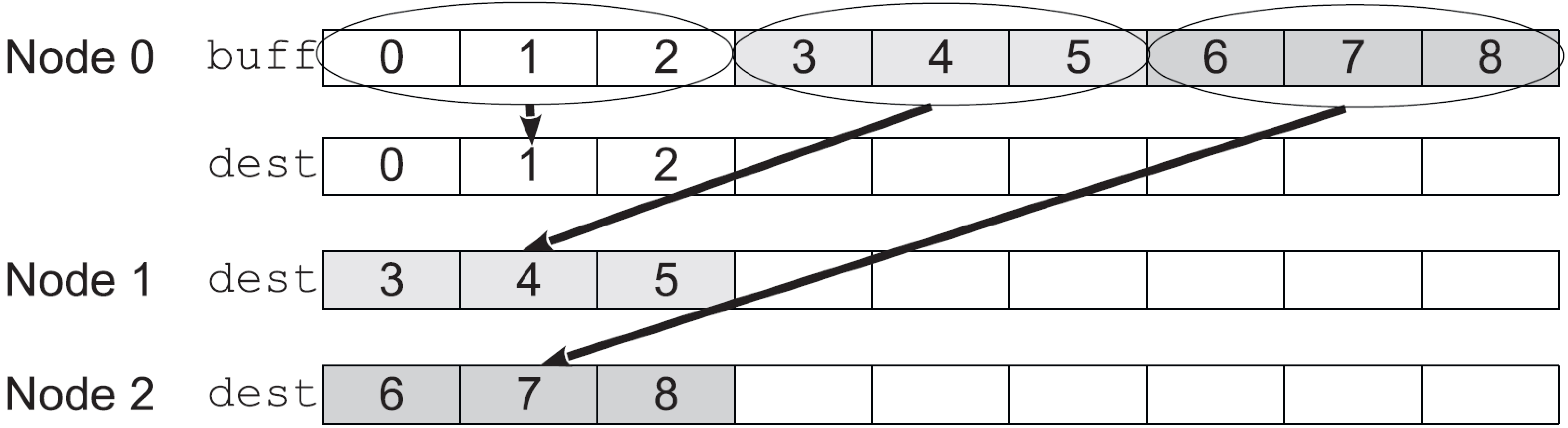
```
int MPI_Scatter(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
   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

ATTENTION: This is the number of elements to send to each process, not the total number of elements!!!

What if I want to send a different number of elements to each rank? MPI_Scatterv

Scatter

```
MPI_Scatter(buff, 3, MPI_INT,
            dest, 3, MPI_INT, 0, MPI_COMM_WORLD);
```



Scatter – In Place

```
if(rank == 0)
   MPI_Scatter(buff, 3, MPI_INT,
               MPI_IN_PLACE, 3, MPI_INT, 0, MPI_COMM_WORLD);
else
   MPI_Scatter(buff, 3, MPI_INT,
               dest, 3, MPI_INT, 0, MPI_COMM_WORLD);
```

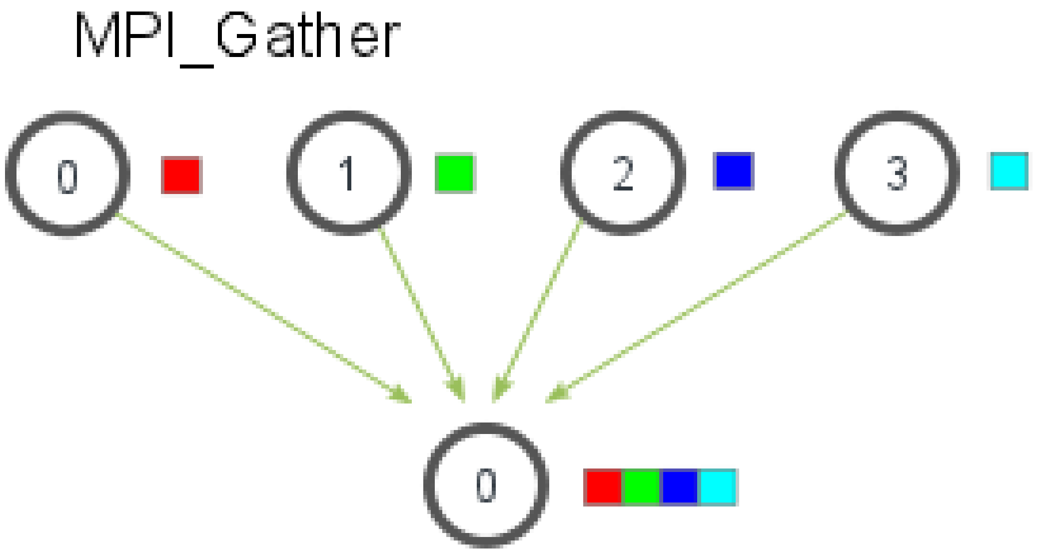
Node 0	buff	0	1	2	3	4	5	6	7	8
	dest									
Node 1	dest	3	4	5						
Node 2	dest	6	7	8						

Reading and distributing a vector

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.



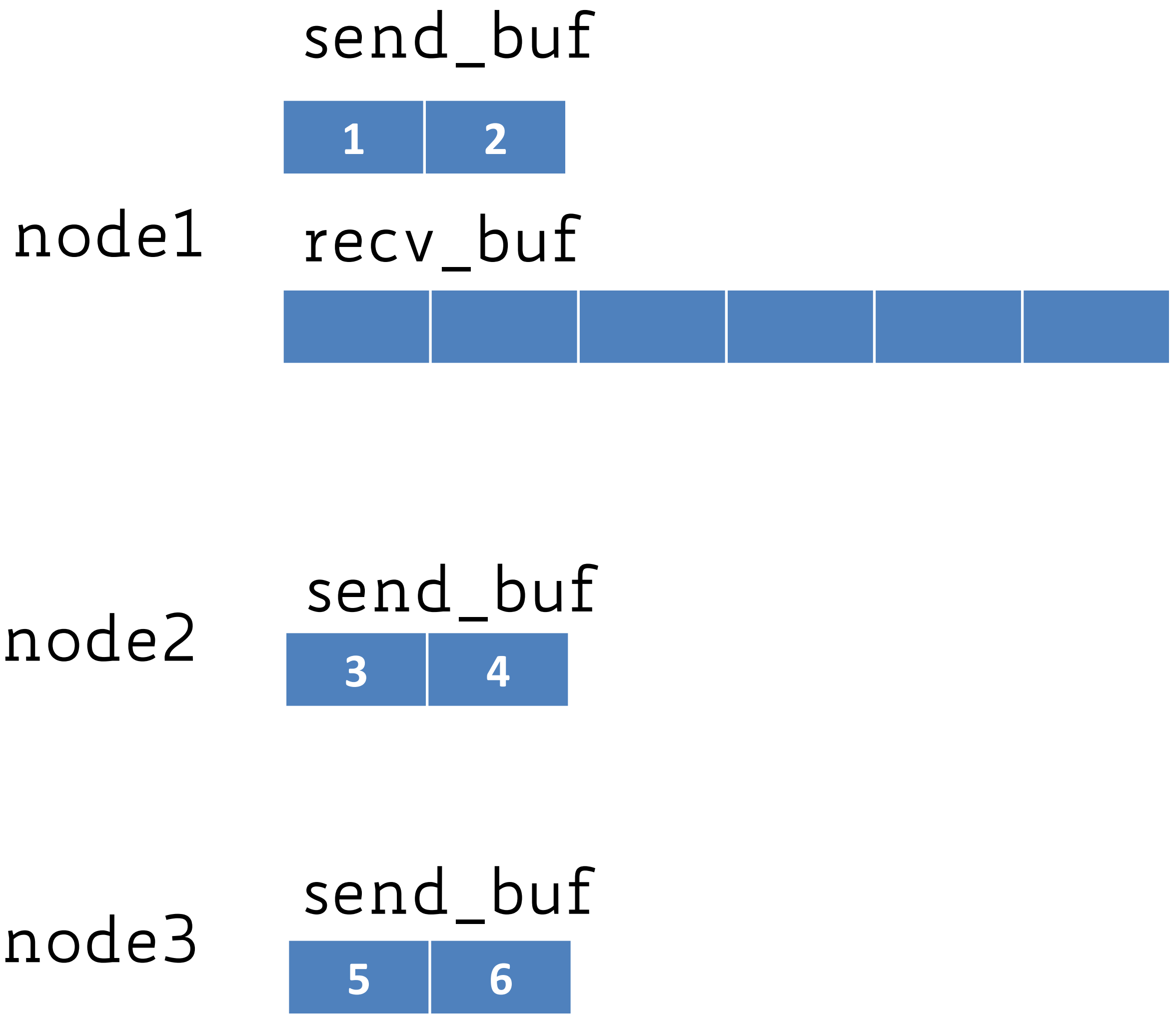
Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

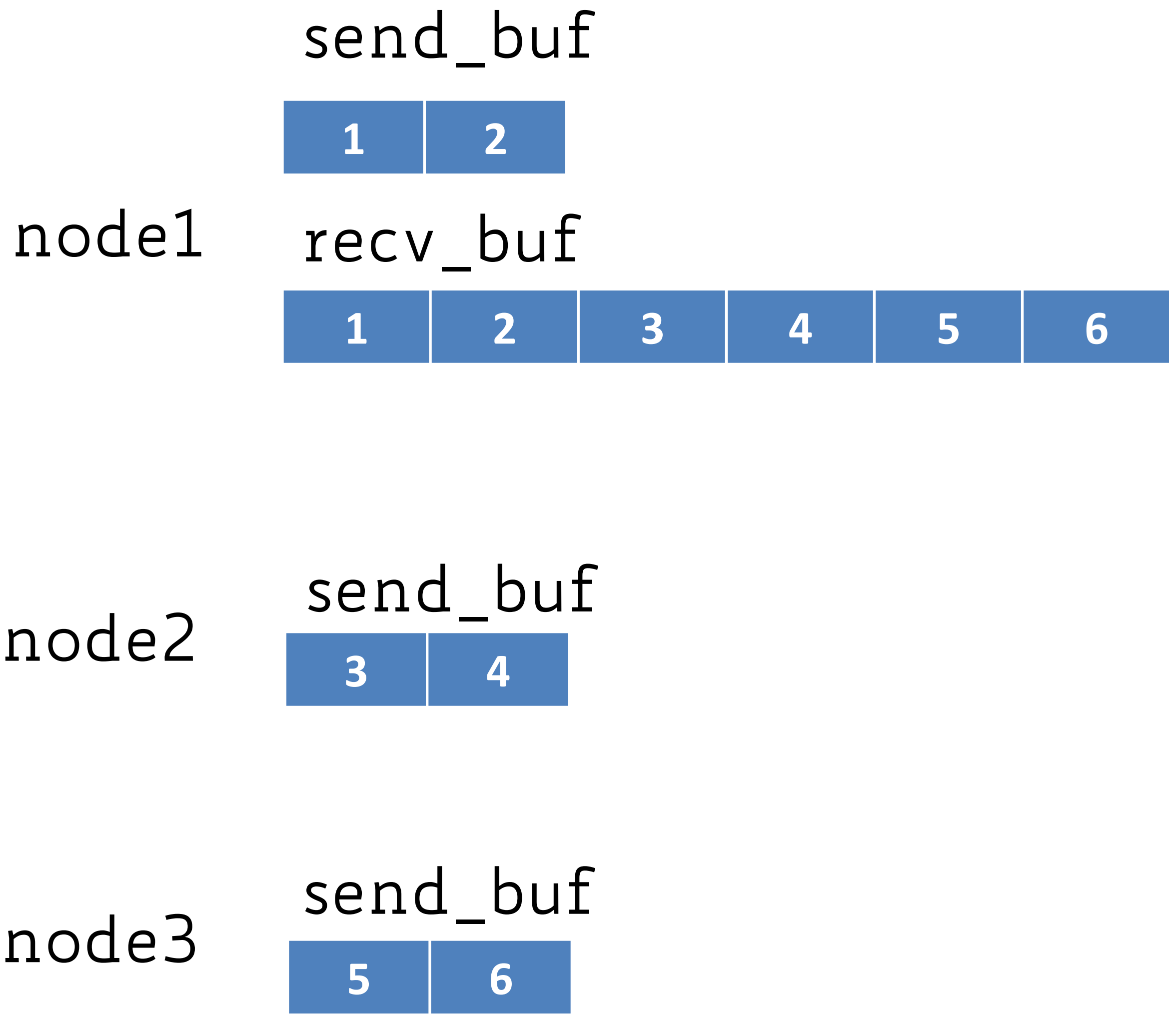
```
int MPI_Gather(
      void*           send_buf_p   /* in   */,
      int             send_count   /* in   */,
      MPI_Datatype    send_type    /* in   */,
      void*           recv_buf_p   /* out  */,
      int             recv_count   /* in   */,
      MPI_Datatype    recv_type    /* in   */,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

ATTENTION: This is the number of elements that each process sends, not the total number of elements in the final vector!!!

Gather



Gather


node1
send_buf
1 2
recv_buf
1 2 3 4 5 6
node2
send_buf
3 4
node3
send_buf
5 6

Print a distributed vector (1)

```
void Print_vector(
      double    local_b[]  /* in */,
      int       local_n    /* in */,
      int       n          /* in */,
      char      title[]    /* in */,
      int       my_rank    /* in */,
      MPI_Comm  comm       /* in */) {

   double* b = NULL;
   int i;
```

Print a distributed vector (2)

```
   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Print_vector */
```

Questions?

Example 2: Matrix-Vector Multiplication

Vector dot product

$$\begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = A_1B_1 + A_2B_2 + A_3B_3$$

Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

$\mathbf{x}$ is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i-th component of y

Dot product of the ith row of A with x.

Matrix-vector multiplication

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

Multiply a matrix by a vector

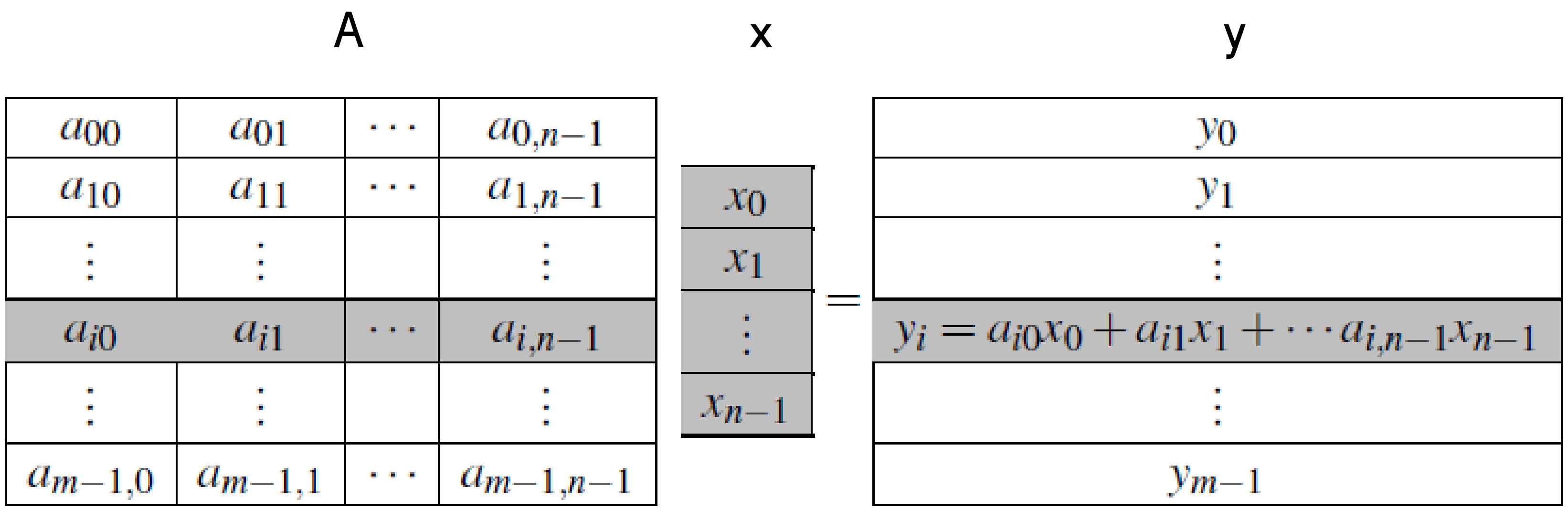
```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;

   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

Serial pseudo-code

Serial matrix-vector multiplication

```
void Mat_vect_mult(
        double  A[]  /* in   */,
        double  x[]  /* in   */,
        double  y[]  /* out  */,
        int     m    /* in   */,
        int     n    /* in   */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
}  /* Mat_vect_mult */
```

How to parallelize?



1. **Broadcast** the vector **x from rank 0 to the other processes**
2. **Scatter** the rows of the matrix A from rank 0 to the other processes
3. Each process **computes** a subset of the elements of the resulting vector **y**
4. **Gather** the final vector **y to rank 0**

Matrix Scattering

The A matrix is scattered by rows, so each process contains local_m rows

```
void Read_matrix(
      char      prompt[]  /* in  */,
      double    local_A[] /* out */,
      int       m         /* in  */,
      int       local_m   /* in  */,
      int       n         /* in  */,
      int       my_rank   /* in  */,
      MPI_Comm  comm      /* in  */) {
   double* A = NULL;
   int local_ok = 1;
   int i, j;

   if (my_rank == 0) {
      A = malloc(m*n*sizeof(double));
      printf("Enter the matrix %s\n", prompt);
      for (i = 0; i < m; i++)
         for (j = 0; j < n; j++)
            scanf("%lf", &A[i*n+j]);
      MPI_Scatter(A, local_m*n, MPI_DOUBLE,
            local_A, local_m*n, MPI_DOUBLE, 0, comm);
      free(A);
   } else {
      MPI_Scatter(A, local_m*n, MPI_DOUBLE,
            local_A, local_m*n, MPI_DOUBLE, 0, comm);
   }
}  /* Read_matrix */
```

How to parallelize?

Now, let's assume that this is done in a loop, and the output y is used as the input vectpr for the next iteration

e.g.:
Iteration 0: $y_0 = A \cdot x$
Iteration 1: $y_1 = A \cdot y_0$
Iteration 2: $y_2 = A \cdot y_1$
…

First iteration:

1. **Broadcast** the vector **x from rank 0**
2. **Scatter** the rows of the matrix **A** from rank 0 to the other processes
3. Each process **computes** a subset of the elements of the resulting vector y_0
4. **Gather** the final vector y_0 to rank 0
5. **Broadcast** y_0 from rank 0 to the other processes

Is there any collective that does that?

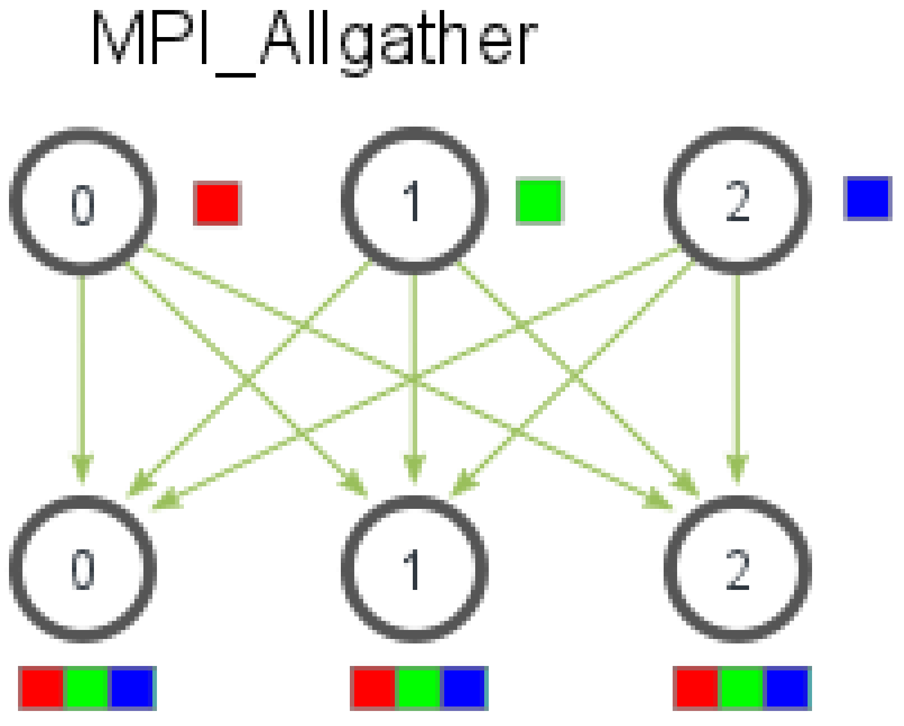
All the other iterations:

1. Each process **computes** a subset of the elements of the resulting vector y_i using y_{i-1} received in previous step
2. **Gather** the final vector y_i to rank 0
3. **Broadcast** y_i from rank 0 to the other processes

Is there any collective that does that?

Allgather

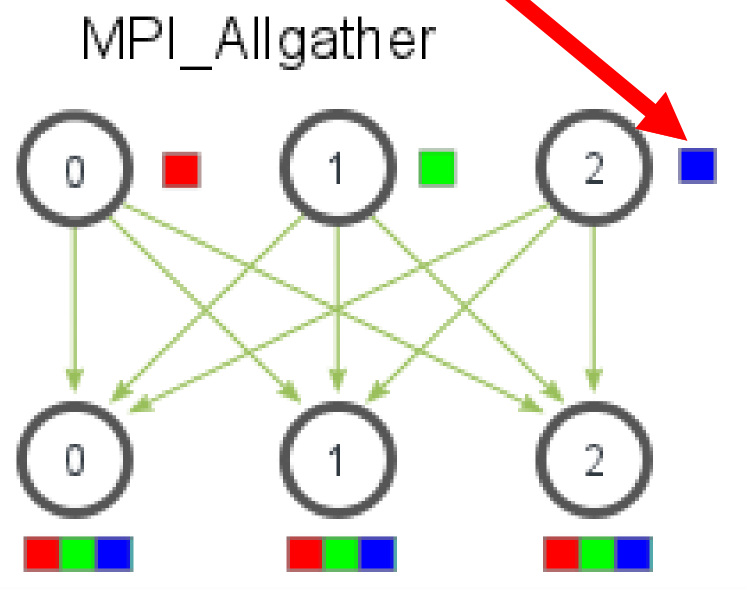
- Conceptually, it is like a Gather + Broadcast
- In practice, it might be implemented in a more efficient way



Allgather

```
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  */);
```

ATTENTION: Number of elements sent by each process



Questions?

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(
      double    local_A[]  /* in   */,
      double    local_x[]  /* in   */,
      double    local_y[]  /* out  */,
      int       local_m    /* in   */,
      int       n          /* in   */,
      int       local_n    /* in   */,
      MPI_Comm  comm       /* in   */) {
   double* x;
   int local_i, j;
   int local_ok = 1;
```

An MPI matrix-vector multiplication function (2)

```
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (2)

Collect x from
Different processes

```
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (2)

```
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

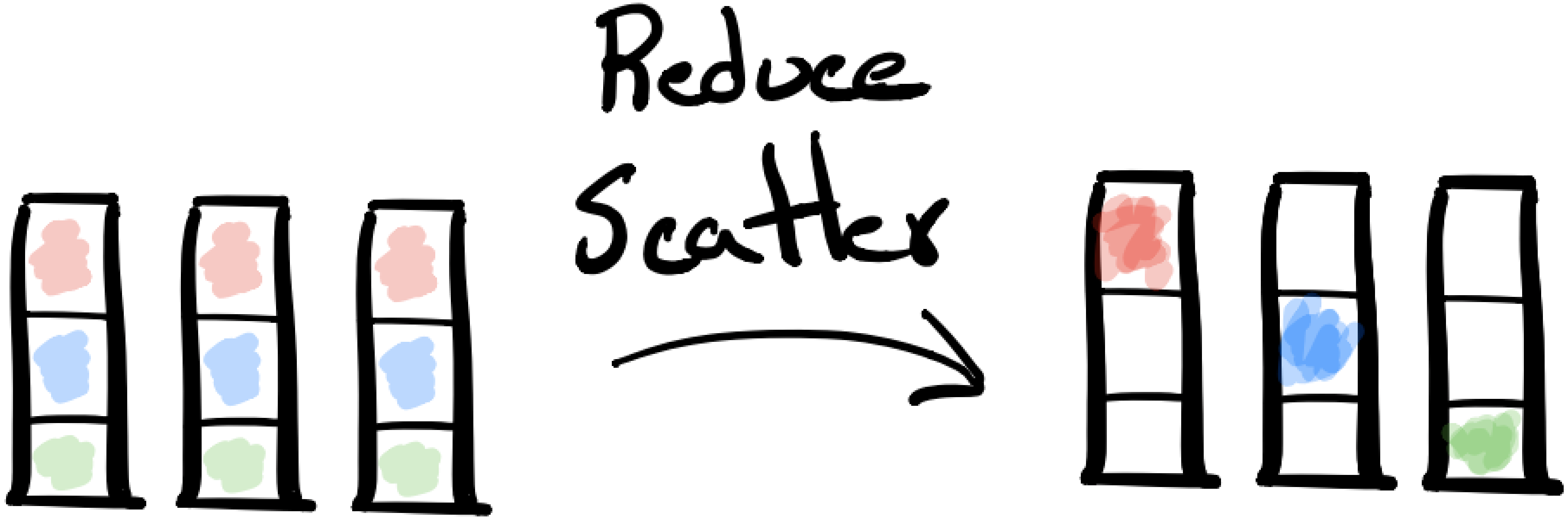
Only *local_m* rows are used

Questions?

Another couple of useful collectives

Reduce-Scatter

- Each rank gets the sum of just a part of the vector



MPI_Alltoall

node1

send_buf

1	2	3	4	5	6

recv_buf

node2

send_buf

11	12	13	14	15	16

recv_buf

node3

send_buf

21	22	23	24	25	26

recv_buf

MPI_Alltoall

node1

send_buf

1	2	3	4	5	6

recv_buf

1	2	11	12	21	22

node2

send_buf

11	12	13	14	15	16

recv_buf

3	4	13	14	23	24

node3

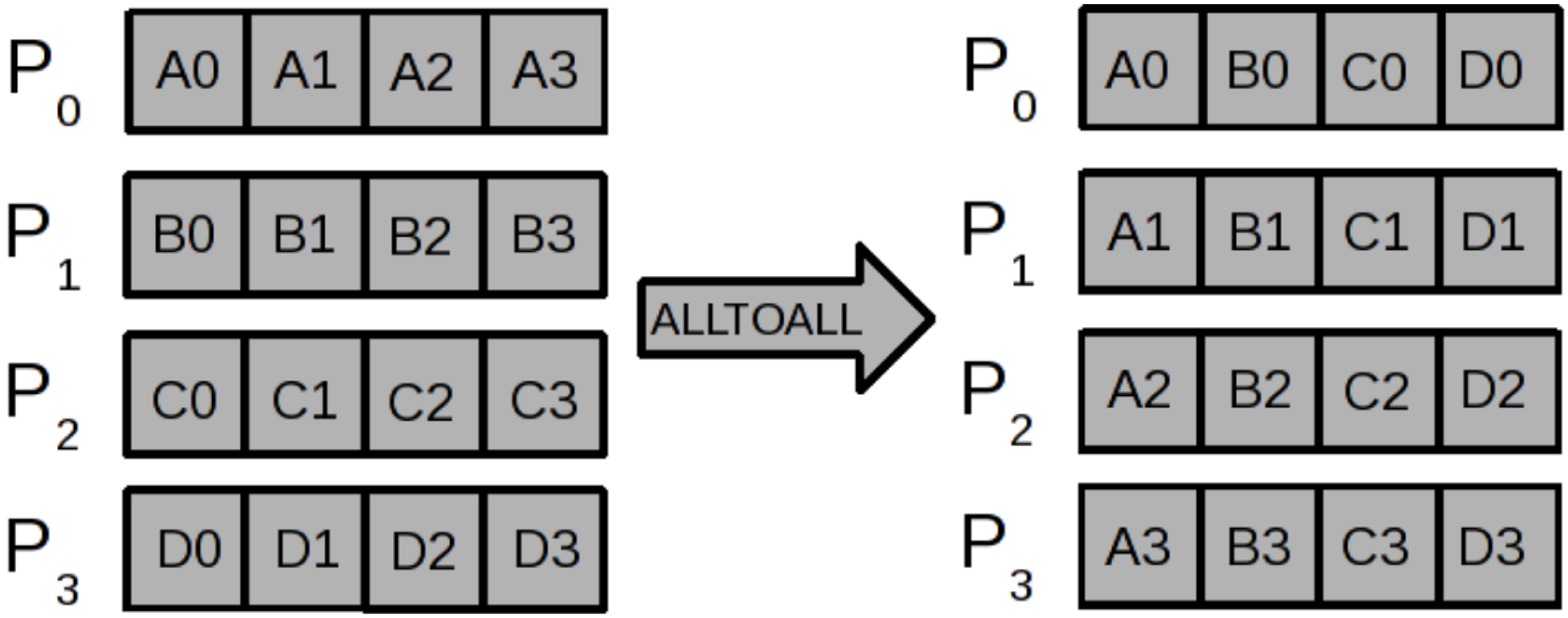
send_buf

21	22	23	24	25	26

recv_buf

5	6	15	16	25	26

MPI_Alltoall



MPI Derived Datatypes

Trapezoidal rule: Function for reading user input

```
void Get_input(
      int      my_rank   /* in  */,
      int      comm_sz   /* in  */,
      double*  a_p       /* out */,
      double*  b_p       /* out */,
      int*     n_p       /* out */) {
   int dest;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
   } else { /* my_rank != 0 */
      MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
   }
}  /* Get_input */
```

Not efficient (3 send, we can do with 1, but how?)

Not efficient (3 recv, we can do with 1, but how?)

Derived datatypes

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- The idea is that if a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.
- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received.

Communicating Objects

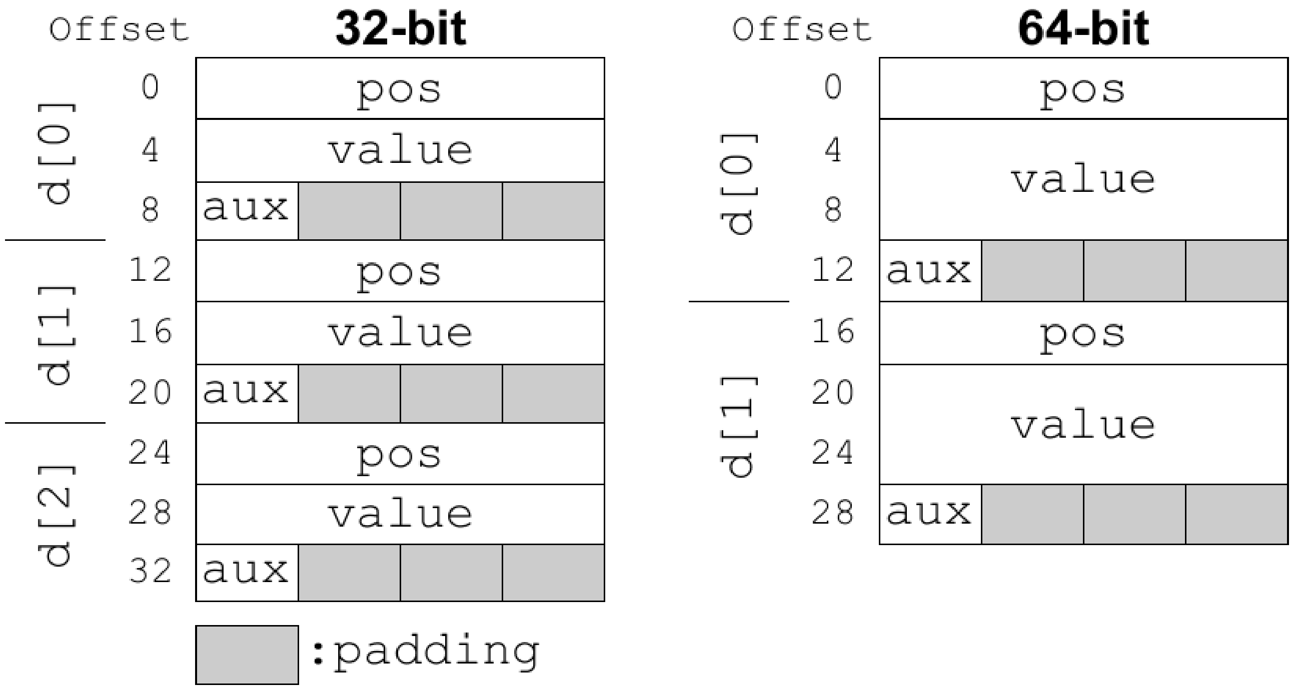
- Will the following work?

```
const int N = ...;

typedef struct Point
  {
    int x;
    int y;
    int color;
  };

Point image[N];
. . .
MPI_Send(image, N*sizeof(Point), MPI_BYTE, dest, tag, MPI_COMM_WORLD);
```

A counter-example

A illustrating example of a stucture's layout in two different platforms:

```
struct T{
    int pos;
    long value;
    char aux;
} d[N];
```



Derived datatypes

- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.
- Trapezoidal Rule example:

Variable	Address
a	24d
b	40d
n	48d

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}$$

MPI_Type_create_struct

Builds a derived datatype that consists of individual elements that have different basic types.

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

e.g., for

```
struct t{
        double a;
        double b;
        int n;
};
```

count = 3

blocklengths = 1,1,1

displacements = 0,16,24

types = MPI_DOUBLE, MPI_DOUBLE, MPI_INT

MPI_Type create_struct

Builds a derived datatype that consists of individual elements that have different basic types.

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

e.g., for

```
struct t{
        double a[10];
        double b;
        int n;
};
```

count = 3

blocklengths = 10,1,1

displacements = ?,?,?

types = MPI_DOUBLE, MPI_DOUBLE, MPI_INT

How to get the displacements? MPI_Get_address

- Returns the address of the memory location referenced by location_p.
- The special type MPI_Aint is an integer type that is big enough to store an address on the system.

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

e.g., for

```
struct t{
        double a;
        double b;
        int n;
};
```

```
MPI_Aint a_addr, b_addr, n_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(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

How to get the displacements? MPI_Get_address

Why instead of doing this:

```
MPI_Aint a_addr, b_addr, n_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(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

```
struct t{
        double a;
        double b;
        int n;
};
```

We don't do:

```
array_of_displacements[0] = 0;
array_of_displacements[1] = &b - &a;
array_of_displacements[2] = &n - &a;
```

?

Note, however, that & *cast-expression* is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at --- although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to ``reference'' C variables guarantees portability to such machines as well. (*End of advice to users.*)

https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node74.htm

TL;DR: Dereferencing with & should work in 99% of the cases, but better to be safe and use MPI_Get_address

MPI_Type_commit

- Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

```
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```

MPI_Type_free

- When we’re finished with our new type, this frees any additional storage used.

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

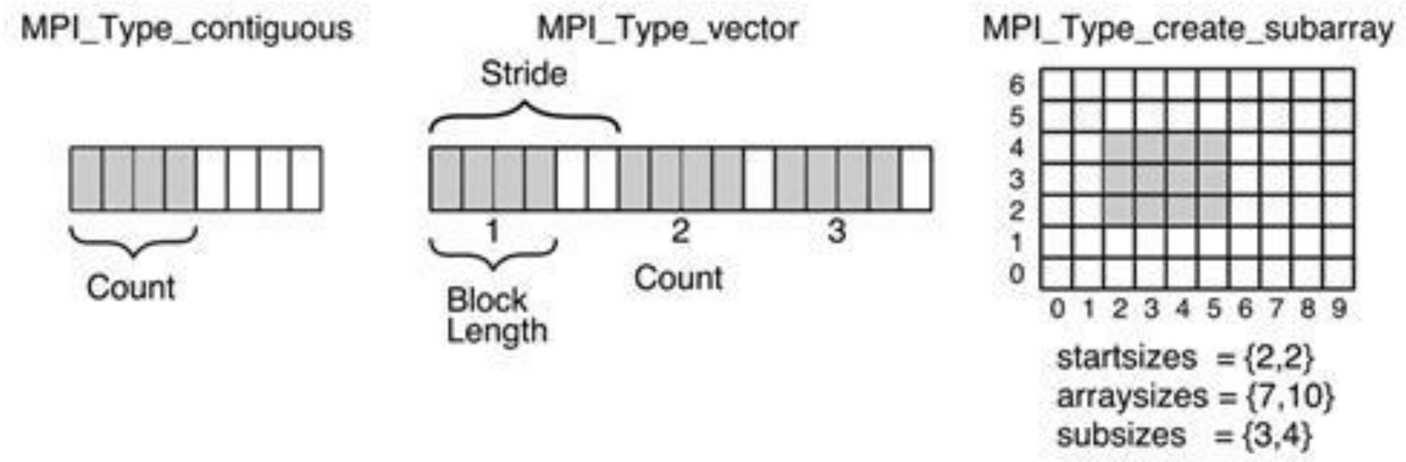
Putting Everything Together

```
void Build_mpi_type(
      double*        a_p              /* in   */,
      double*        b_p              /* in   */,
      int*           n_p              /* in   */,
      MPI_Datatype*  input_mpi_t_p    /* out  */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};
   MPI_Get_address(a_p, &a_addr);
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */
```

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

It is called MPI_Type_create_struct, but the elements must not be necessarily come from a struct (you can use any sequence of variables)

Other functions for new datatypes



- MPI_Type_contiguous: Contiguous elements
- MPI_Type_vector: consists of a number of elements of the same datatype repeated with a certain stride (see ex. 3.17 and 3.18)
- MPI_Type_create_subarray: for multidimensional subarrays
- MPI_pack /MPI_Unpack: to pack data and send/receive packed data (see ex. 3.20)

MPI defines around 400 functions, around 40 of which for managing datatypes