An Intro to Distributed Memory Computing and MPI

Jack Poulson, Rob Schreiber
Stanford ICME
Class 4

Collectives in Pictures

A			
B			
C			
D			

broadcast

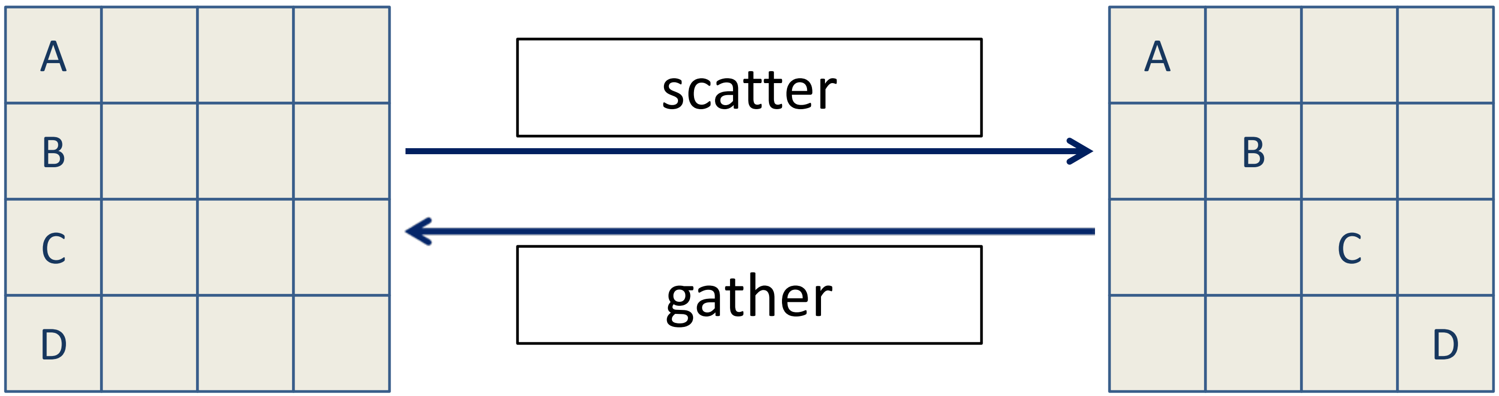
A	A	A	A
B	B	B	B
C	C	C	C
D	D	D	D

28			
32			
36			
40			

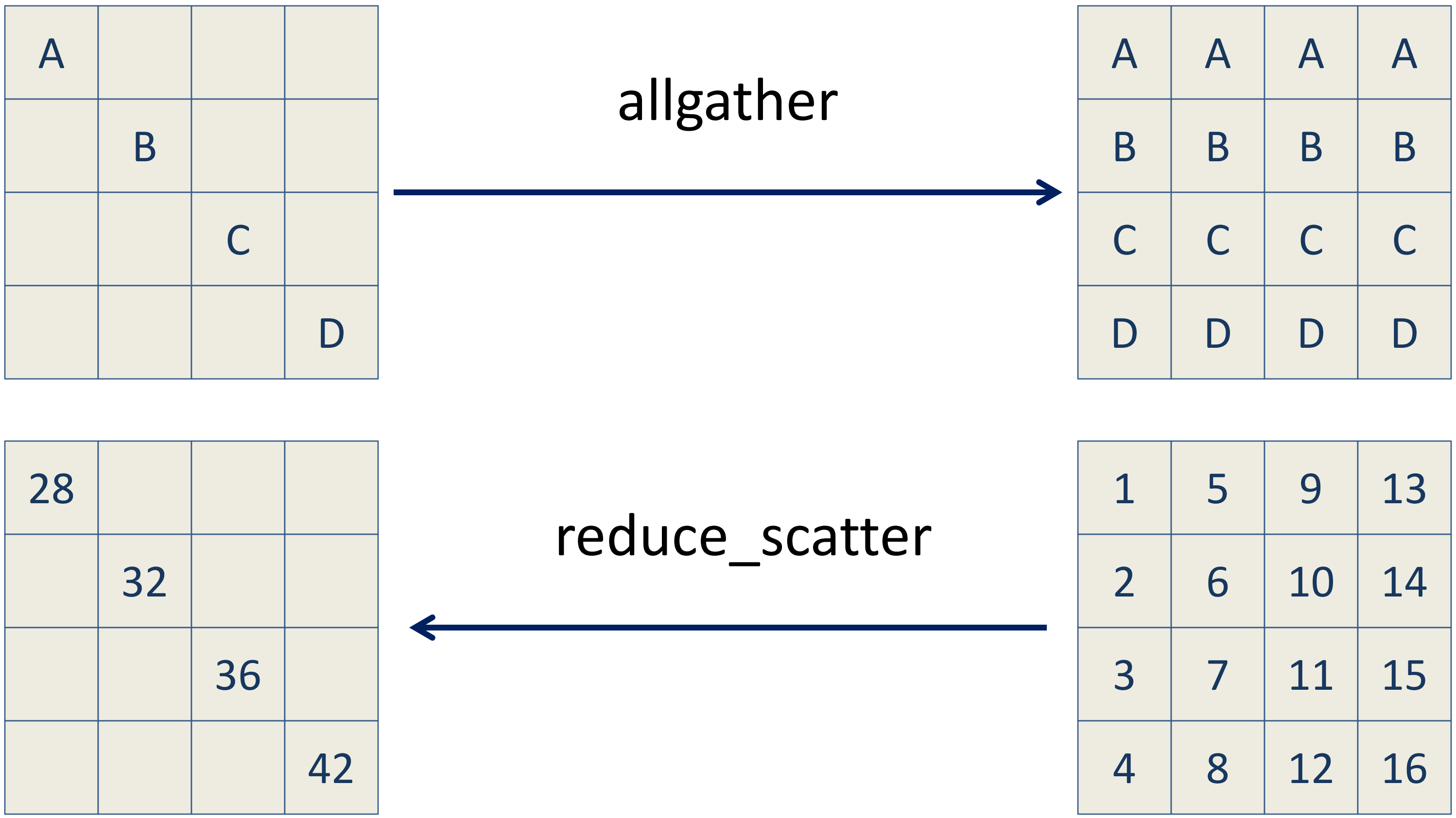
reduce

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

Collectives in Pictures



More pictures



Theorem: broadcast $=$ allgather $\circ$ scatter

Suppose all have a local value x, a vector, of length nitems

All want the sum of these vectors, sumx

MPI_ALLREDUCE(x, sumx, nitems,
    MPI_DOUBLE, MPI_SUM, MCW)

Operators: SUM, PROD, LAND, BAND, MAXLOC

More pictures

28	28	28	28
32	32	32	32
36	36	36	36
40	40	40	40

allreduce

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

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

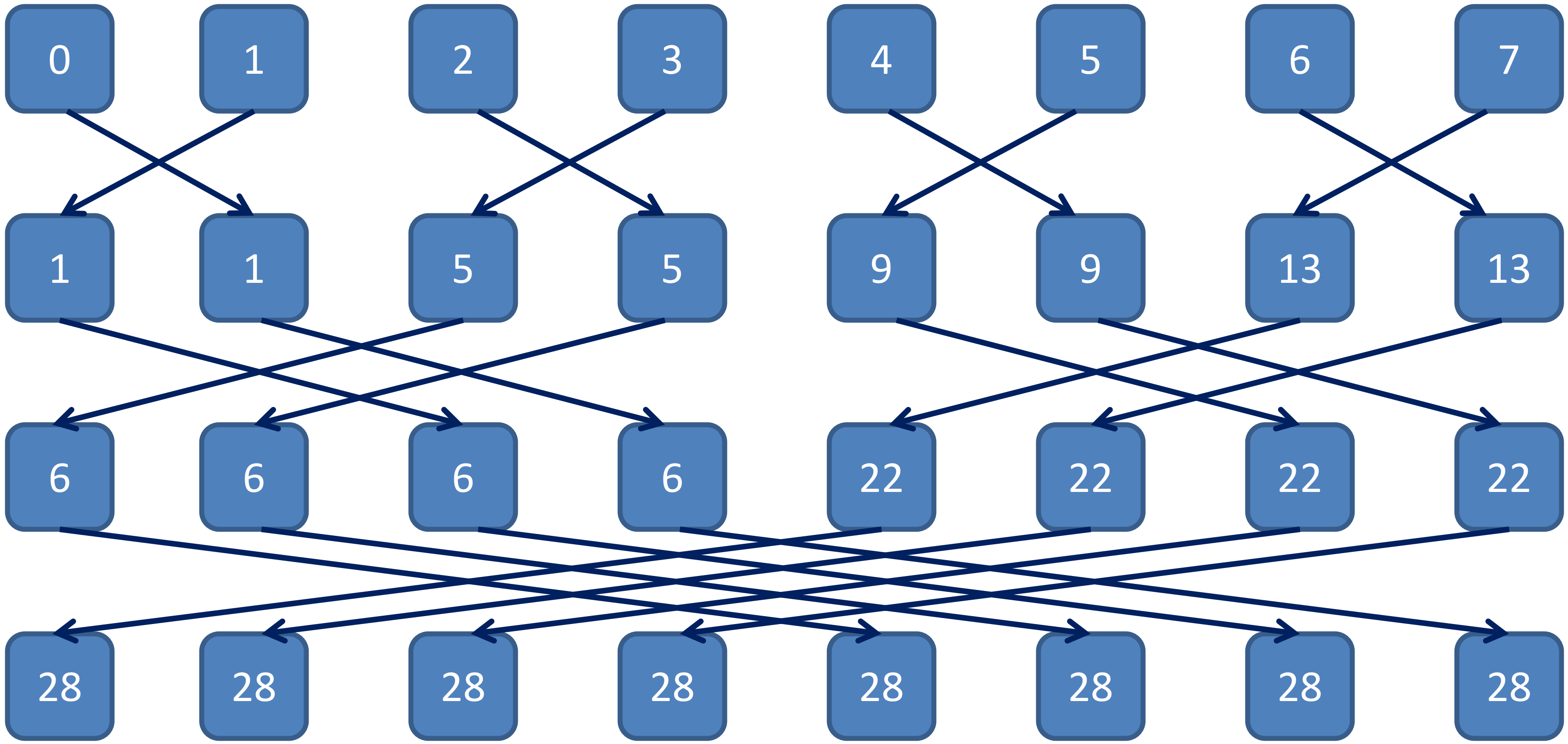
all_to_all

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

Theorem: allreduce $=$ broadcast $\circ$ reduce

$=$ allgather $\circ$ scatter $\circ$ reduce

Butterfly Allreduce – for short vectors



$T_{comm} \sim \log P\,(\alpha + \beta\, n)$ – communicate n-vectors

$T_{comm} \sim 2 \log P\, \alpha + 2\, \beta\, n$ – vector length halves

A collective operation – all ranks in the communicator must call it or nothing happens.

Blocking! Results are in the recvbuffer when it exits.

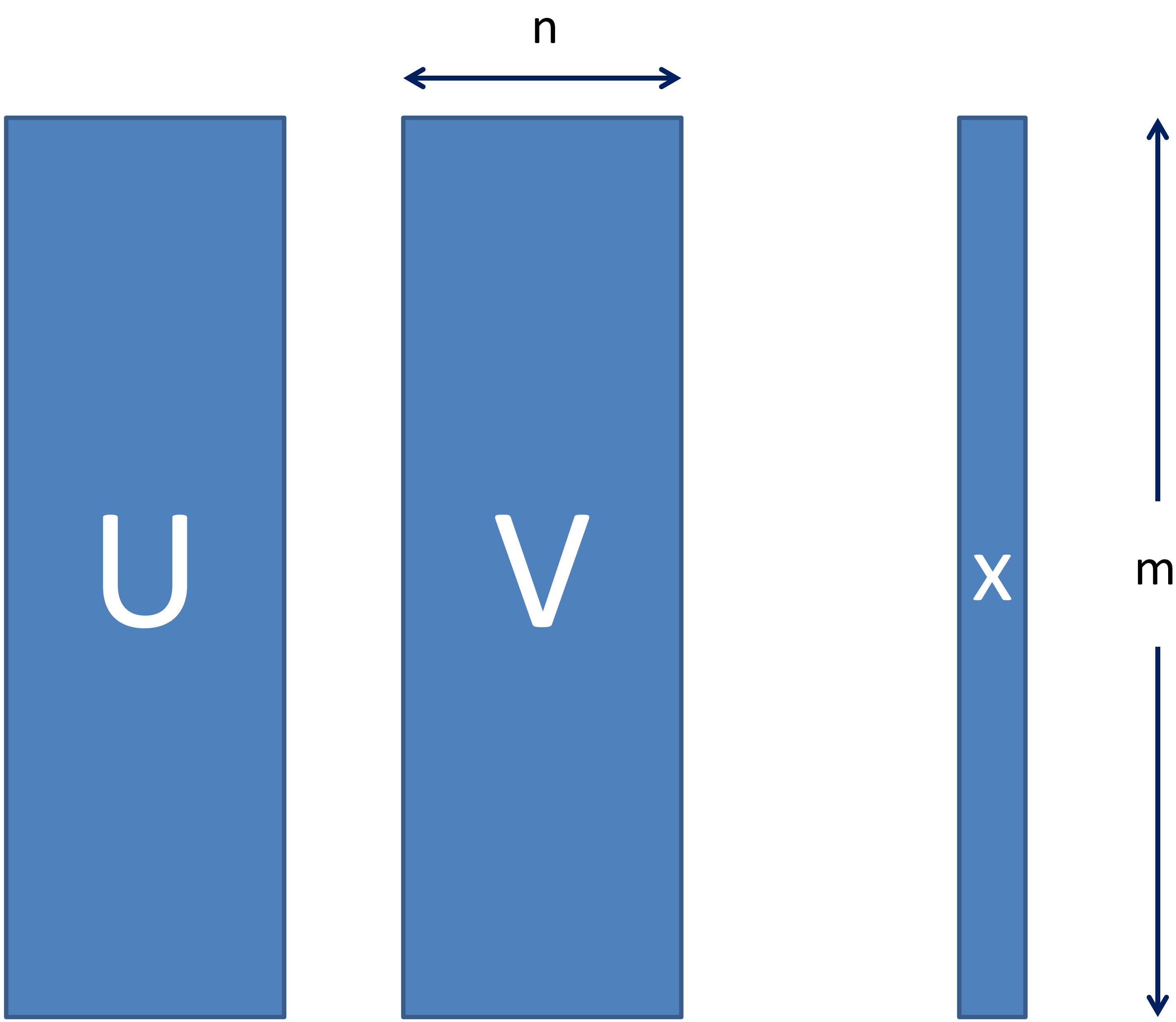
Use the MPI collectives when they match your needs!

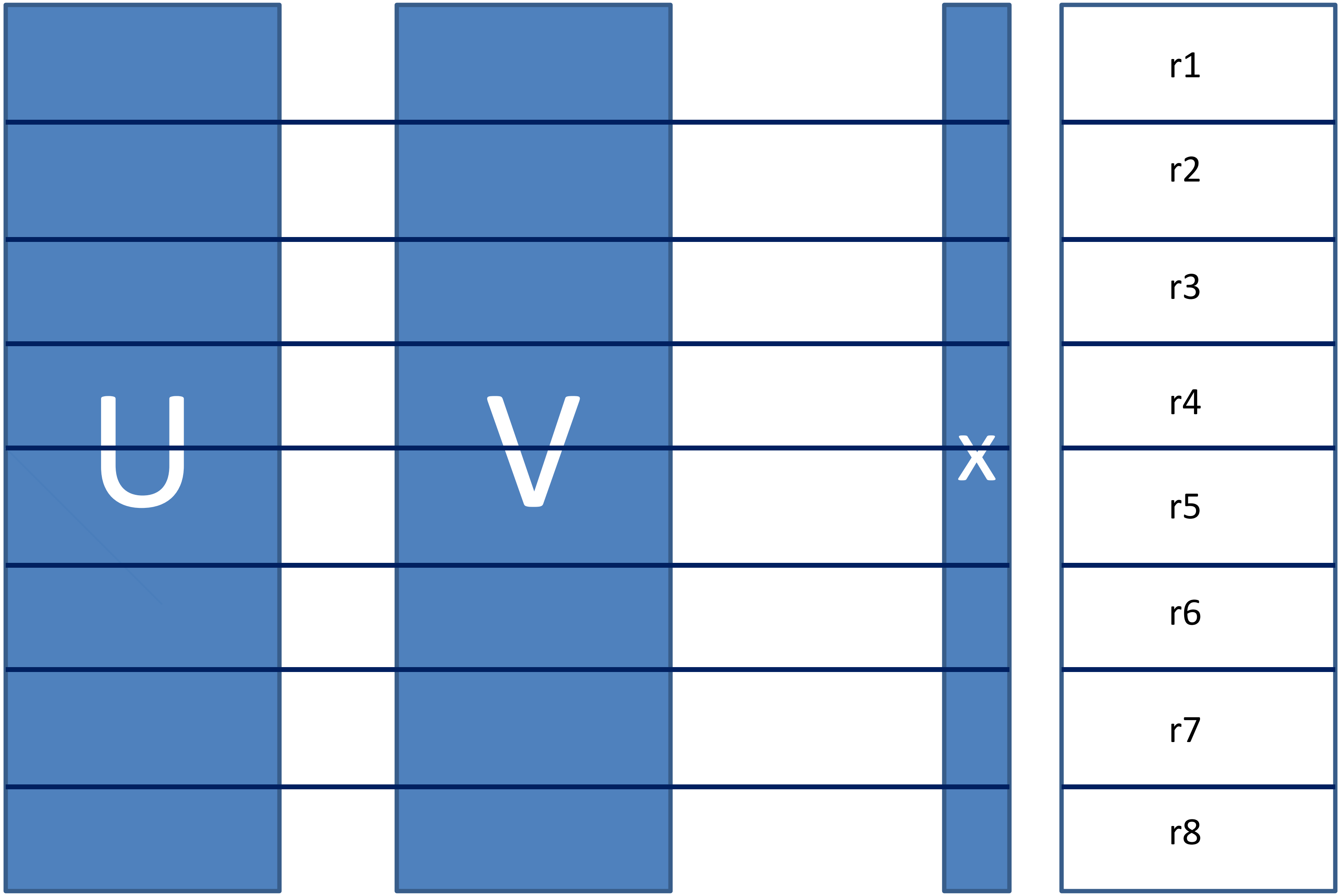
Using Collective Communication

Let U and V be m x n matrices with the TS property – they are tall and skinny, and let x be an m-vector.

Compute y = U V’ x.

(Matlab notation. V’ is the transpose of V)


n
U
V
x
m


U
V
x
r1
r2
r3
r4
r5
r6
r7
r8

They are distributed by rows across the ranks of an MPI communicator

How did they get that way?

Your program makes it so – you have total control in MPI over the distribution of the conceptual data structures. Your program directly manipulates their local “shadows.”

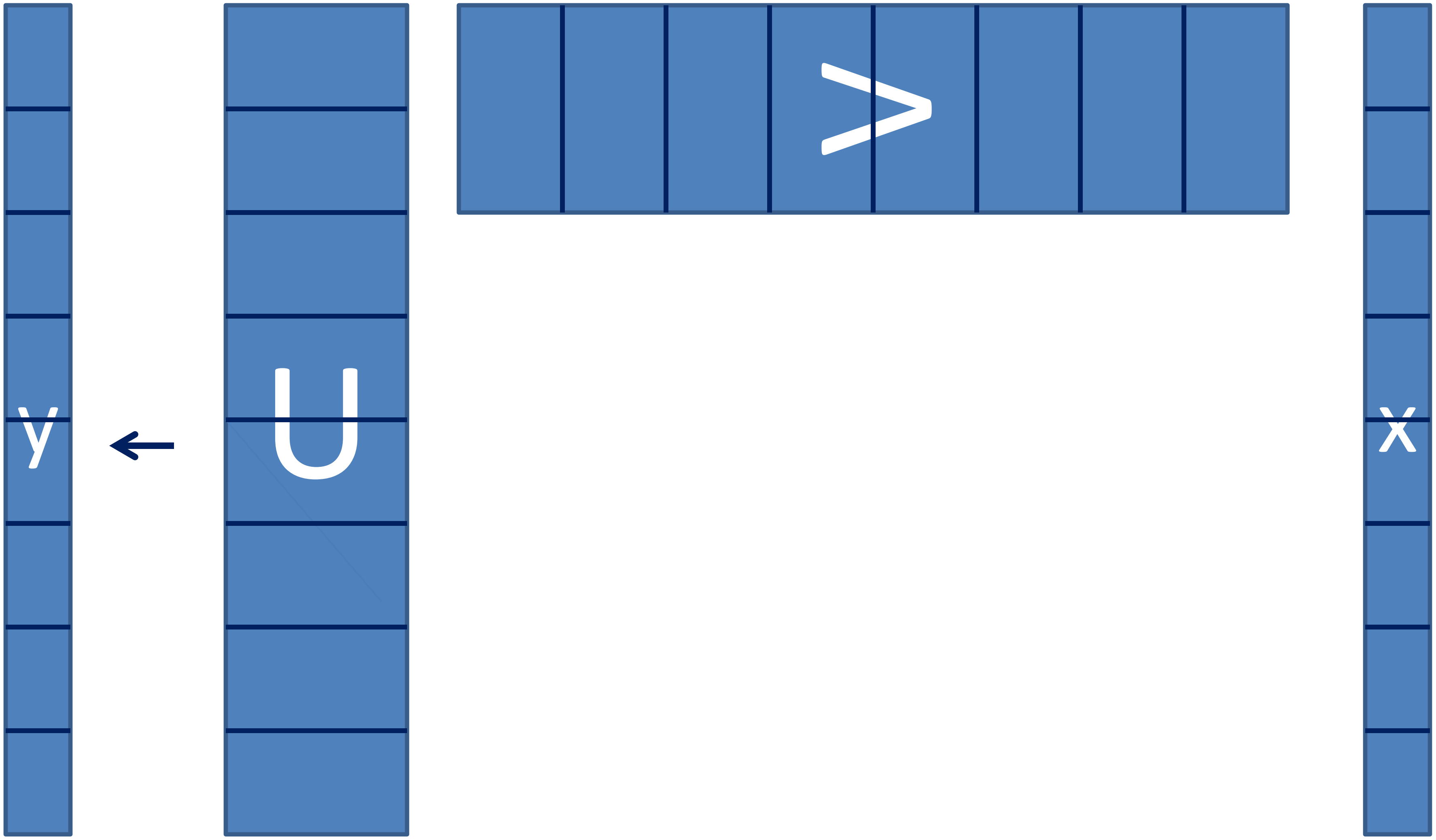
```
void matrix_ops(int m, int n, np, me)
{  int mym;
   double *Uloc, *Vloc, *xloc, *z;

   /* how many rows of U, V, x do I have */
   mloc = m / np;                     // round down

   /* first processors need an extra row */
   if (me < ( m – mloc*np ) ) mloc++;

   Uloc = (double *) malloc(mloc*n*sizeof(double));
   z = (double *) malloc(n * sizeof(double));
     …
}
```

Problem: Compute $y = U V' x$

$$z = V' x$$

$$y = U z$$

y should also have the by-rows distribution.


y
U
V
x

- What we want:
  1) Locally: $z_{loc} = V_{loc}' x_{loc}$
  2) Add up everyone's z_{loc}
  3) Give everyone a copy of the sum of these: z
  4) Locally: $y_{loc} = U_{loc} z$

  – This is the distributed result vector
  – How do we do Steps 2, 3?

MPI_Allreduce(zloc, z, mloc, mpidbl, MPI_SUM, mcw)

Communicators

A communicator defines the collection in all collectives

→ we may want to construct subset communicators to do collectives on subsets of nodes

A communicator is like a name space for tags

→ we may want to construct a communicator to protect a libraries tag space from the calling code's tag space, and vice versa.

Cloning a Communicator

MPI_Comm Mars;

MPI_Comm_dup(MPI_COMM_WORLD, &Mars);

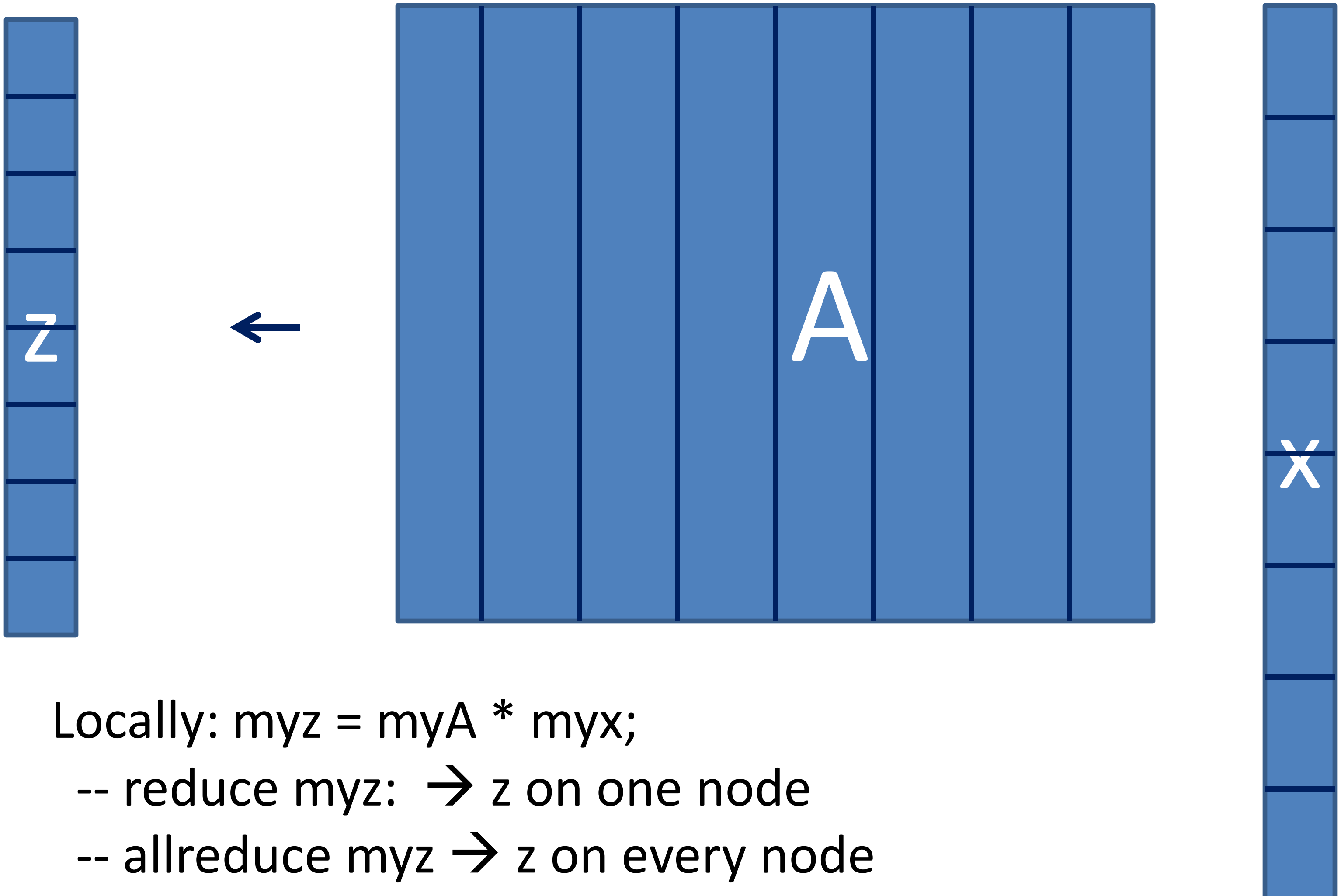
/* blocking. You may go immediately to Mars*/

MPI_Comm_free(&Mars);

Subsetting a Communicator

```
int me, sum_of_me, xranks[] = { 1 };
MPI_Group  gworld, gnot_one;
MPI_Comm comm_not_one;

MPI_Comm_group(MPI_COMM_WORLD, &gworld);
MPI_Group_excl(gworld, 1, xranks, &gnot_one);
MPI_Comm_create(MPI_COMM_WORLD, gnot_one, &comm_not_one);

MPI_comm_rank(MPI_COMM_WORLD, &me);
MPI_Allreduce(&me, &sum_of_me, 1, MPI_INT, MPI_SUM,
                                            MPI_COMM_WORLD);

MPI_comm_rank(comm_not_one, &me);
MPI_Allreduce(&me, &sum_of_me, 1, MPI_INT, MPI_SUM, comm_not_one);

MPI_Comm_free(&comm_not_one);
MPI_Group_free(&gnot_one);
MPI_Group_free(&gworld);
```

Matrix Vector Product via Reduce_scatter



Locally: myz = myA * myx;

-- reduce myz: → z on one node

-- allreduce myz → z on every node

-- reduce_scatter → effect of reduce, scatter,
z distributed across the nodes

Scaling, weak and strong

-- Let p be the number of nodes used for a job.

-- We can run on the whole machine: p may grow to a million or more.

-- What happens to efficiency when p grows?

$$\text{efficiency}(p) = \text{speedup}(p) / p$$

Strong scaling: fix the problem size (n) as p grows.

Weak scaling: let the size of the problem, n, grow with p. $n = n(p)$.

Limits to scaling

strong: efficiency(p) $\rightarrow$ 0 as p $\rightarrow \infty$

(eventually there are more processors than operations to do or data to store)

weak: we hope efficiency > E > 0 as p and n(p) $\rightarrow \infty$

What about n(p)?

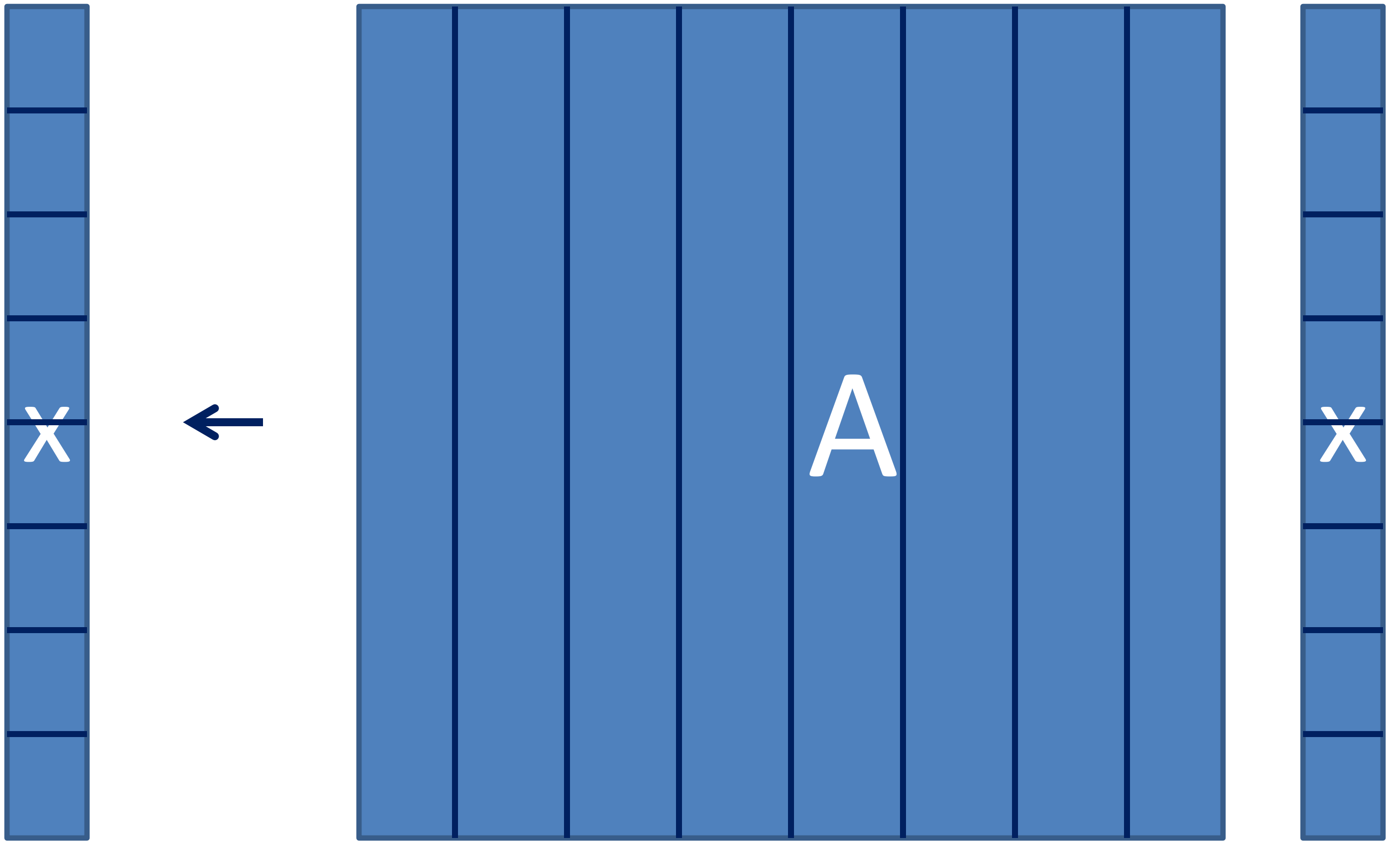
-- processors come with fixed amount of memory per process. Choose n so that the memory required is O(p).

Consider an iterative solver, like Larry Page's web page ranking idea.

while (some criterion)   x := A x;

A is a large, sparse matrix.

Power method



No sequential part. Load balanced.
Reduce_scatter does the communication.

Weak Scaling

T_{comp} = constant

dense matrix: n^2 = O(p) for constant memory per node

sparse matrix: n = O(p)

compute time = elements per node = constant

T_{comm} = O(n)

(= O(sqrt(p)) (dense), or O(p) (sparse)). In the reduce_scatter.

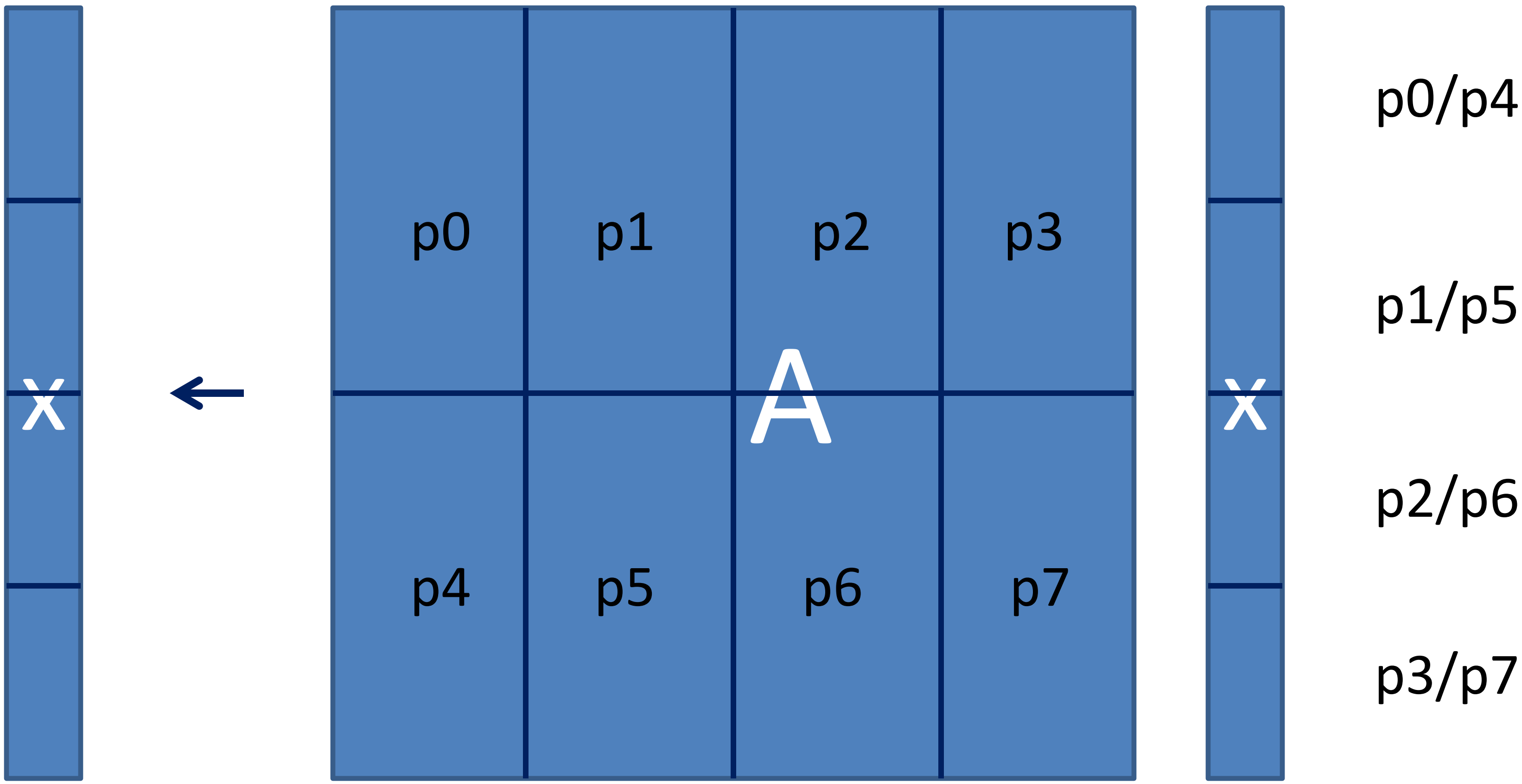
Proof: The local product has O(n) words; almost all of them -- (n (p-1)/p) -- must be injected into the network. Bandwidth is constant per node.

Is there any hope for this?

Think outside the box.

Matrices are two dimensional.

Distribute blocks of the matrix. View the computer as a 2D array of processors.



This is far trickier. Is it faster?

2D Distribution, Weak Scaling

Recall: $n^2 = O(p)$ (dense) or $n = O(p)$ (sparse)

T_{comp} = constant

T_{comm} = ???

--- n / p vector elements per processor. constant or declining

--- gather: n/sqrt(p) elements before local multiply

(dense: constant, sparse: sqrt(p))

--- reduce_scatter: same costs.

Bottom line:

T_{comm} is O(n) for 1D

and O(n / sqrt(p)) for 2D matrix mapping.