Matrix times Vector in MPI

MatVec Problem: Compute product of matrix (*m x n*) and vector (*n x 1*)

- Compute

$$y = \begin{pmatrix} y_0 \\ y_1 \\ \cdot \\ \cdot \\ y_{m-1} \end{pmatrix} = Ax$$

- Given

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdot & \cdot & a_{0,n-1} \\ a_{1,0} & a_{1,1} & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ a_{m-1,0} & & & & a_{m-1,n-1} \end{pmatrix}$$

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \\ \\ x_{n-1} \end{pmatrix}$$

MatVec: Serial Code

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

MatVec: Parallel?

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

- **But if our matrix is *BIG* it won't fit in memory on one node of blueshark.**
- **We're going to have to use distributed memory.**
- **We'll need to code this in MPI**
- **We'll need some more MPI functions.**

MatVec: How to parallelize?

- Pretty clear we want to break the nested loop into pieces to be done on each processor.

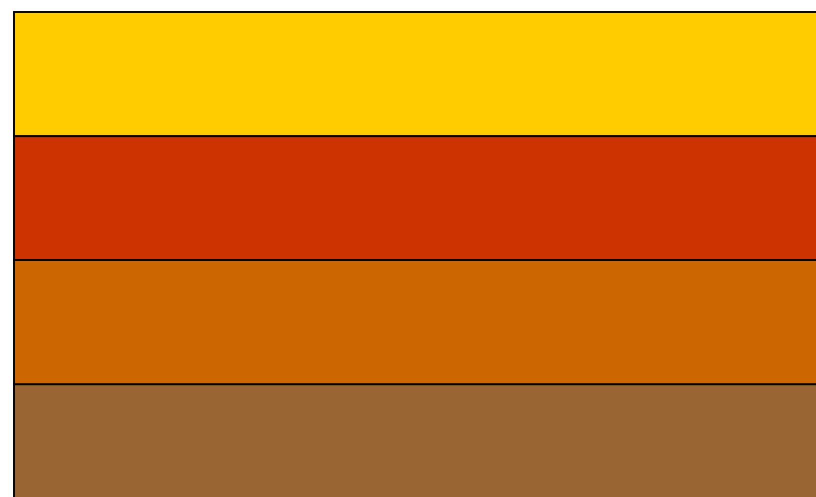
- Related questions:
  - A processor is going to do which part of the computation? (parallelize computation)
  - A processor is going to “own” which part of the data? (data distribution)

Parallel thinking

- You thought about
  - How to distribute data
  - What computation each processor does
  - What communication is needed
  - To make life easy, assume that the matrix sizes n and m are divisible by the number of processes, or even that $n=m=kp$

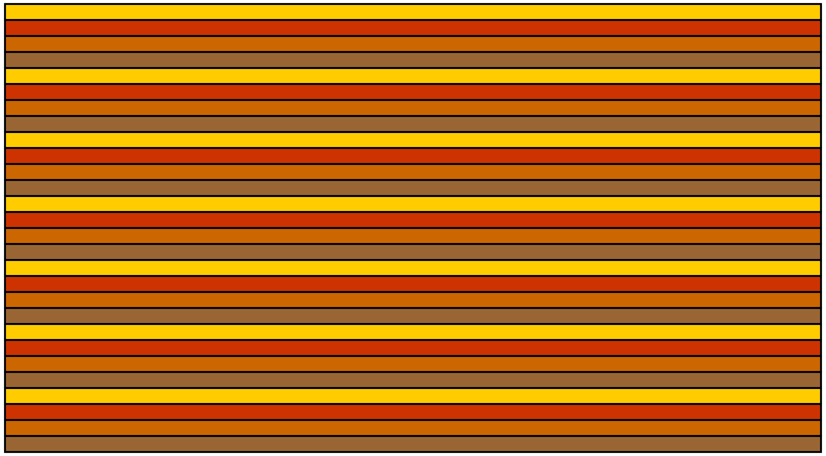
MatVec Data Distribution 1: Block-row distribution

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdot & \cdot & a_{0,n-1} \\ a_{1,0} & a_{1,1} & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ a_{m-1,0} & & & & a_{m-1,n-1} \end{pmatrix}$$



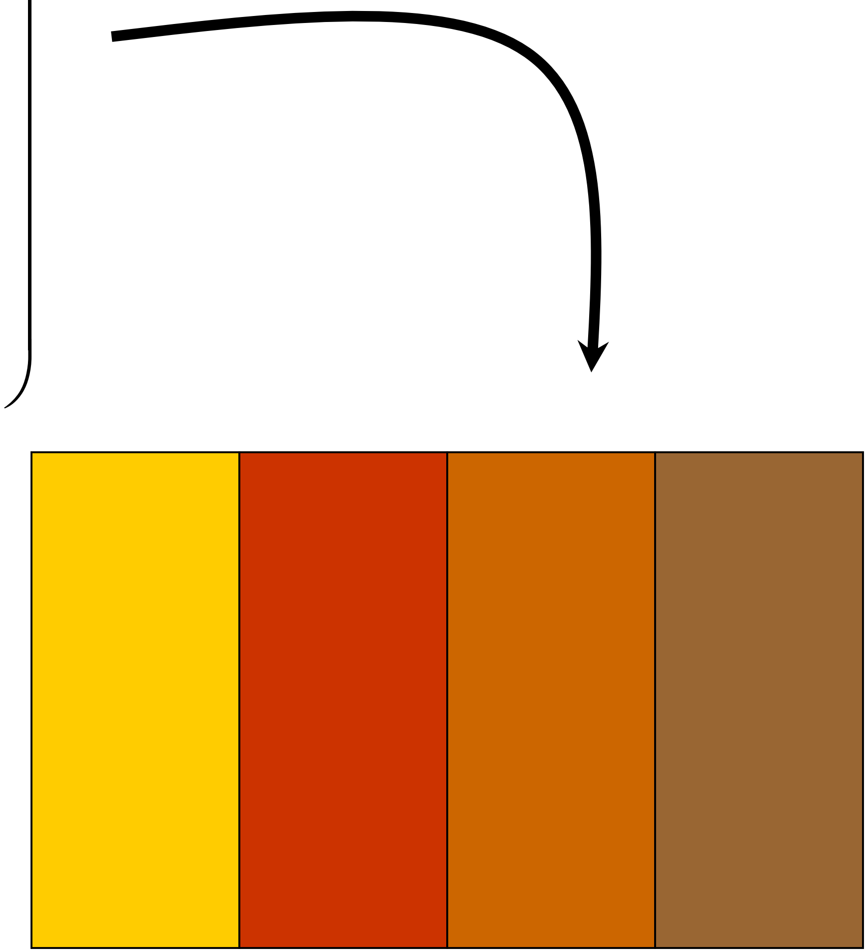
MatVec Data Distribution 2: Block-row Cyclic Distribution

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdot & \cdot & a_{0,n-1} \\ a_{1,0} & a_{1,1} & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ a_{m-1,0} & & & & a_{m-1,n-1} \end{pmatrix}$$



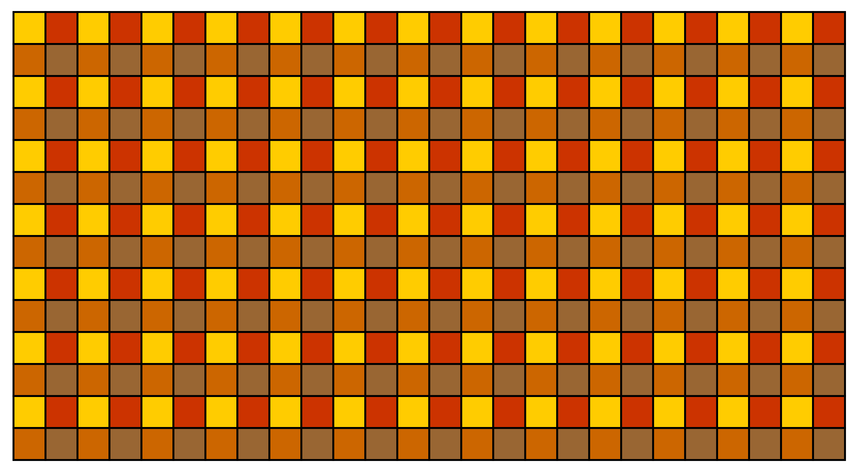
MatVec Data Distribution 3: Block-column Distribution

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdot & \cdot & a_{0,n-1} \\ a_{1,0} & a_{1,1} & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ a_{m-1,0} & & & & a_{m-1,n-1} \end{pmatrix}$$



MatVec Data Distribution 4: Block row/column Cyclic Distribution

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdot & \cdot & a_{0,n-1} \\ a_{1,0} & a_{1,1} & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ a_{m-1,0} & & & & a_{m-1,n-1} \end{pmatrix}$$



MatVec: Let's go with a block-row distribution for matrix and vectors

A x = y

$$
\begin{bmatrix}
a_{00} & a_{01} & a_{02} & a_{03} \\
a_{10} & a_{11} & a_{12} & a_{13} \\
\hline
a_{20} & a_{21} & a_{22} & a_{23} \\
a_{30} & a_{31} & a_{32} & a_{33} \\
\hline
a_{40} & a_{41} & a_{42} & a_{43} \\
a_{50} & a_{51} & a_{52} & a_{53} \\
\hline
a_{60} & a_{61} & a_{62} & a_{63} \\
a_{70} & a_{71} & a_{72} & a_{73}
\end{bmatrix}
\begin{bmatrix}
x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline x_3
\end{bmatrix}
=
\begin{bmatrix}
y_0 \\ y_1 \\ \hline y_2 \\ y_3 \\ \hline y_4 \\ y_5 \\ \hline y_6 \\ y_7
\end{bmatrix}
$$

MatVec: Parallelize computation of first y-entry

$$
\begin{array}{ccccc}
A_{0*} & \cdot & x & = & y_0 \\
\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{bmatrix} & \cdot & \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} & = & \begin{bmatrix} y_0 \end{bmatrix}
\end{array}
$$

- Option 1: send x’s to P0, it does computation.
- Option 2: send a’s to P1,P2 & P3, they do computation.

Collective communication from MPI Intro

- Broadcast
  - send data from one process to all
- Reduce
  - Apply an operator (sum, product , …) to data on all processes
- Like all MPI collective communication calls, these functions should be called by all processes.

MatVec: Parallelize computation of first y-entry

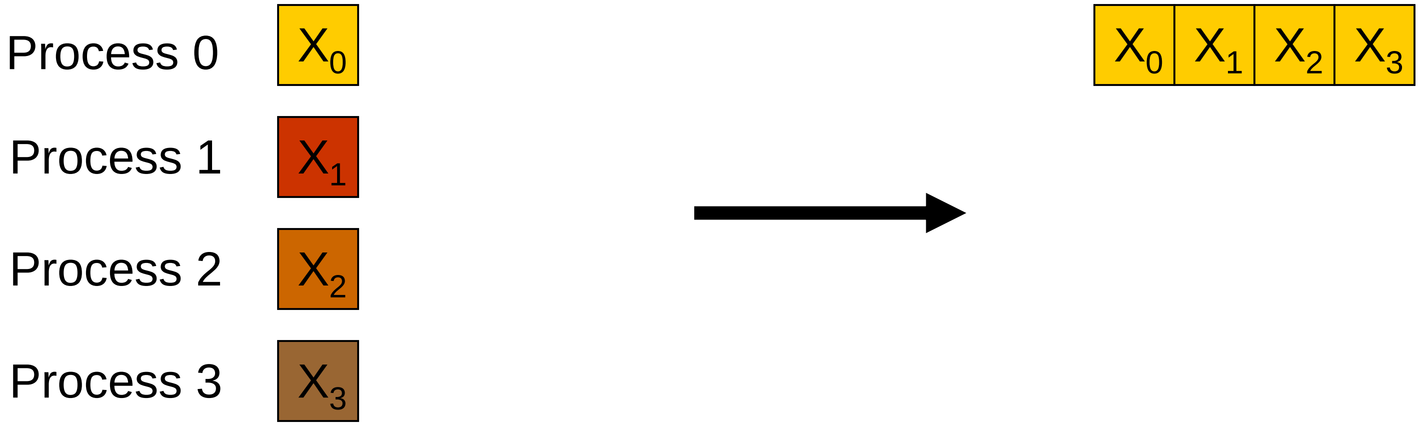
$$A_{0*} \quad \cdot \quad x \quad = \quad y_0$$

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = y_0$$

- Option 1: send x’s to P0, it does computation.
- Option 2: send a’s to P1,P2 & P3, they do computation.

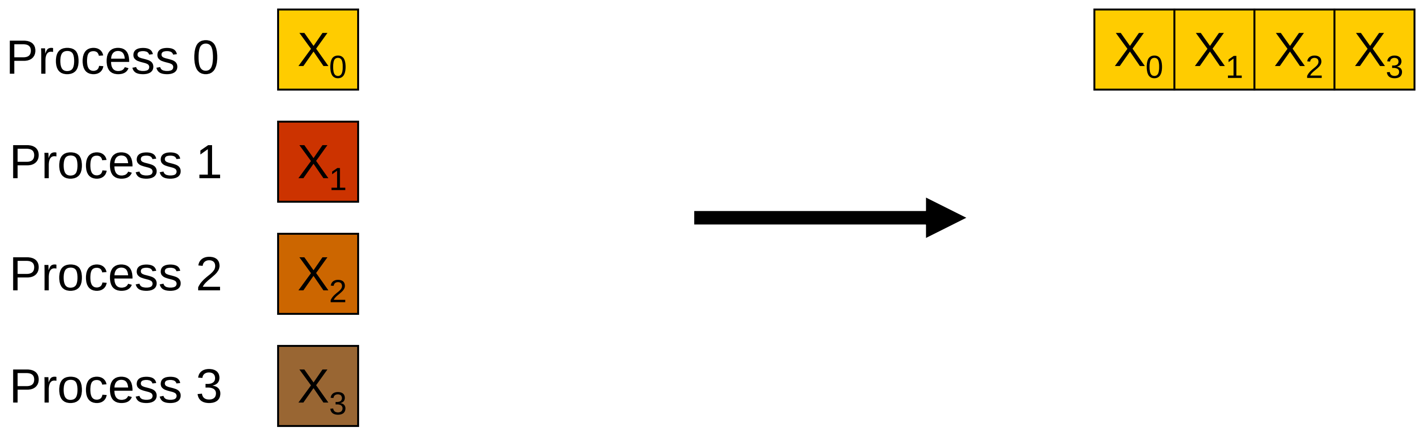
Option 1 requires gathering data from all processors to one processor

- Gather:



Option 1 requires gathering data from all processors to one processor

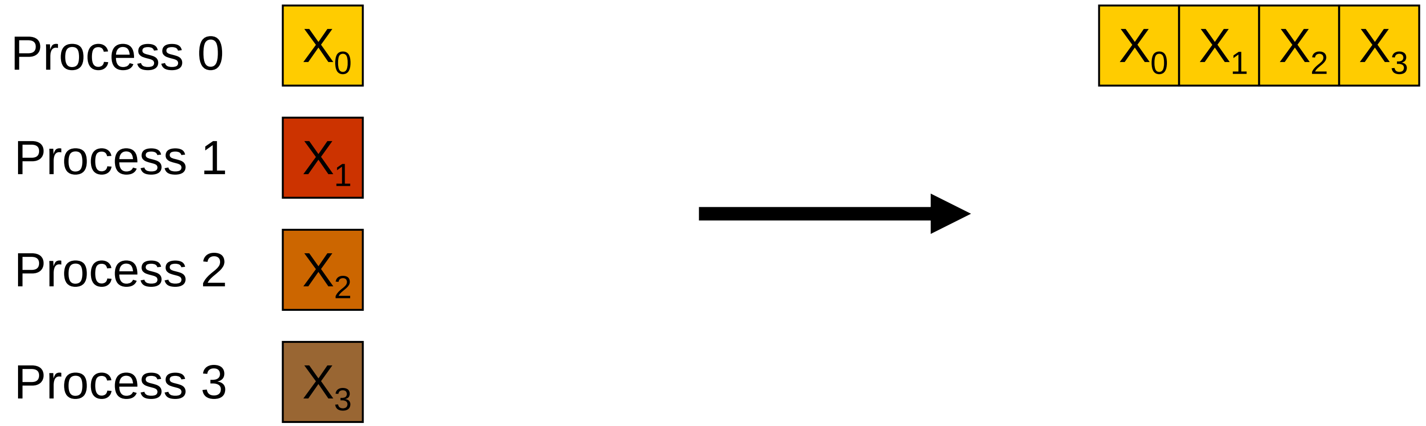
- Gather:



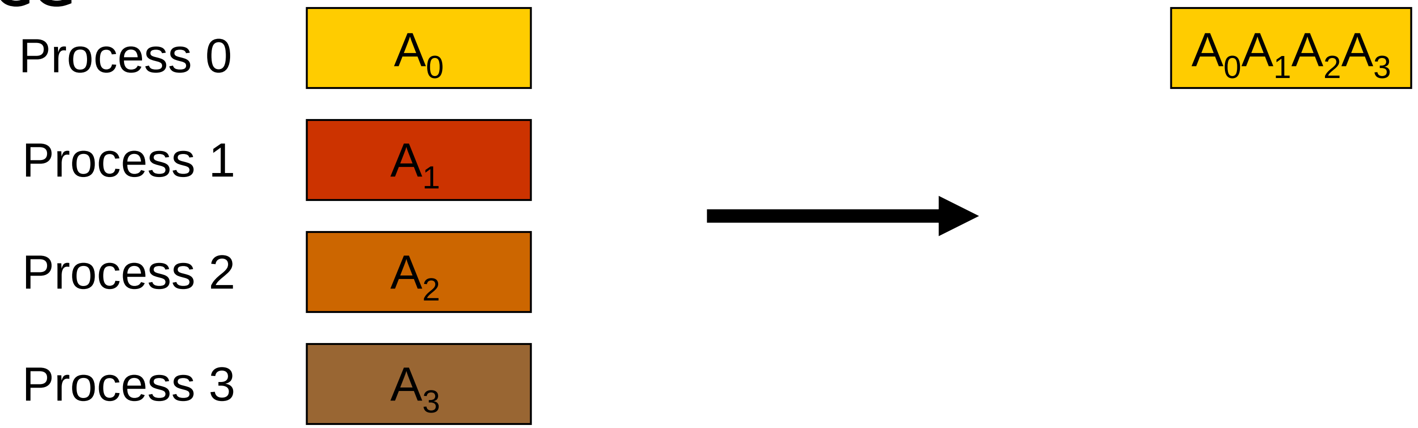
Different concept than reduce where we wanted a composite quantity on the root process like X0+x1+x3+x4 or x1*x2*x3*x4

Gather vs. Reduce

- Gather:



- Reduce



MPI_Gather syntax meaning: all processors

```
MPI_Gather(
        void*           send_data,
        int             send_count,
        MPI_Datatype    send_type,
        void*           recv_data,
        int             recv_count,
        MPI_Datatype    recv_type,
        int             root,
        MPI_Comm        comm)
```

Address of local data to be sent to root processor

Number of local data elements *each* process will send

Type of local data

Who to send to

Everybody who's involved

MPI_Gather syntax meaning: root processor

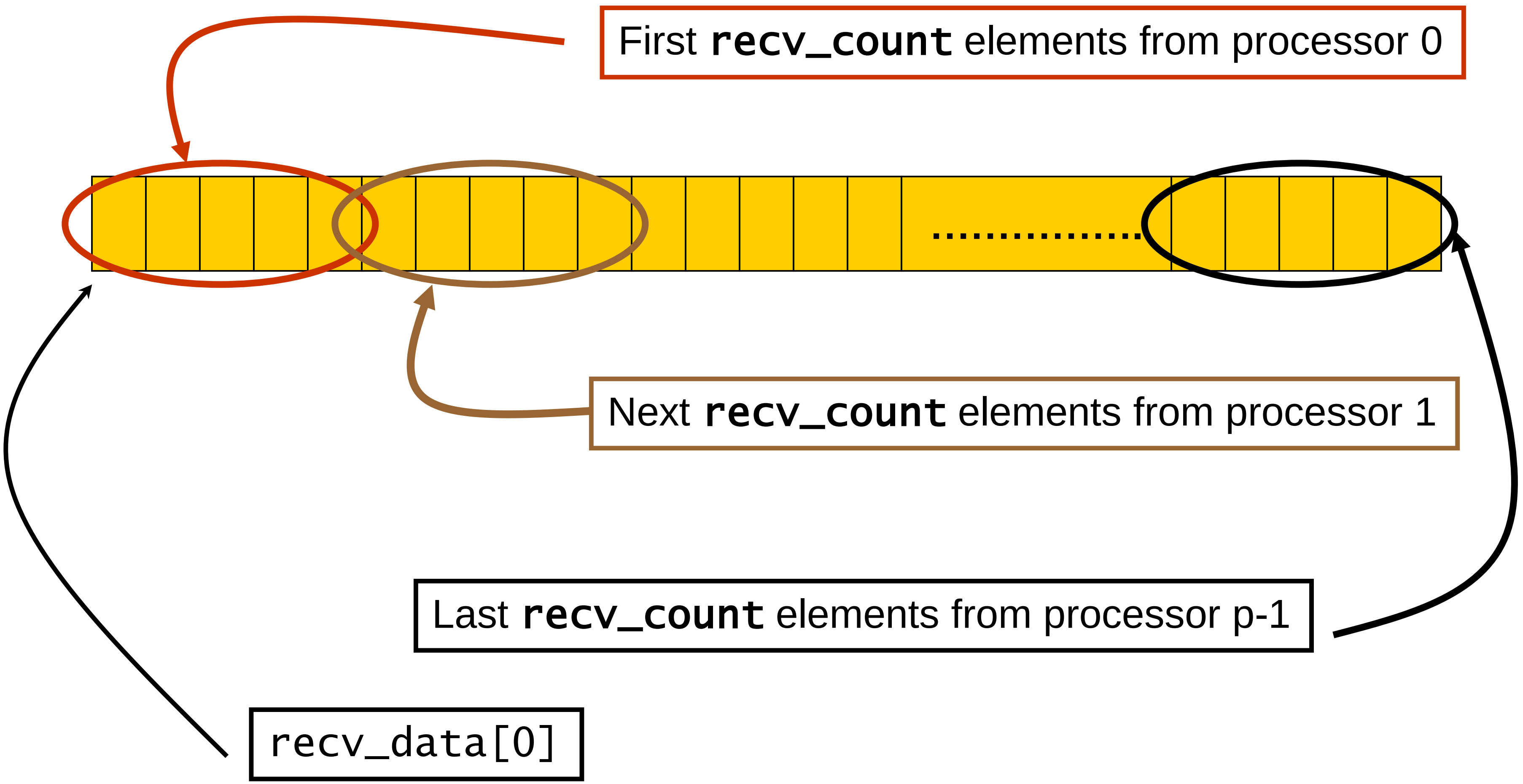
```
MPI_Gather(
        void*          send_data,
        int            send_count,
        MPI_Datatype   send_type,
        void*          recv_data,
        int            recv_count,
        MPI_Datatype   recv_type,
        int            root,
        MPI_Comm       comm)
```

Address where gathered, global data is to be stored. Required storage = P*recv_count*SizeOf(recv_type)

Number of data elements to be received from *each* processor, most often = send_count

Type of local data, most often = send_type

MPI_Gather syntax meaning: root processor



MatVec: Parallelize computation of first y-entry

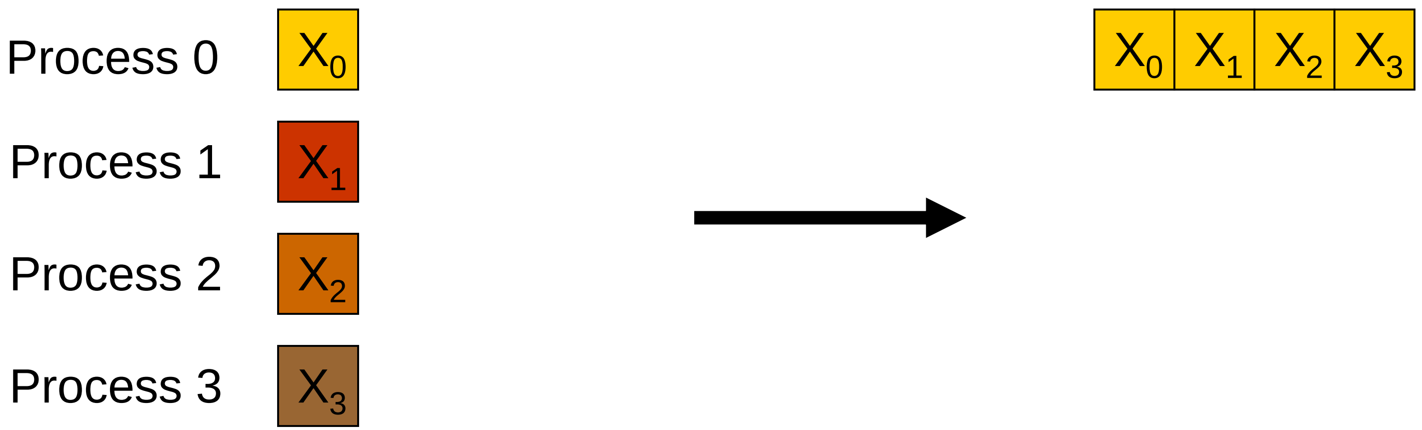
$$A_{0*} \quad \cdot \quad x \quad = \quad y_0$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_0 \end{bmatrix}$$

- Option 1: send x’s to P0, it does computation.
- Option 2: send a’s to P1,P2 & P3, they do computation.

Option 1 requires gathering data from all processors to one processor

- Gather:




```
root = 0;
send_count = 1, recv_count= 1;
MPI_Gather( &X_local, send_count, MPI_FLOAT,
            X_global, recv_count, MPI_FLOAT,
             root, MPI_COMM_WORLD );
```

Option 1 requires gathering data from all processors to one processor

C -- arguments one and four are addresses

The first argument for example should be

- **&X_local if X_local is a single scalar float or double**
- **X_local or &X_local[0] if X_local is an array of floats or doubles**

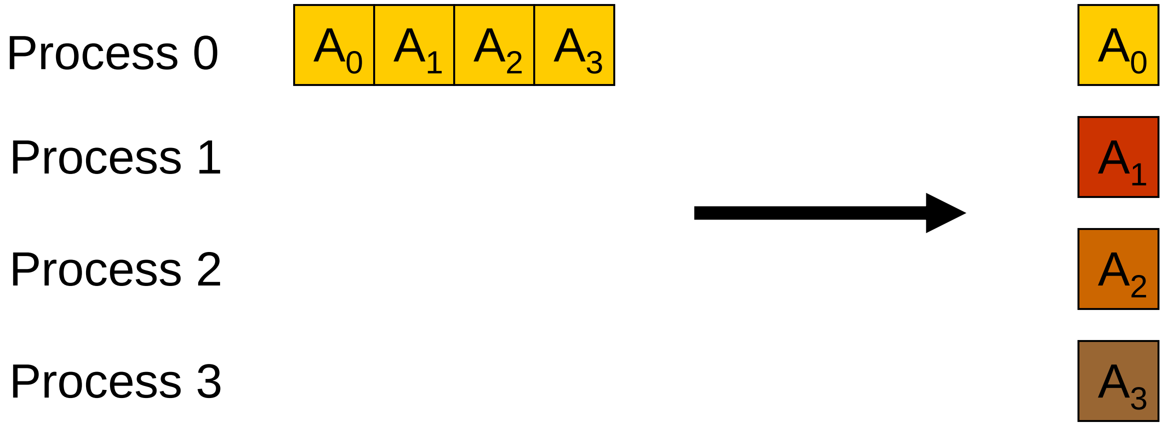
The fourth argument is for an array on the root processor so it should be

- **X_global or &X_global[0] where X_global is an array of floats or doubles**

```
root = 0;
send_count = 1, recv_count= 1;
MPI_Gather( &X_local, send_count, MPI_FLOAT,
            X_global, recv_count, MPI_FLOAT,
             root, MPI_COMM_WORLD );
```

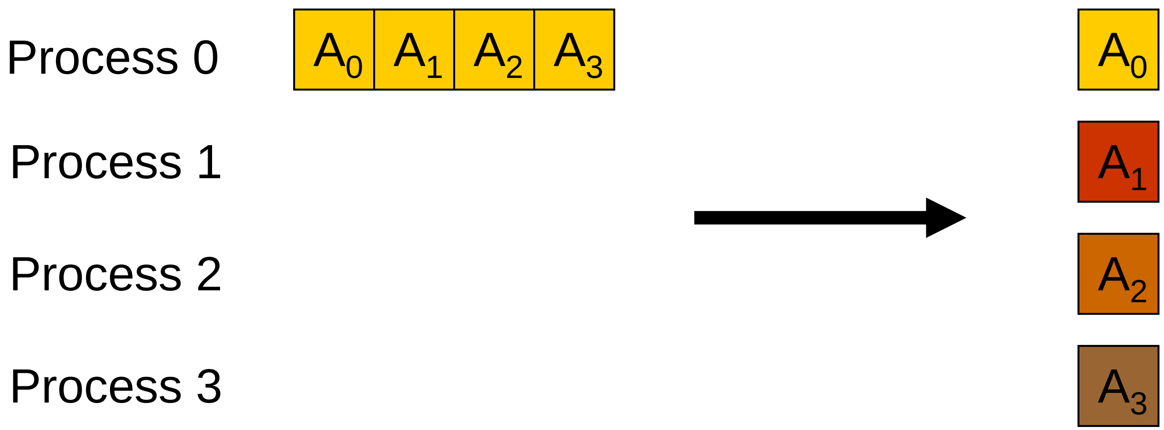
Option 2 requires scattering data from one processor to all processors

- Scatter:



Option 2 requires scattering data from one processor to all processors

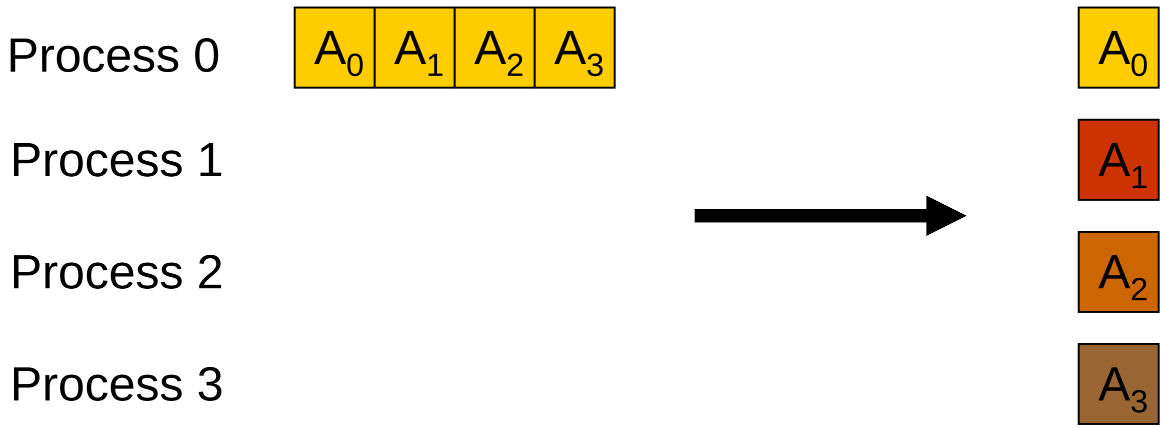
- Scatter:



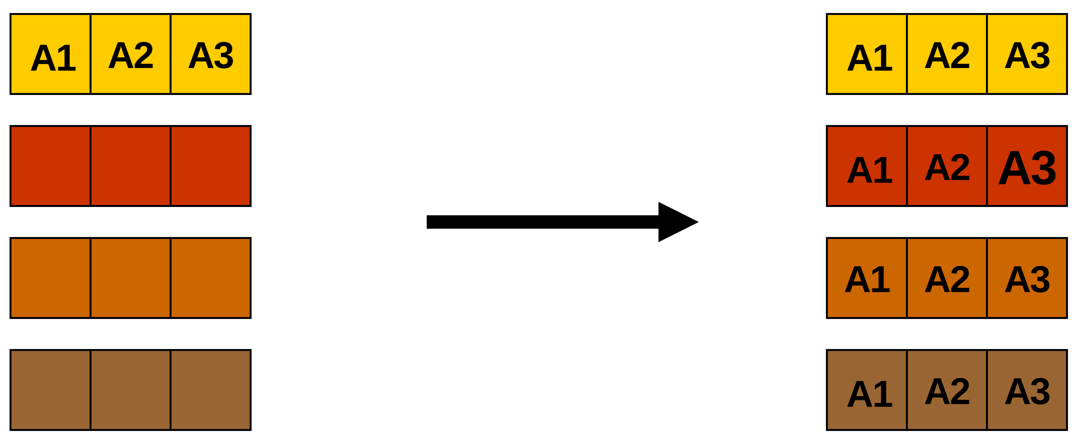
Different concept than broadcast where we wanted the same data sent to all processes

Scatter vs. Broadcast

- Scatter:



- Broadcast



MPI_Scatter syntax meaning: root processor

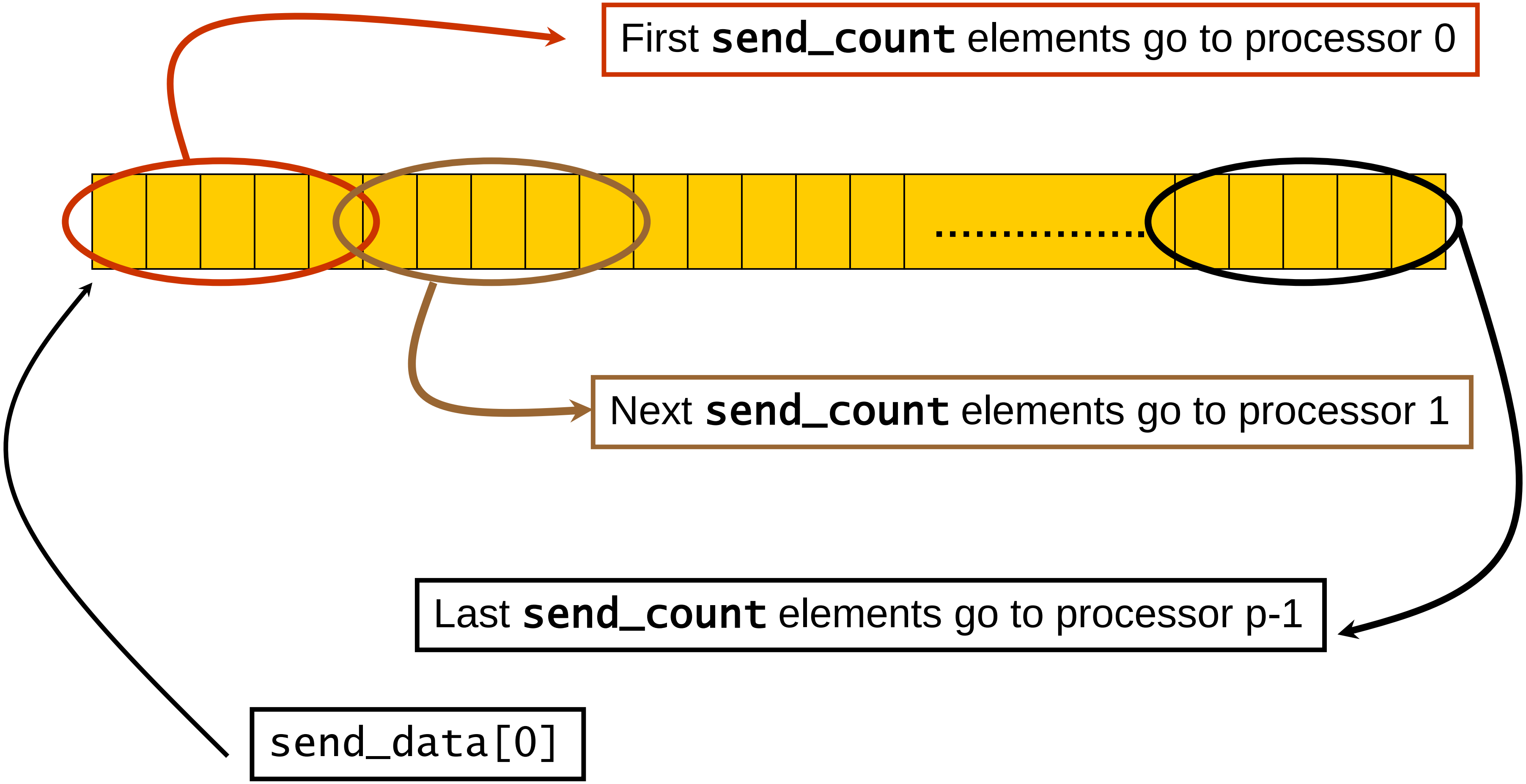
```
MPI_Scatter(
      void*          send_data,
      int            send_count,
      MPI_Datatype   send_type,
      void*          recv_data,
      int            recv_count,
      MPI_Datatype   recv_type,
      int            root,
      MPI_Comm       comm)
```

Starting address of root's local data to be scattered to all processors. Required storage = P*send_count*SizeOf(send_type)

Number of data elements to be sent to each processor

Type of root's local data

MPI_Scatter syntax meaning: root processor



MPI_Scatter syntax meaning: all processors

```
MPI_Scatter(
        void*          send_data,

        int            send_count,

        MPI_Datatype   send_type,

        void*          recv_data,

        int            recv_count,

        MPI_Datatype   recv_type,

        int            root,

        MPI_Comm       comm)
```

Address where scattered data is to be stored. Required storage = recv_count*SizeOf(recv_type)

Number of data elements to be received by each processor, most often = send_count

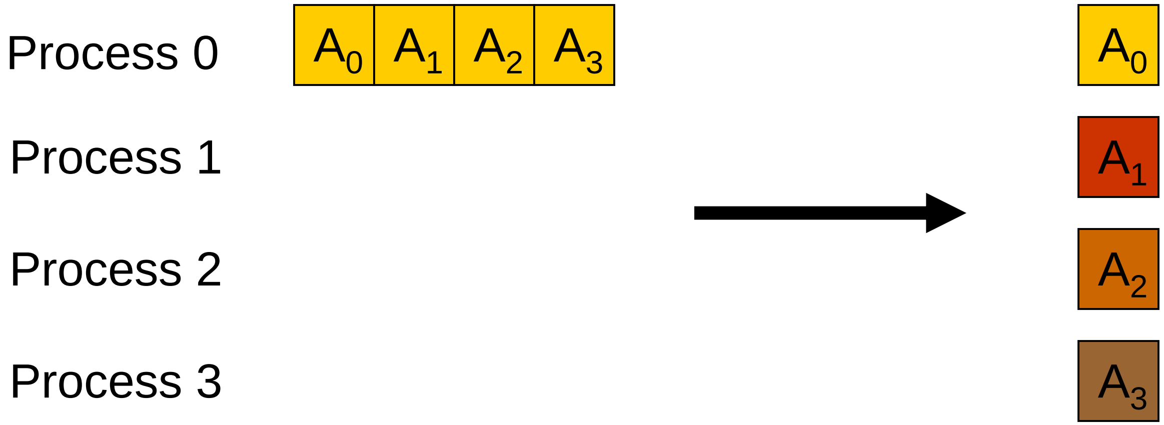
Type of received data, most often = send_type

Original owner of scattered data

Everybody who's involved

Option 2 requires scattering data from one processor to all processors

- Scatter:




```
root = 0;
send_count = 1, recv_count= 1;
MPI_Scatter(A_row, send_count, MPI_FLOAT,
            &A_row_segment, recv_count, MPI_FLOAT,
             root, MPI_COMM_WORLD );
```

MatVec: Parallelize computation of first y-entry, pick an option.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_0 \end{bmatrix}$$

- Option 1: gather x’s to P0, it does computation, no additional communication
- Option 2: scatter a’s to P1,P2 & P3, they do computation, then MPI_Reduce ax’s to get result to P0.

MatVec: To compute its rows, each processor needs all of x.

$$\begin{array}{cccc} A & x & = & y \\ \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \\ a_{40} & a_{41} & a_{42} & a_{43} \\ a_{50} & a_{51} & a_{52} & a_{53} \\ a_{60} & a_{61} & a_{62} & a_{63} \\ a_{70} & a_{71} & a_{72} & a_{73} \end{bmatrix} & \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} & = & \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} \end{array}$$

- Just as P0 needs entire x to compute its y’s, so do all other processes.
- Gather to P0, then Broadcast to all, or better yet call Allgather.

MatVec
process

All **prefix means:**
Result is available on all processes rather than just on root.

Reduce () vs Allreduce()

Gather () vs Allgather()

$$
\begin{array}{ccc}
A & x & = & y \\
\begin{bmatrix}
a_{00} & a_{01} & a_{02} & a_{03} \\
a_{10} & a_{11} & a_{12} & a_{13} \\
a_{20} & a_{21} & a_{22} & a_{23} \\
a_{30} & a_{31} & a_{32} & a_{33} \\
a_{40} & a_{41} & a_{42} & a_{43} \\
a_{50} & a_{51} & a_{52} & a_{53} \\
a_{60} & a_{61} & a_{62} & a_{63} \\
a_{70} & a_{71} & a_{72} & a_{73}
\end{bmatrix}
&
\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}
& = &
\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix}
\end{array}
$$

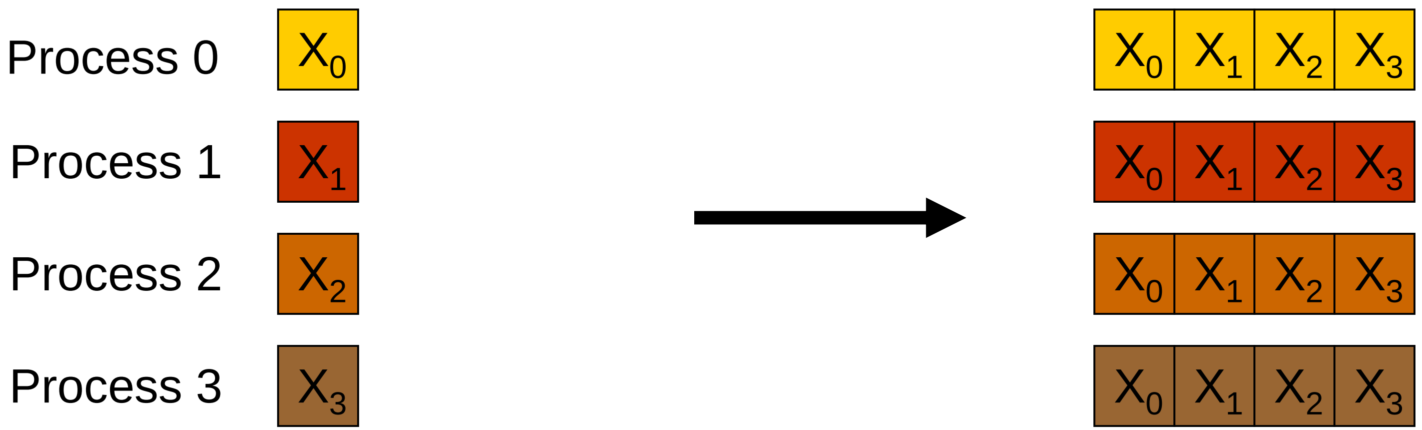
- Just as P0 needs entire x to compute its y's, so do all other processes.
- Gather to P0, then Broadcast to all, or better yet call Allgather.

MPI_Allgather syntax

```
MPI_Allgather(
        void*          send_data,

        int            send_count,

        MPI_Datatype   send_type,

        void*          recv_data,

        int            recv_count,

        MPI_Datatype   recv_type,

        MPI_Comm       comm)
```

MatVec: Gathering all of x to each processor.

- Allgather:




```
send_count = 1, recv_count= 1;
MPI_Allgather( &X_local, send_count, MPI_FLOAT,
               X_global, recv_count, MPI_FLOAT,
                MPI_COMM_WORLD );
```

MPI data arguments are ***addresses***.

- Allgather:

X_local **X_global**

Process 0 X_0 X_0 X_1 X_2 X_3

Process 1 X_1 X_0 X_1 X_2 X_3

Process

Process

C -- arguments one and four are addresses
The first argument for example should be
- **&X_local if X_local is a single scalar float or double**
- **X_local or &X_local[0] if X_local is an array of floats or doubles**

```
send_count = 1, recv_count= 1;
MPI_Allgather( &X_local, send_count, MPI_FLOAT,
               X_global, recv_count, MPI_FLOAT,
                MPI_COMM_WORLD );
```

MPI data arguments are ***addresses***.

- Allgather:

Can use "IN PLACE" option to optimize storage
Arguments 2 and 3 are ignored
The input data (what was X_local) for each process is assumed to be stored in the area of X_global where that process would receive its own contribution.

```
send_count = 1, recv_count= 1;
MPI_Allgather( MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
               X_global, recv_count, MPI_FLOAT,
                MPI_COMM_WORLD );
```

MatVec: Parallel Code

```
MPI_Allgather(local_x, local_n, MPI_FLOAT,
              global_x, local_n, MPI_FLOAT,
              MPI_COMM_WORLD);
for (i=0; i < local_m; i++)
      {
              local_y[i]=0.0;
              for(j=0; j < n; j++)
                     local_y[i] += local_A[i*n+j]*global_x[j];
      }
```