CSE 891 - Section 1: Parallel Computing - Fundamentals and Applications

Fall 2014 - Lecture 8:

Programming using the Message Passing Paradigm

H. Metin Aktulga

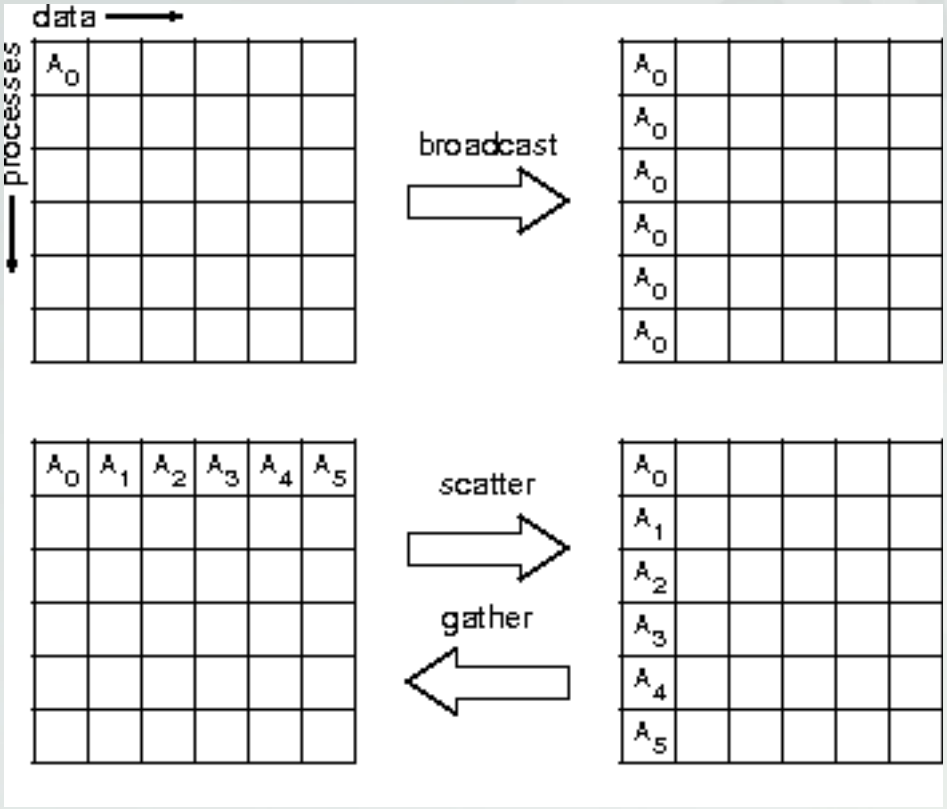
hma@cse.msu.edu

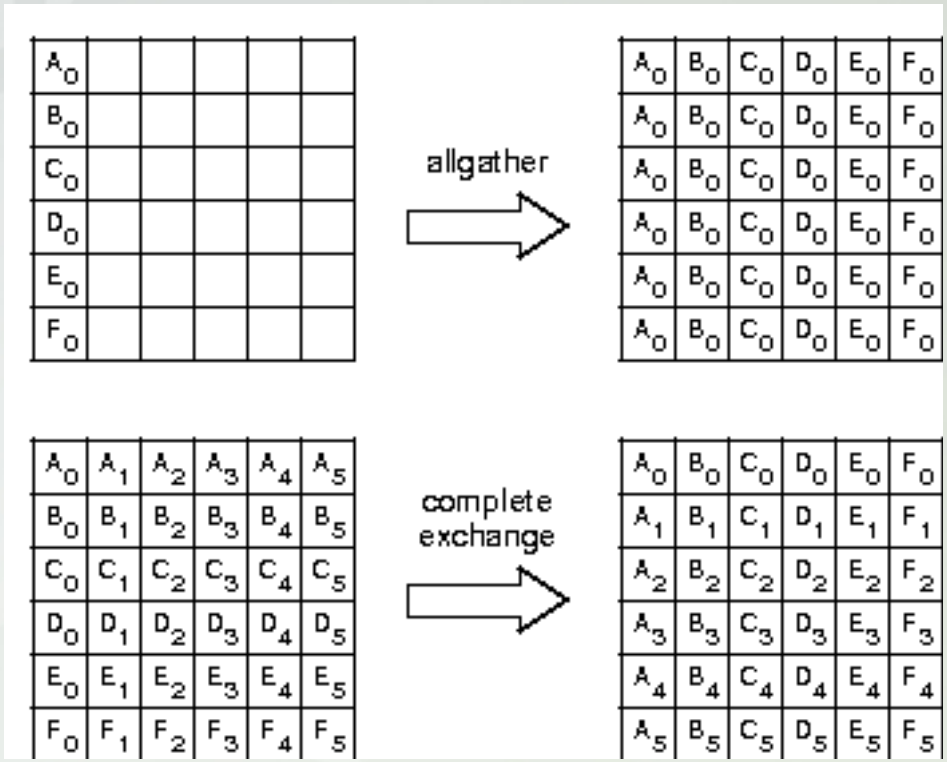
517-355-1646

Lecture 7- Summary

- Programming Using the Message Passing Paradigm
  - Asynchronous Send & Recv
  - Parallel Sum Reduction
    - 3 different algorithms
    - Parallel complexity analyses
  - Collective Communications in MPI
    - Barrier, Broadcast, Reduce, All_reduce
    - Gather, Scatter, All_gather, Alltoall
    - Variable variants

Lecture 7- Summary





Groups and Communicators

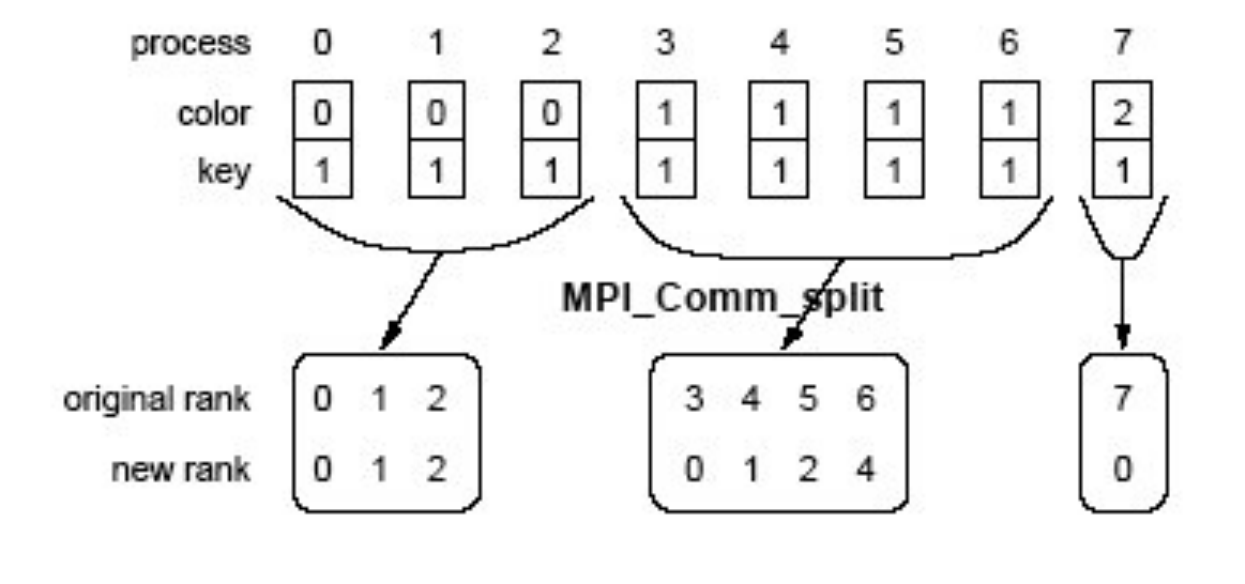
- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color,
                   int key, MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key.

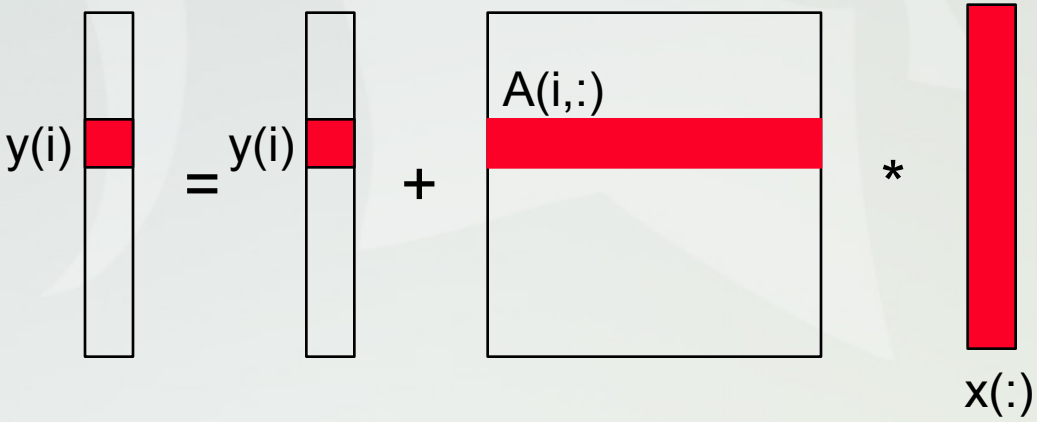
Groups and Communicators

Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.



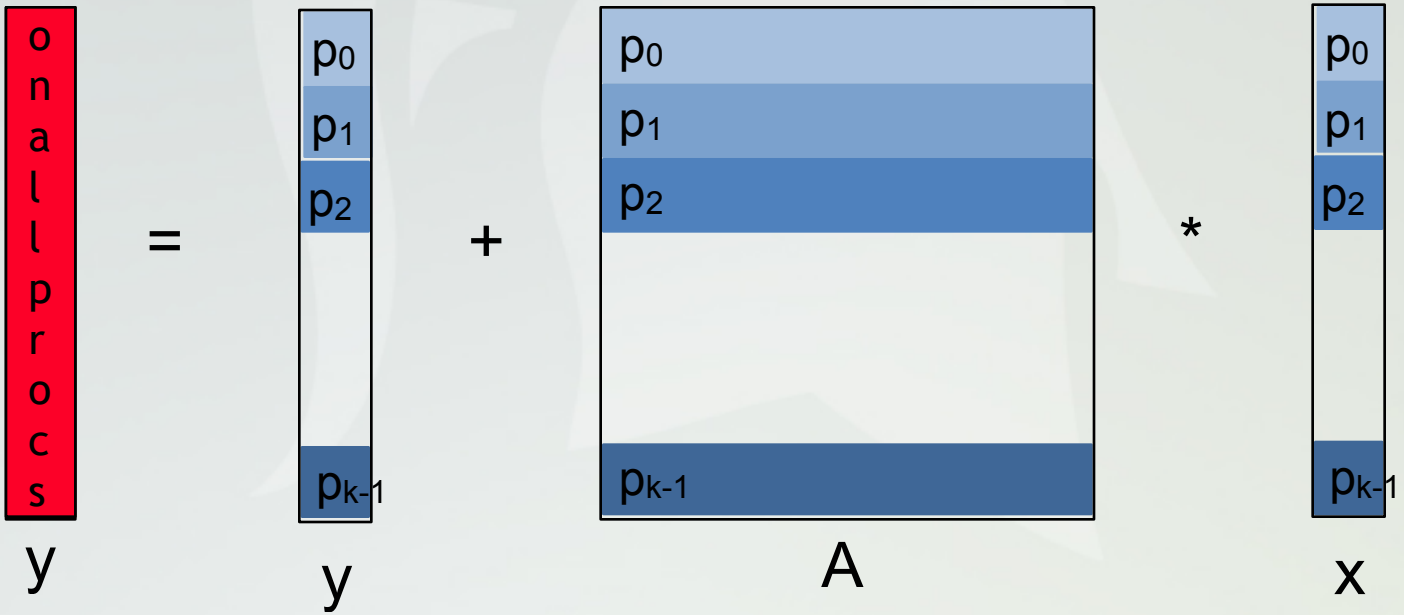
An MPI Example: Parallel Matrix-Vector Multiply

```
{implements y = y + A*x}
for i = 1:n
   for j = 1:n
      y(i) = y(i) + A(i,j)*x(j)
```



An MPI Example: Parallel Matrix-Vector Multiply

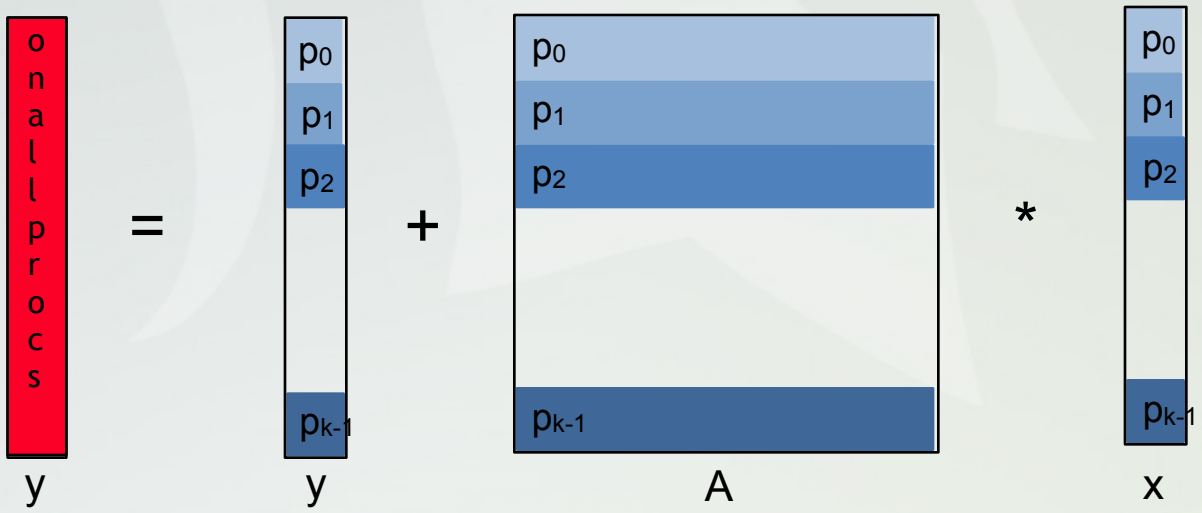
Assume we have *k* processors with the following initial data layout



An MPI Example: Parallel Matrix-Vector Multiply

A high-level parallel algorithm:

- collect the entire *x* vector on all processors
- perform local matrix-vector multiplications - y(i) = y(i) + A(i,:) * x(:)
- collect partial output vectors *y* on all processors



An MPI Example: Parallel Matrix-Vector Multiply

```
#include <mpi.h>
#include <stdio.h>
#include <sys/time.h>

#define N 3000

int main(int argc, char *argv[])
{
  int P, i, myrank, M;
  int i, j;
  double t_start, t_end;
  double **my_A, *my_x, *my_y, *x, *y;

  /* Initializations */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  M = N / P; // Assuming N is a multiple of P
  my_x = (double*) malloc(M * sizeof(double));
  my_y = (double*) malloc(M * sizeof(double));
  x = (double*) malloc(N * sizeof(double));
  y = (double*) malloc(N * sizeof(double));

  A = (double**) malloc(M * sizeof(double*));
  for (i = 0; i < M; ++i)
    A[i] = (double*) malloc(N * sizeof(double));

  srand(time(NULL));
  for (i = 0; i < M; ++i) {
    my_x[i] = rand()/N;
    my_y[i] = rand()/N;
    for (j = 0; j < N; ++j)
      A[i][j] = rand()/N;
  }
```

An MPI Example: Parallel Matrix-Vector Multiply

```
/* collect x vector on all processors */
/* first copy my_x into x */
for (j = 0; j < M; ++j)
  x[myrank*M +j] = my_x[j];

for (i = 0; i < P; ++i)
  MPI_Bcast(&(x[myrank*M]), M, MPI_DOUBLE, i, MPI_COMM_WORLD);
```

What is the complexity of the operations above? $O(N/P+P\log P)$

What is the total communication volume? $\Omega(PN)$

Is there a better way? YES!

```
/* collect x vector on all processors */
MPI_Gather(my_x, M, MPI_DOUBLE, x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Or better yet:

```
/* collect x vector on all processors */
MPI_Allgather(my_x, M, MPI_DOUBLE, x, N, MPI_DOUBLE, MPI_COMM_WORLD);
```

An MPI Example: Parallel Matrix-Vector Multiply

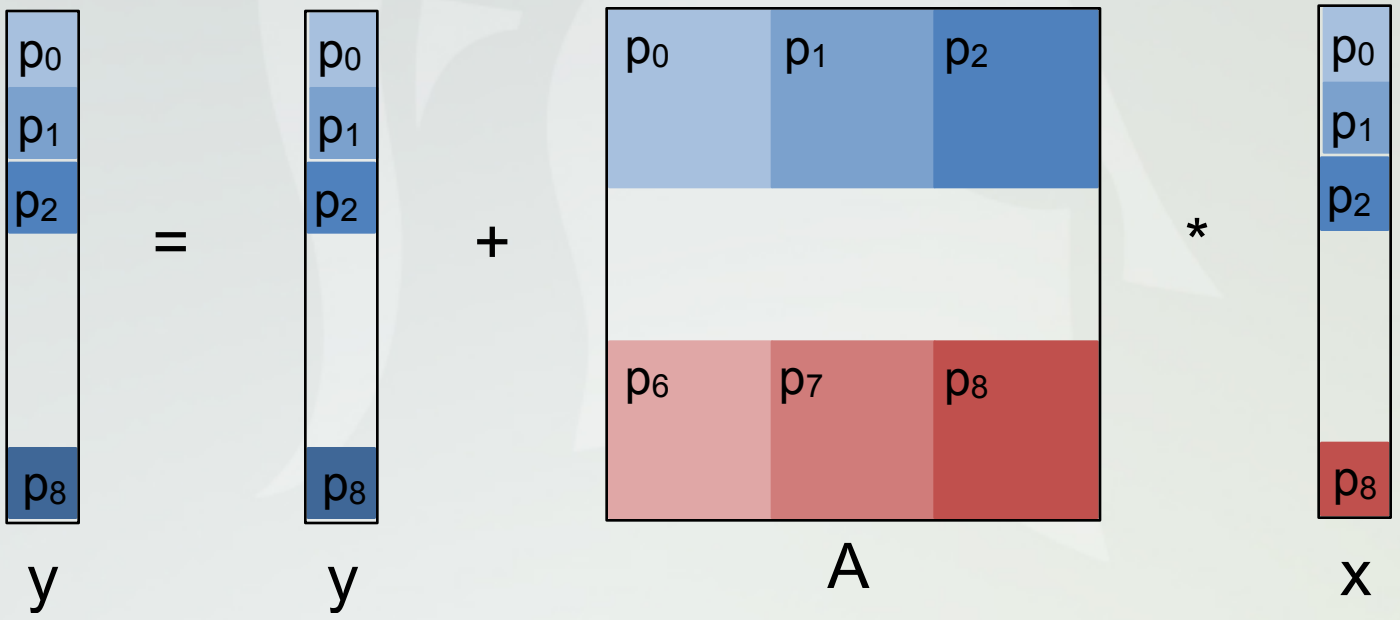
```
/* local computations */
for (i = 0; i < M; ++i)
  for (j = 0; j < N; ++j)
    my_y[i] += A[i][j] * x[j];

/* collect partial results at all processors */
MPI_Allgather(my_y, M, MPI_DOUBLE, x, N, MPI_DOUBLE, MPI_COMM_WORLD);
```

MPI_Allgather is simpler to implement and potentially will have a better execution time than a series of broadcasts. Nevertheless, in the current scheme, $\Omega(PN)$ is the lower bound on the communication volume and it is not likely to scale well.

Can we do better?

An MPI Example: Parallel Matrix-Vector Multiply

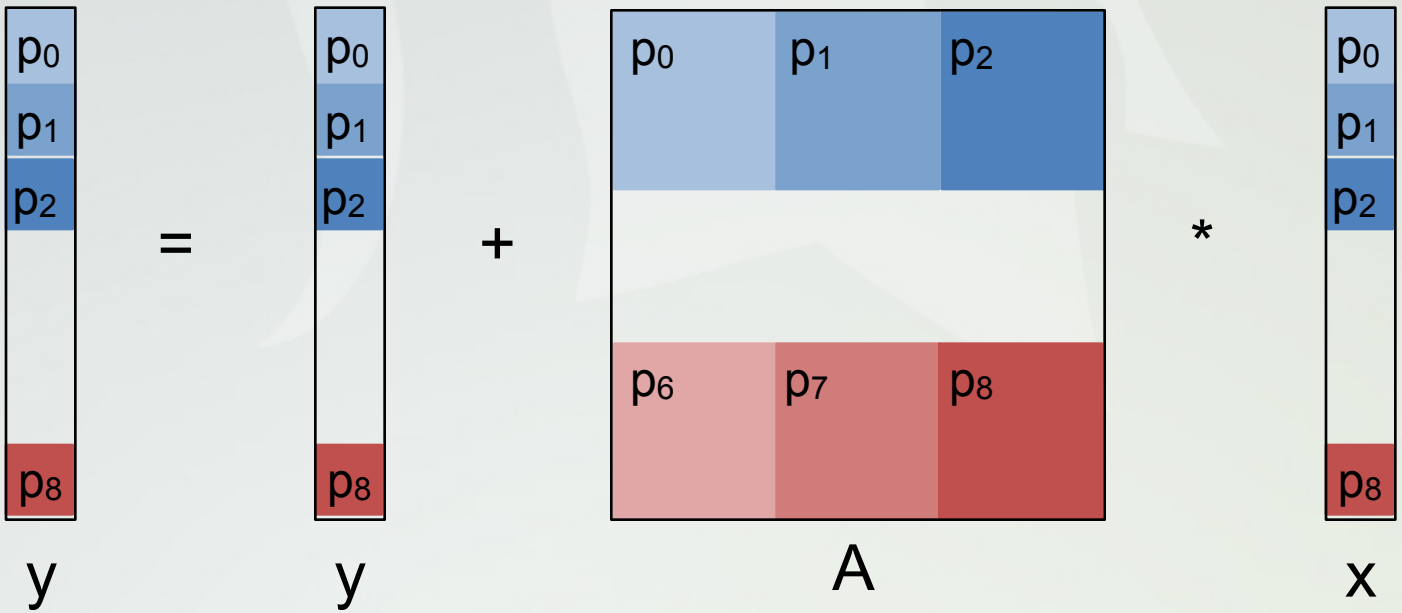
Assume we have *9* processors using a 2D decomposition:



An MPI Example: Parallel Matrix-Vector Multiply

How is a 2D decomposition different from 1D decomposition?

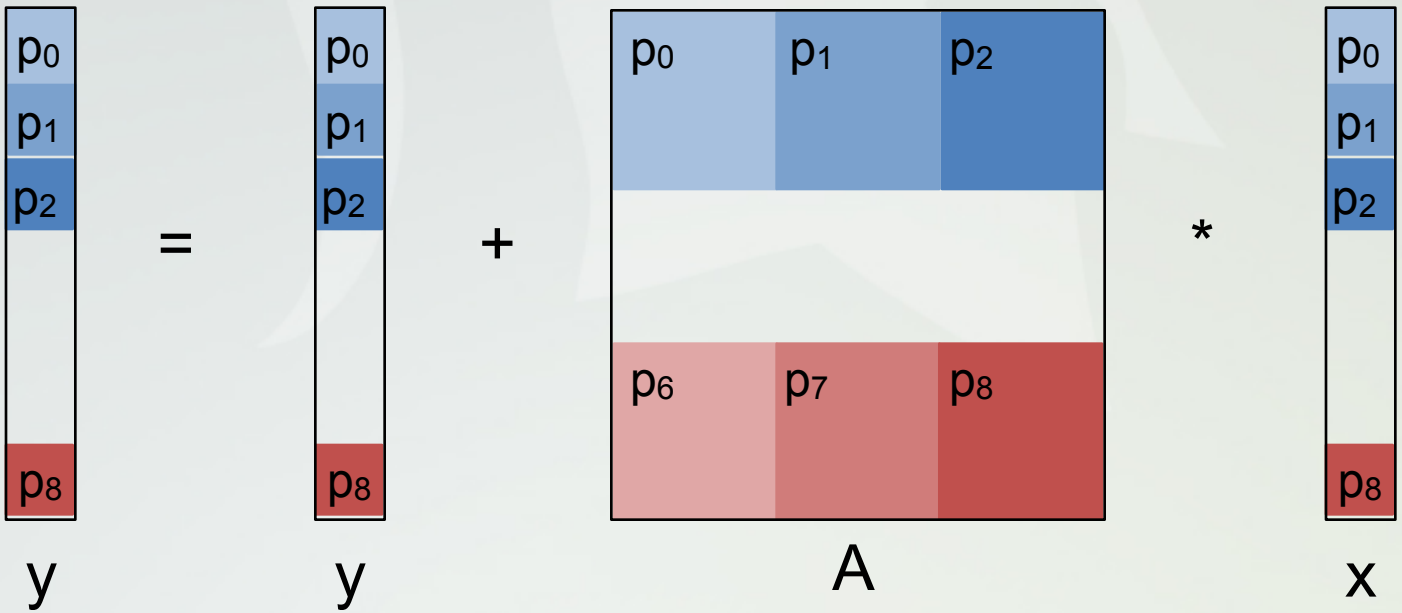
- Number of matrix elements? Same, N^2/P. So the computational work does not change.
- How about communication volume?



An MPI Example: Parallel Matrix-Vector Multiply

Each processor still starts with an N/P partition of *x*, but now each one needs only a partition of *x* which is of length $N/P^{1/2}$.

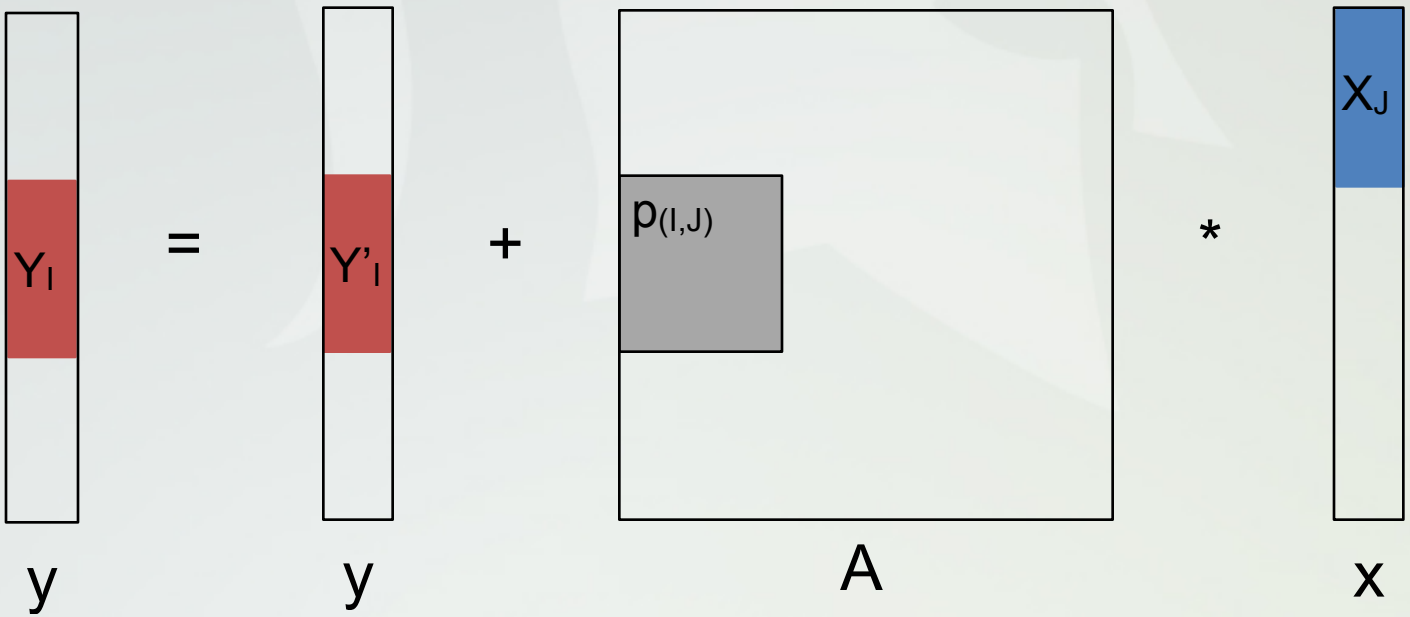
So input communication volume is now $\Omega(P(N/P^{1/2} - N/P)) = \Omega(P^{1/2}N)$ - compare with the 1D partitioning case which was $O(PN)$!



An MPI Example: Parallel Matrix-Vector Multiply

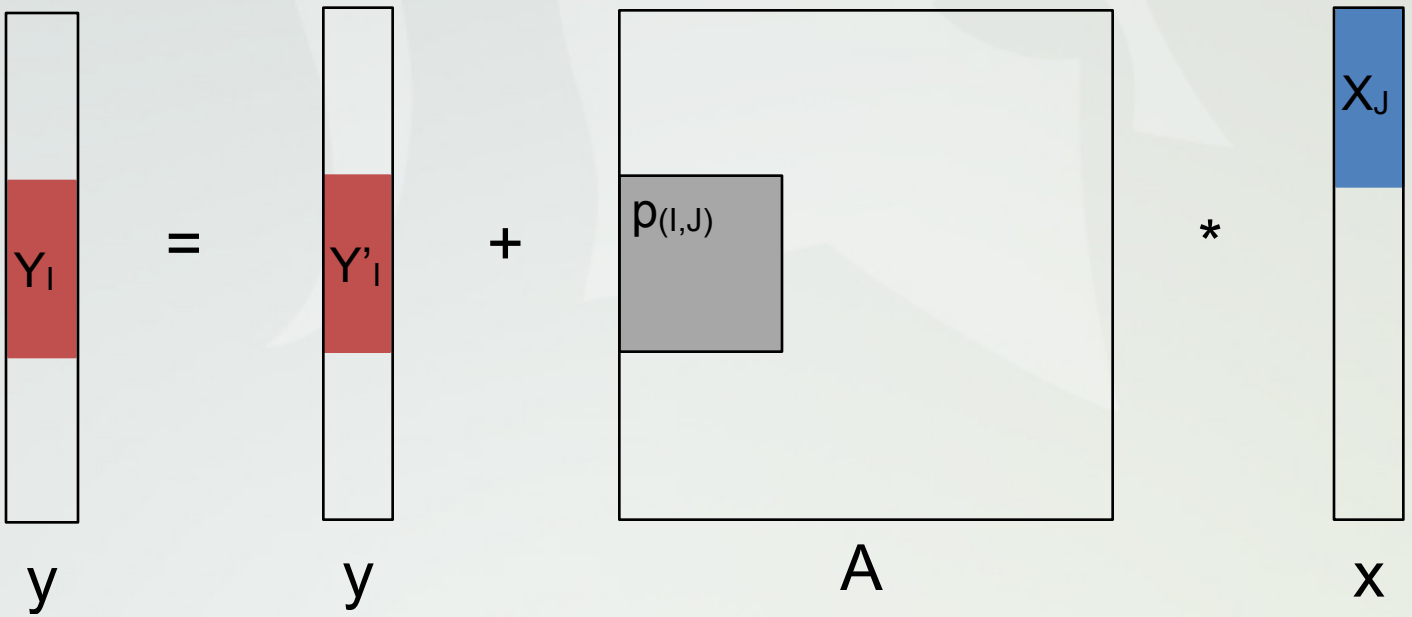
Let us consider the processor at the *i*th row and *j*th column:

- It needs to talk to $P^{1/2}$ other processors on its column for input
- So the input vector can be collected through a gather along rows



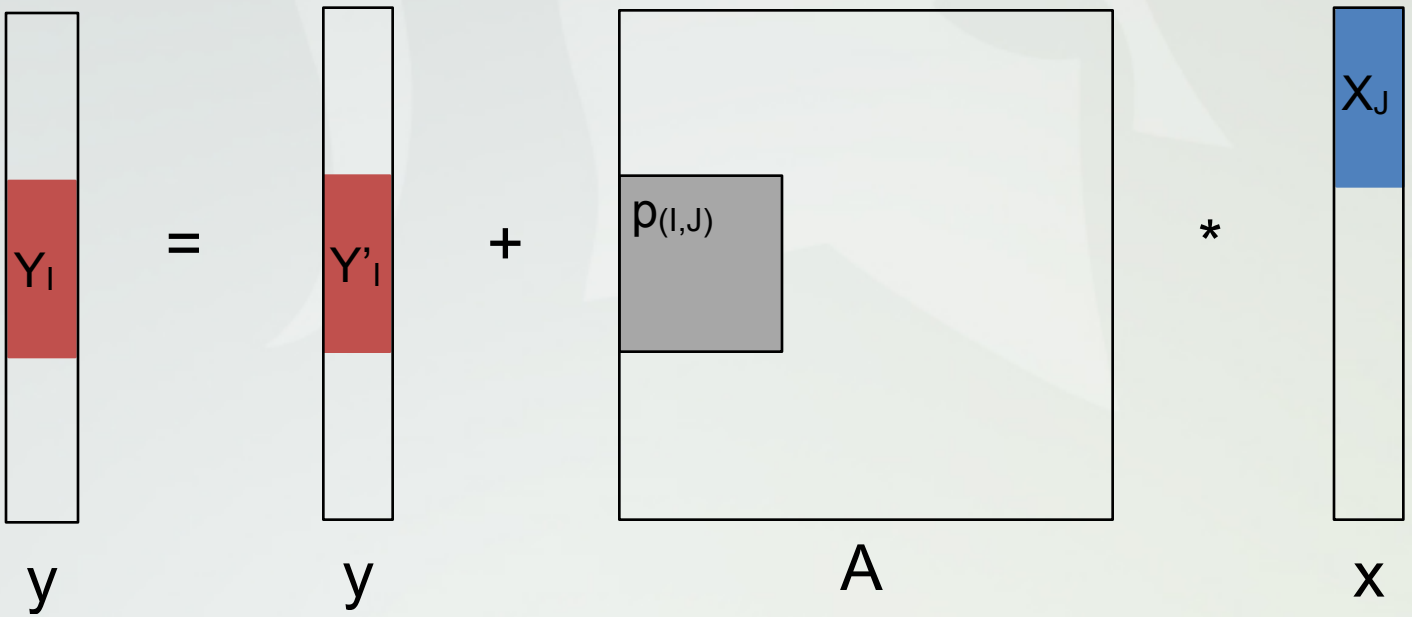
An MPI Example: Parallel Matrix-Vector Multiply

- But what about the output vector?
- Matrix-vector multiply in block notation: Y'[I] += A[I][J] * X[J]
- The output of each processor is only partial, hence denoted by Y'$_I$
- Need a reduction this time along rows



An MPI Example: Parallel Matrix-Vector Multiply

- So instead of the default communicator `MPI_COMM_WORLD`, we now need column and row communicators!
- This can be implemented using `MPI_Comm_split`
- Use the row and and column index values as colors



An MPI Example: Parallel Matrix-Vector Multiply

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#define N 3000

int main(int argc, char *argv[])
{
  int P, i, myrank, M, myrow, mycol;
  int i, j;
  double t_start, t_end;
  double **my_A, *my_x, *my_y, *x, *y;
  MPI_Comm rowcomm, colcomm;

  /* Initializations */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  sqrtP = sqrt(P); // Assuming P is a perfect square
  M = N / P; // Assuming N is a multiple of P
  bigM = N / sqrtP; // Assuming N is a multiple of P
  my_x = (double*) malloc(M * sizeof(double));
  my_y = (double*) malloc(M * sizeof(double));
  x = (double*) malloc(bigM * sizeof(double));
  y = (double*) malloc(bigM * sizeof(double));

  A = (double**) malloc(bigM * sizeof(double*));
  for (i = 0; i < bigM; ++i)
    A[i] = (double*) malloc(bigM * sizeof(double));

  srand(time(NULL));
  for (i = 0; i < M; ++i) {
    my_x[i] = rand()/N;
    my_y[i] = rand()/N;
  }
  for (i = 0; i < bigM; ++i)
    for (j = 0; j < bigM; ++j)
      A[i][j] = rand()/N;
```

An MPI Example: Parallel Matrix-Vector Multiply

```
/* setup communication groups */
myrow = myrank / sqrtP;
mycol = myrank % sqrtP;
MPI_Comm_split(MPI_COMM_WORLD, myrow, myrank, &rowcomm);
MPI_Comm_split(MPI_COMM_WORLD, mycol, myrank, &colcomm);

/* collect x vectors along column comms, y vector along row comms */
MPI_Allgather(my_x, M, MPI_DOUBLE, x, bigM, MPI_DOUBLE, colcomm);
MPI_Allgather(my_y, M, MPI_DOUBLE, y, bigM, MPI_DOUBLE, rowcomm);

/* local computations */
for (i = 0; i < bigM; ++i)
  for (j = 0; j < bigM; ++j)
    y[i] += A[i][j] * x[j];

/* collect partial results along rows */
for (i = 0; i < sqrtP; ++i)
  MPI_Reduce(&(y[M*i]), my_y, M, MPI_DOUBLE, MPI_SUM, i, rowcomm);
```

Or there is a (potentially) more efficient way :

```
/* collect partial results along rows */
MPI_Reduce_scatter(y, my_y, array_of_M, MPI_DOUBLE, MPI_SUM, rowcomm);
```

An MPI Example: Parallel Matrix-Vector Multiply

- So what is overall communication volume of the 2D algorithm?
- We have seen that decisions made at an algorithmic level have important consequences in terms of communication overheads
- Not only that, but how we implement the algorithm is important! (a series of bcasts vs. allgather, or a series of reduces vs. reduce_scatter)

Creating and Using Cartesian Topologies

- We can create cartesian topologies using the function:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods, int reorder,
                    MPI_Comm *comm_cart)
```

- This function takes the processes in the old communicator and creates a new communicator with dims dimensions.

- Each processor can now be identified in this new cartesian topology by a vector of dimension dims.

Creating and Using Cartesian Topologies

- Since sending and receiving messages still require (one-dimensional) ranks, MPI provides routines to convert ranks to cartesian coordinates and vice-versa.

```
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
                   int *coords)

int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```

- The most common operation on cartesian topologies is a shift. To determine the rank of source and destination of such shifts, MPI provides the following function:

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step,
                   int *rank_source, int *rank_dest)
```

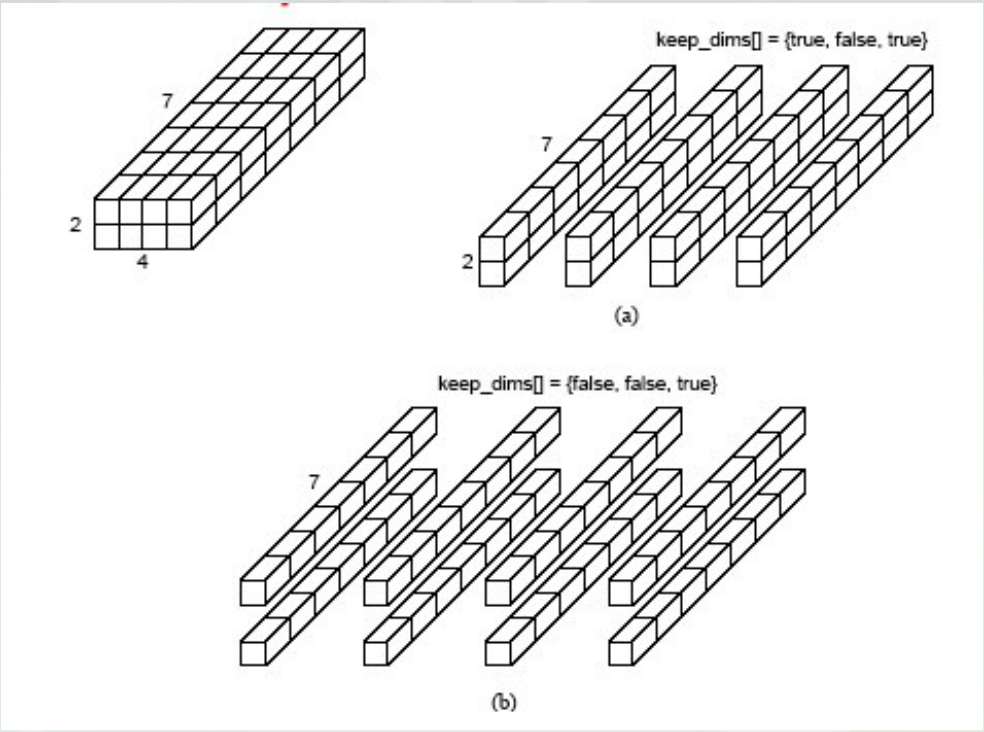
Creating and Using Cartesian Topologies

- MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
                 MPI_Comm *comm_subcart)
```

- If `keep_dims[i]` is true (non-zero value in C) then the `i`th dimension is retained in the new sub-topology.
- The coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained.

Creating and Using Cartesian Topologies



Splitting a Cartesian topology of size 2 x 4 x 7 into

(a) four subgroups of size 2 x 1 x 7, and

(b) eight subgroups of size 1 x 1 x 7.

How to time your application?

- C provide gettimeofday() - not so practical
- MPI alternative - MPI_Wtime()
- Make sure to put a barrier before starting the timer and right before stopping the timer!
- Average over multiple executions for small tasks!