CME 194 Introduction to MPI

Coetibus Opus

http://cme194.stanford.edu

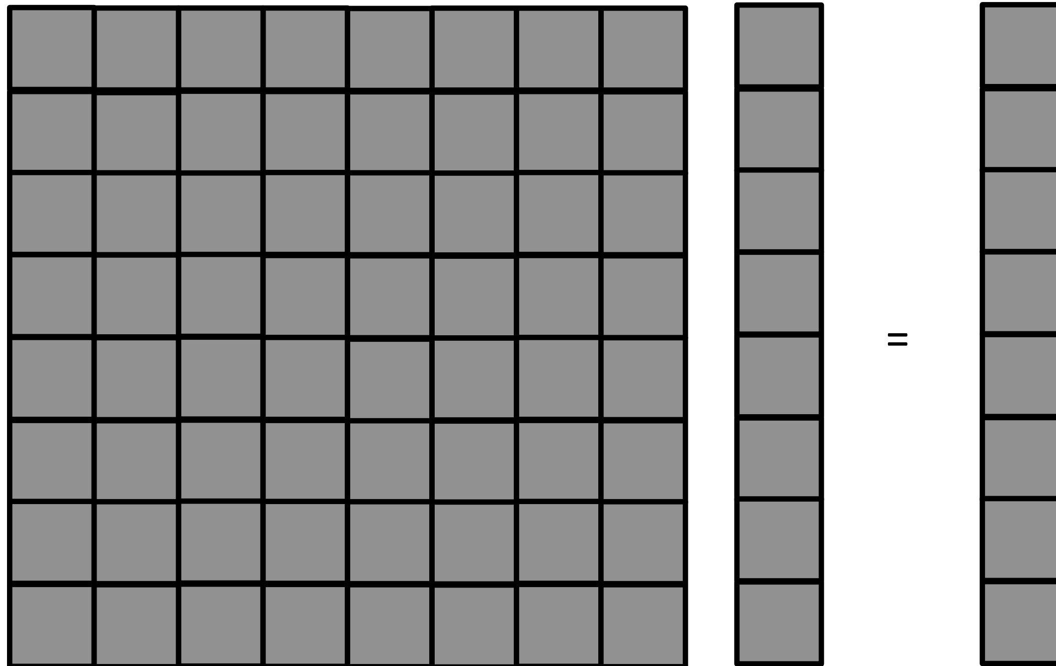
Recap

- **Last class:** Collective Operations
  - communication protocols (algorithms) such as reduce, broadcast, etc..
  - Naïve implementations of them with send/recv are slow.
  - Limitation: Needed all processes to work together
- **This class:** Communicators & Derived Datatypes
  - Remove this limitation
  - See how to communicate new datatypes.

Matrix-Vector Multiplication

Input: Matrix A and vector x

Output: **y** = Ax

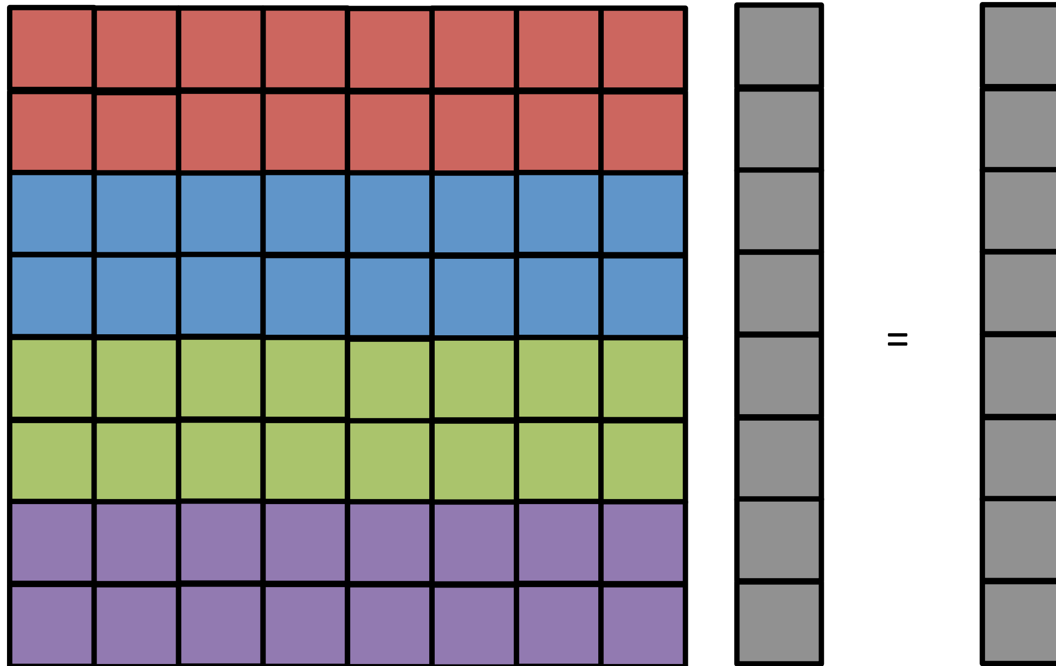


Matrix-Vector Multiplication

Input: Matrix A and vector x

Output: **y** = Ax

Suppose: Matrix partitioned into **blocks of rows** and all processes have **x**



Then: Use **MPI_Allgather** to combine the result

Matrix-Vector Multiplication

Input: Matrix A and vector x

Output: **y** = Ax

Suppose: Matrix partitioned into **blocks of rows** and all processes have **x**

All of x is not needed on each machine

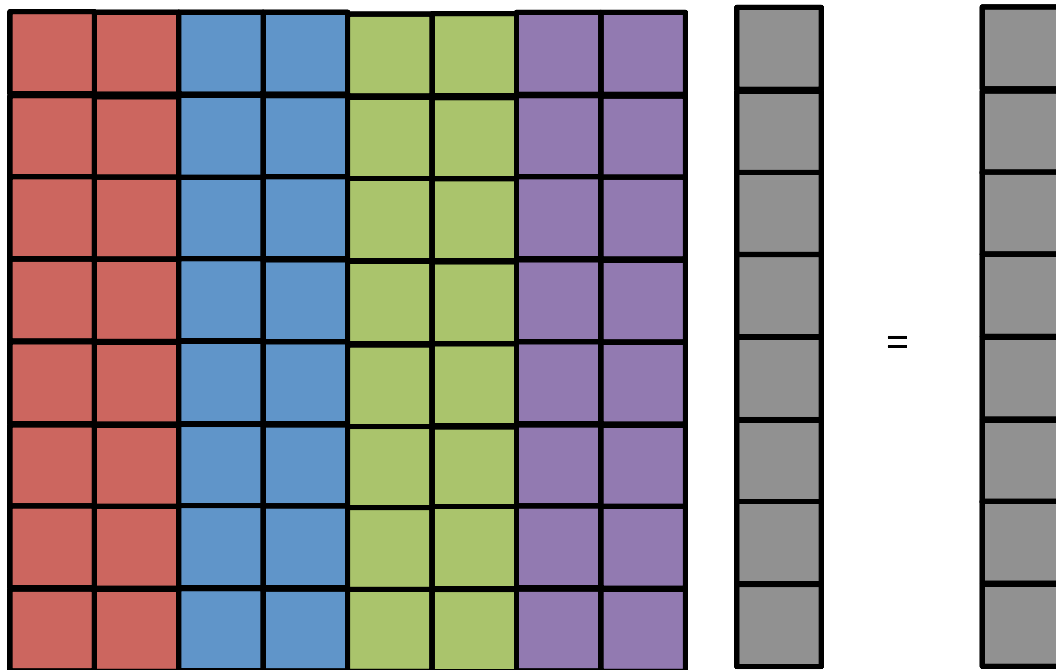
Then: Use **MPI_Allgather** to combine the result

Matrix-Vector Multiplication

Input: Matrix A and vector x

Output: **y** = Ax

Suppose: Matrix partitioned into **blocks of columns** and all processes have portion of **x**



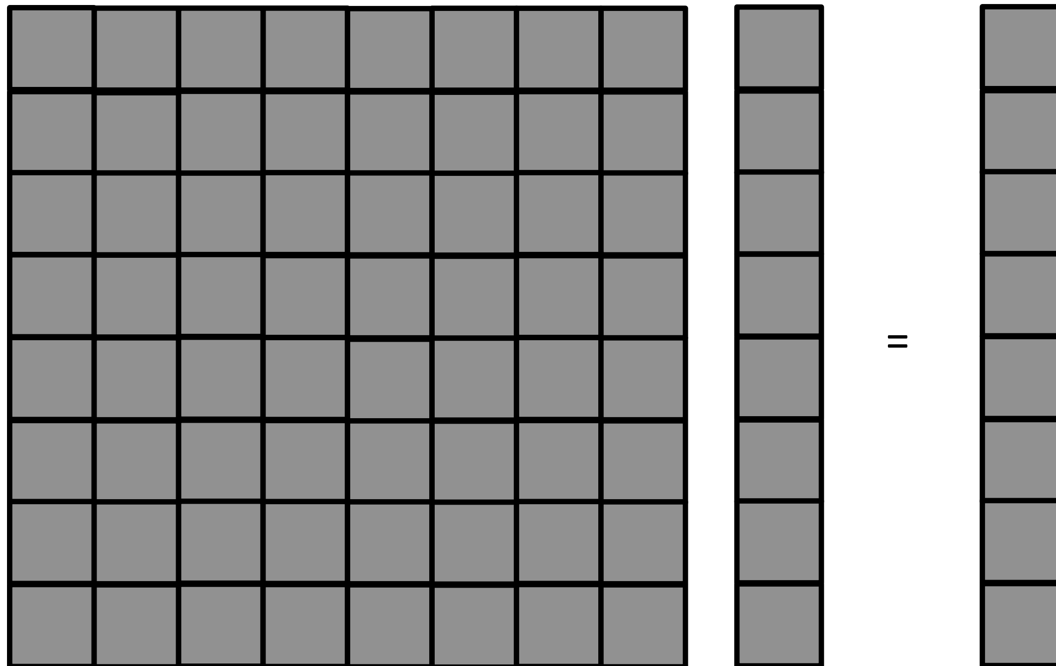
Then: Use ***n*** **reduces** to combine the results. Followed by **scatter**

Matrix-Vector Multiplication

Input: Matrix A and vector x

Output: **y** = Ax

Suppose: Matrix partitioned into **blocks**



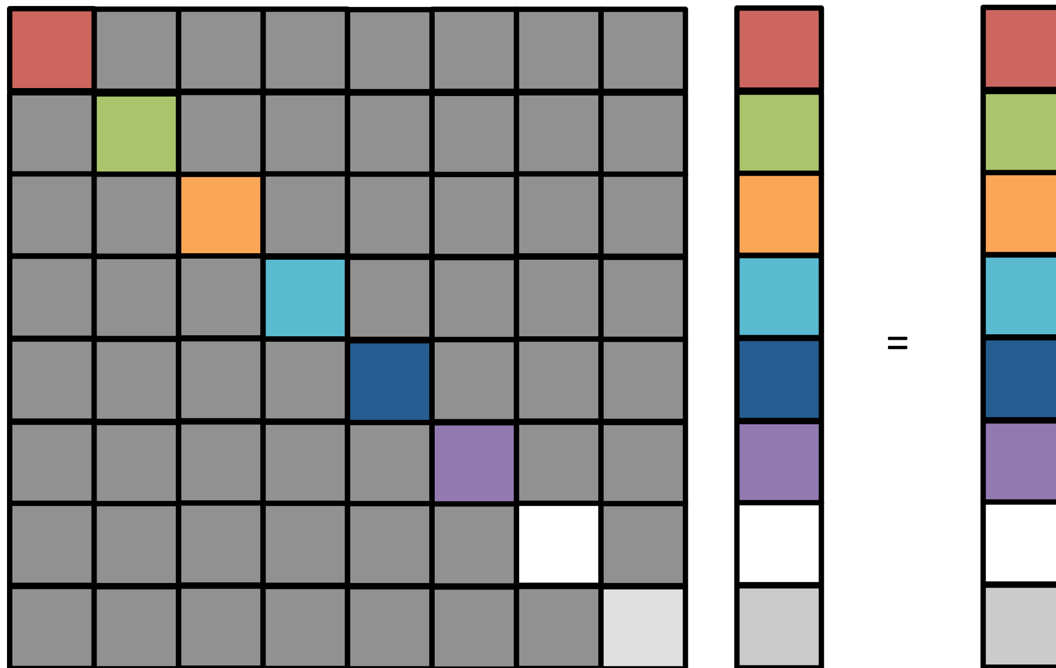
Each block is a process!

Matrix-Vector Multiplication

Input: Matrix A and vector x

Output: y = Ax

Suppose: Matrix partitioned into **blocks** and initially **diagonals** have **x**



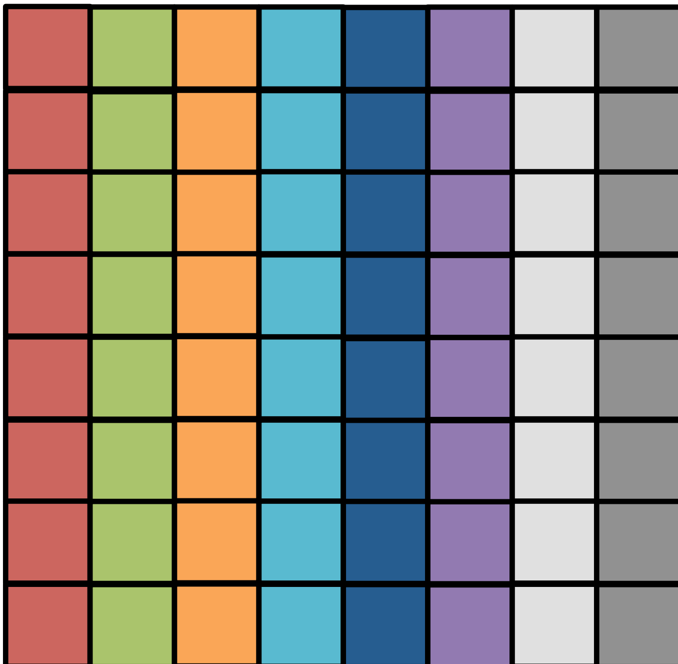
Each block is a process!

Matrix-Vector Multiplication

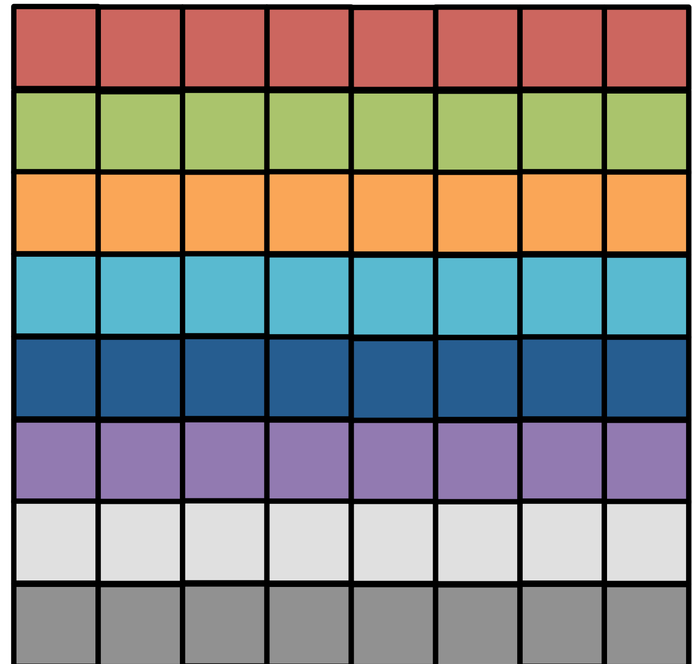
Input: Process **k** has a block of the matrix A

Output: **y** = Ax

Suppose: Each block of **x** is initially stored on at most one processor



Data in **x** must be **striped** like this



Result of Ax must be **reduced** like this

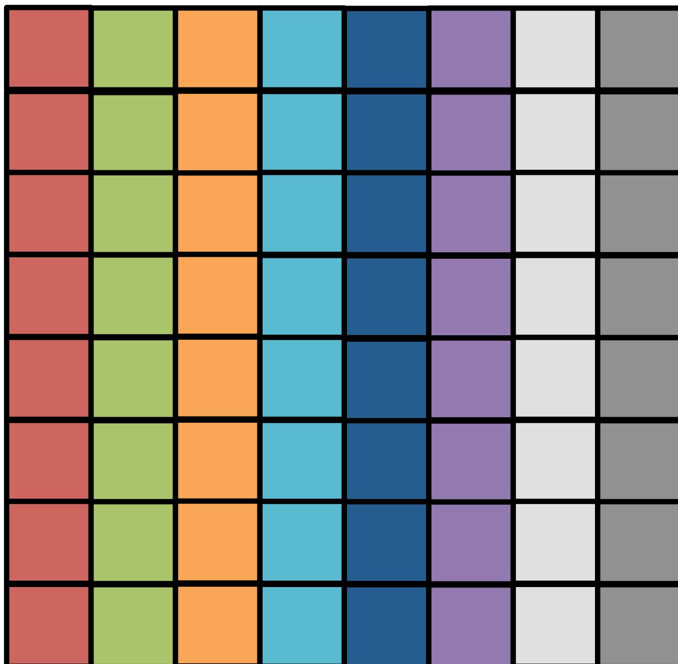
Matrix-Vector Multiplication

Input: Process **k** has a block of the matrix A

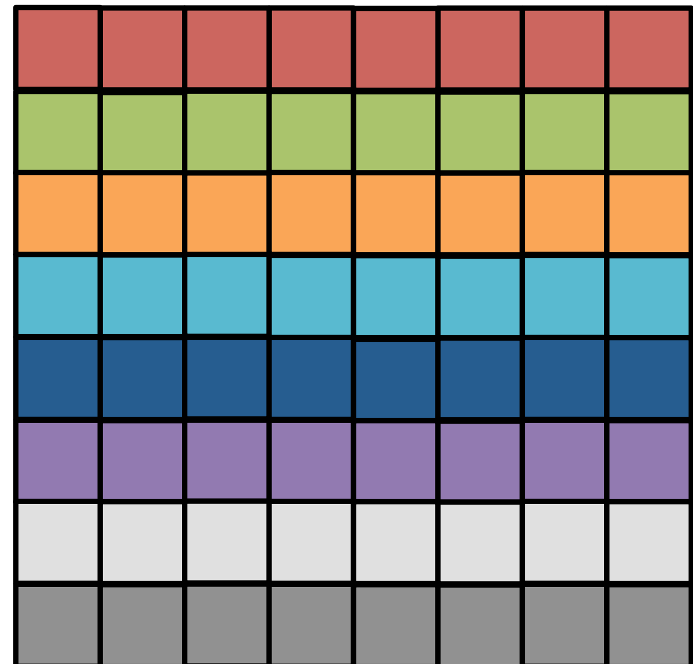
Output: **y** = Ax

Suppose: Each block of **x** is initially stored on at most one processor

Need: Communication **exclusively between rows** and **exclusively between columns**

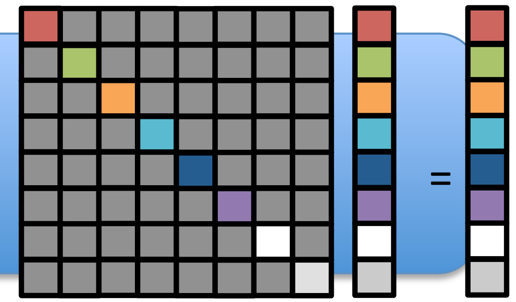


Data in **x** must be **striped** like this



Result of Ax must be **reduced** like this

Communicators



- Need a separate communicator for each
  - column of processors
  - row of processors

MPI_Comm_create(MPI_Comm oldcomm, MPI_Group group, MPI_Comm* newcomm)

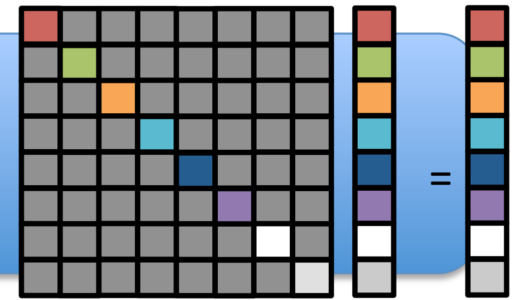
oldcomm original communicator

group subset of ranks from old communicator

newcomm the new communicator composed of the processes from **group**.

What is an MPI_Group?

Groups

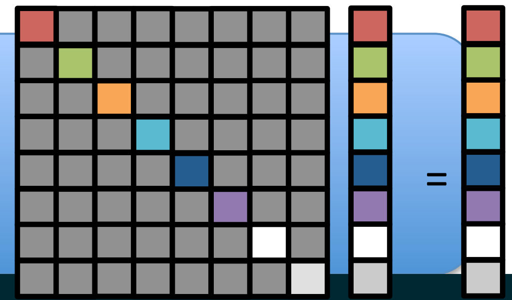


int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

Returns: process group corresponding to a communicator

- Groups are **sets** MPI defines the usual set arithmetic:
  - **MPI_Group_size**(MPI_group g, int* size) (size of this **group**)
  - **MPI_Group_rank**(MPI_group g, int* rank) (local processors rank in group)
  - **MPI_Group_union**(MPI_Group a, MPI_Group b, MPI_Group* out)
  - **MPI_Group_intersection**(MPI_Group a, MPI_Group b, MPI_Group* out)
  - **MPI_Group_difference**(MPI_Group a, MPI_Group b, MPI_Group* out)
- Also may **add and remove** non group ranks like this:
  - **MPI_Group_incl**(MPI_Group g, int n, const int ranks[], MPI_Group* out)
  - **MPI_Group_excl**(MPI_Group g, int n, const int ranks[], MPI_Group* out)

Communicators

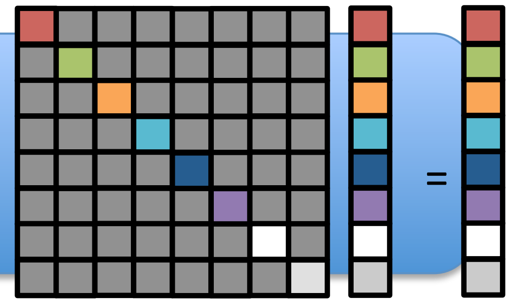



```
1  //assume p = q^2 processes;
2  int  q = sqrt( world_size);
3  if( rank < q){
4    MPI_Group group_world;
5    MPI_Group first_row_group;
6    MPI_Comm first_row_comm;
7    int* process_ranks;
8    process_ranks = (int*)malloc( q*sizeof(int));
9
10   for( auto proc = 0; proc < q; ++proc){ process_ranks[ proc] = proc; }
11
12   MPI_Comm_group( MPI_COMM_WORLD, &group_world);
13   MPI_Group_incl( group_world, q, process_ranks, &first_row_group);
14   MPI_Comm_create( MPI_COMM_WORLD, first_row_group, &first_row_comm);
15 }
```

Issues:

- MPI_Comm_create is a **blocking, collective** operation
- This is straightforward, but, not simple, or short
- Error prone code

Communicators



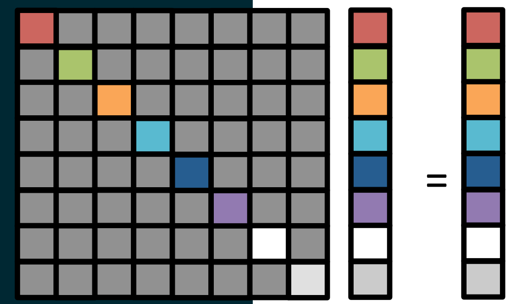
MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm* newcomm)

All processes in **comm** must execute this, but:

- Processes are **grouped** by their **color**
- Assigned a **new rank** in **0,…,group_size** based on their unique key (e.g. old ranks)

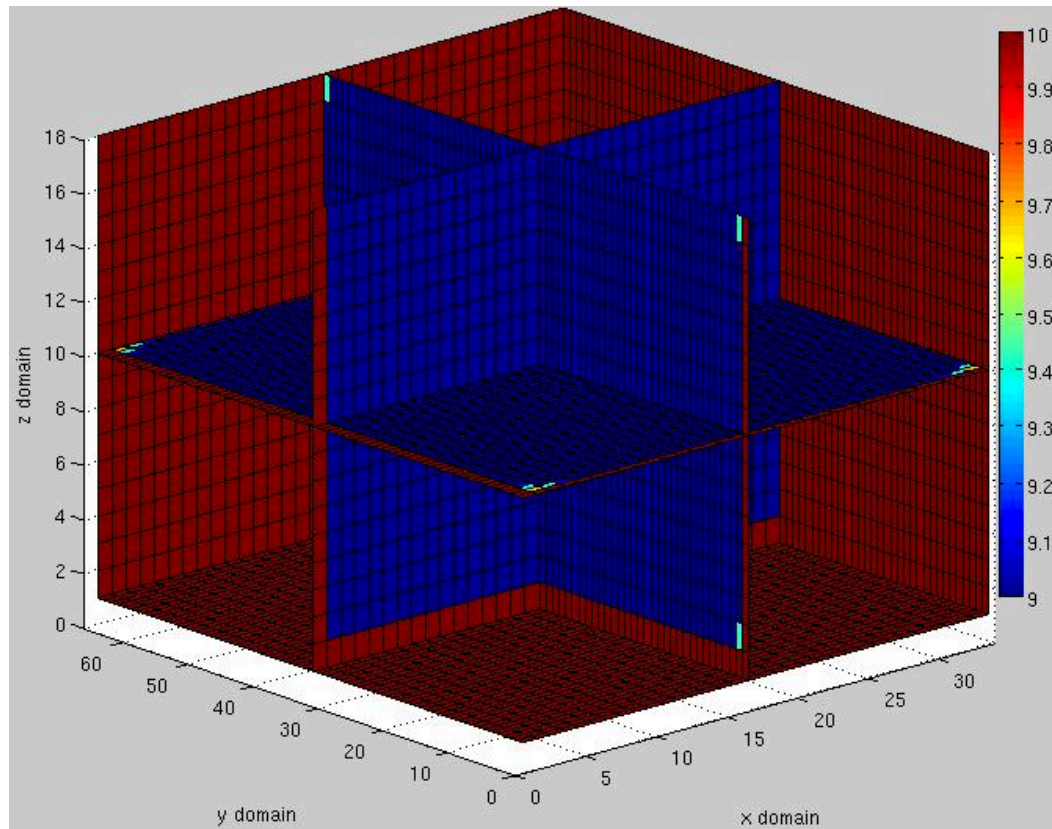
```
2 MPI_Comm row_comm;
3 MPI_Comm col_comm;
4 int  q = sqrt( world_size);
5 int  row_rank = world_rank % q;
6 int  col_rank = world_rank / q;
7 MPI_Comm_split( MPI_COMM_WORLD, col_rank, world_rank, &row_comm);
8 MPI_Comm_split( MPI_COMM_WORLD, row_rank, world_rank, &col_comm);
```

```
int main( int argc, char ** argv){
  MPI_Init( &argc, &argv);
  int world_rank, world_size;
  MPI_Comm_rank( MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size( MPI_COMM_WORLD, &world_size);
  double* A;
  library::get_local_matrix( A, world_rank, world_size);
  MPI_Comm row_comm;
  MPI_Comm col_comm;
  int  q = (int) sqrt( world_size);
  int  row_number = world_rank % q, col_number = world_rank / q;
  MPI_Comm_split( MPI_COMM_WORLD, col_number, world_rank, &row_comm);
  MPI_Comm_split( MPI_COMM_WORLD, row_number, world_rank, &col_comm);
  int row_rank = col_rank = 0;
  MPI_Comm_rank( row_comm, &row_rank);
  MPI_Comm_rank( col_comm, &col_rank);
  double* x;
  size_t x_len;
  if( on_the_diagonal){
        x_len = library::get_local_vector( x, row_rank);
        MPI_Bcast( &x, x_len, MPI_DOUBLE, col_rank, col_comm);
  }else{
        MPI_Bcast( &x, x_len, MPI_DOUBLE, row_number, col_comm);
  }
  double* local_y, y;
  size_t y_len = library::matvec( A, x, local_y);
  y = (double*) malloc( y_len*sizeof( double));
  MPI_Allreduce( local_y, y, y_len, MPI_DOUBLE, MPI_SUM, row_comm);
}
```



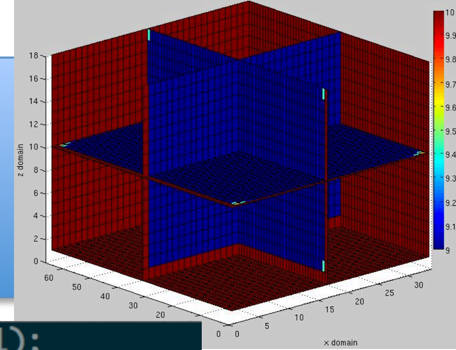
Differential Equations

PDES regularly require grid style communication



How do we deal with this style of communication?

Differential Equations



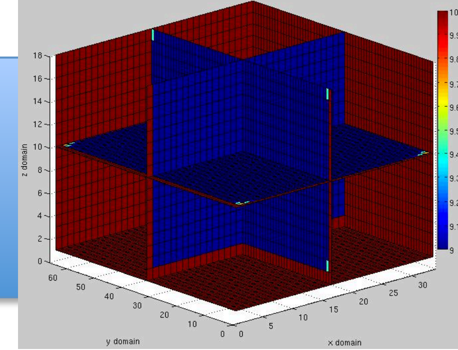
```
 1 // setup the grid size
 2 int p1 = 4;      // processes in h-direction
 3 int p2 = 4;      // processes in v-direction
 4 int *gridSize1;
 5 int *gridSize2;
 6 gridSize1 = malloc(p1*sizeof(int));
 7 gridSize2 = malloc(p2*sizeof(int));
 8 for (i=0; i<p1; i++) {
 9     gridSize1[i] = n1/p1;
10 }
11 gridSize1[p1-1] += fmod(n1,p1);
12 for (i=0; i<p2; i++) {
13     gridSize2[i] = n2/p2;
14 }
15 gridSize2[p2-1] += fmod(n2,p2);
16
17 // setup the grid location
18 int *gridLoc1;
19 int *gridLoc2;
20 gridLoc1 = malloc(p1*sizeof(int));
21 gridLoc2 = malloc(p2*sizeof(int));
22 gridLoc1[0] = 0;
23 for (i=1; i<p1; i++) {
24     gridLoc1[i] = gridLoc1[i-1]+gridSize1[i-1];
25 }
26 gridLoc2[0] = 0;
27 for (i=1; i<p2; i++) {
28     gridLoc2[i] += gridLoc2[i-1]+gridSize2[i-1];
29 }
```

Setting up a grid of processors

```
 1 int proc_h = fmod(proc_id,p1);
 2 int proc_v = proc_id/p1;
 3 int gs1 = gridSize1[proc_h];
 4 int gs2 = gridSize2[proc_v];
 5 int gl1 = gridLoc1[proc_h];
 6 int gl2 = gridLoc2[proc_v];
 7 int TOP = FALSE;
 8 int BOT = FALSE;
 9 int LEFT = FALSE;
10 int RIGHT = FALSE;
11 int POS_H = ODD;
12 int POS_V = ODD;
13
14 // Set location information of block
15 if ( proc_h == 0 ) LEFT = TRUE;
16 if ( proc_h == p1-1 ) RIGHT = TRUE;
17 if ( proc_v == 0 ) TOP = TRUE;
18 if ( proc_v == p2-1 ) BOT = TRUE;
19
20 // set even/odd status
21 if ( (proc_h % 2) == 0 ) POS_H = EVEN;
22 if ( (proc_v % 2) == 0 ) POS_V = EVEN;
```

Finding your process within the grid

Differential Equations



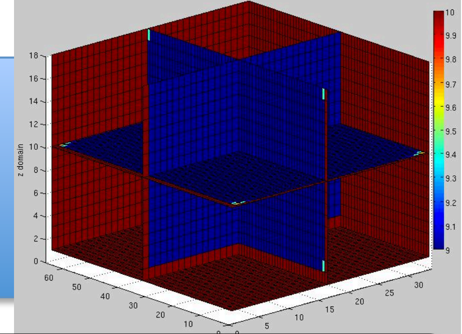
```
1  // iteration must begin here
2  for ( iter=0; iter<niter; iter++ ) {
3
4  // data transfer step 1: even horizontal
5  if ( POS_H == EVEN && !RIGHT ) {
6      MPI_Sendrecv(&A[gs1],1,columntype,proc_id+1,tag,
7                   &A[gs1+1],1,columntype,proc_id+1,tag,MPI_COMM_WORLD,&status);
8  }
9  else if ( POS_H == ODD ) {
10     MPI_Sendrecv(&A[1],1,columntype,proc_id-1,tag,
11                  A,1,columntype,proc_id-1,tag,MPI_COMM_WORLD,&status);
12 }
13
14 // data transfer step 2: odd horizontal
15 if ( POS_H == ODD && !RIGHT ) {
16     MPI_Sendrecv(&A[gs1],1,columntype,proc_id+1,tag,
17                  &A[gs1+1],1,columntype,proc_id+1,tag,MPI_COMM_WORLD,&status);
18 }
19 else if ( POS_H == EVEN && !LEFT) {
20     MPI_Sendrecv(&A[1],1,columntype,proc_id-1,tag,
21                  A,1,columntype,proc_id-1,tag,MPI_COMM_WORLD,&status);
22 }
23
24 // data transfer step 3: even vertical
25 if ( POS_V == EVEN && !BOT ) {
26     MPI_Sendrecv(&A[gs2*(gs1+2)],1,rowtype,proc_id+p1,tag,
27                  &A[(gs2+1)*(gs1+2)],1,rowtype,proc_id+p1,tag,MPI_COMM_WORLD,&status);
28 }
29 else if ( POS_V == ODD ) {
30     MPI_Sendrecv(&A[gs1+2],1,rowtype,proc_id-p1,tag,
31                  A,1,rowtype,proc_id-p1,tag,MPI_COMM_WORLD,&status);
32 }
33
34 // data transfer step 4: odd vertical
35 if ( POS_V == ODD && !BOT ) {
36     MPI_Sendrecv(&A[gs2*(gs1+2)],1,rowtype,proc_id+p1,tag,
37                  &A[(gs2+1)*(gs1+2)],1,rowtype,proc_id+p1,tag,MPI_COMM_WORLD,&status);
38 }
```

Many edge cases!

Alternative: MPI_Cart_create

Communication at each iteration

MPI_Cart_create



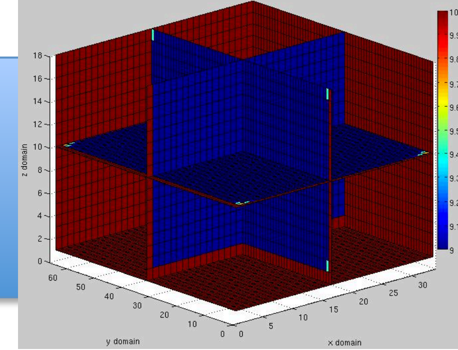
```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int dims[],
    int periods[], int reorder,
    MPI_Comm* comm_cart)
```

oldcomm the original communicator
ndims dimension of the grid
dims number of grid points in each dimension
periods specifies if the endpoints wrap around or not (e.g. true grid, torus, etc..)
reorder (bool) permute the input ranks based on the topology of the actual network
comm_cart the resulting communicator

Each processor may now be indexed by a vector in $\mathbb{Z}^n$

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims,
    int coords[])
```

MPI_Cart_rank



Given coordinates we may get the rank

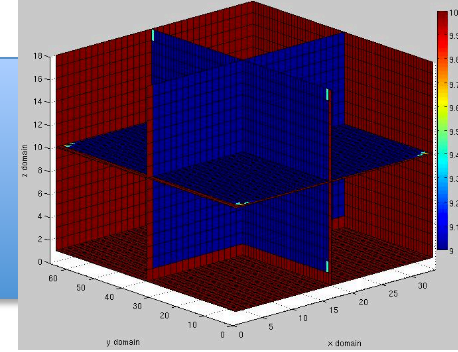
MPI_Cart_create(MPI_Comm comm, int coordinates[], int* rank)

Even better:

MPI_Cart_shift(MPI_Comm comm, int dir, int disp, int* source, int* dest)

- **dir** the index of the dimension to move along
- **disp** how far to move
- **source** the source rank
- **dest** the destination rank

HW2



- **Available soon!**
- Will help to use these commands!

Changing gears. Any questions?

MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

How about a struct?

What MPI must know

- There are three elements to be transmitted
  - The first is a **float**
  - The second is a **float**
  - The third is an **int**
- The first element has address **&a**
- The second element has address **&b**
- The third element has address **&n**

Addresses may be **replaced** with **displacements**

Derived Datatypes

A derived datatype is a **collection**

$$\{(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})\}$$

t_i is an MPI Type

d_i Is a displacement

```
 2 void build_derived_type(float* a_ptr, float* b_ptr, int* n_ptr,
 3                         MPI_Datatype*  mesg_mpi_t_ptr) {
 4
 5     /* The number of elements in each "block" of the   */
 6     /*      new type.  For us, 1 each.                  */
 7     int block_lengths[3];
 8
 9     /* Displacement of each element from start of new  */
10     /*      type.  The "d_i's."                         */
11     /* MPI_Aint ("address int") is an MPI defined C    */
12     /*      type.  Usually an int.                      */
13     MPI_Aint displacements[3];
14
15     /* MPI types of the elements.  The "t_i's."        */
16     MPI_Datatype typelist[3];
17
18     /* Use for calculating displacements               */
19     MPI_Aint start_address;
20     MPI_Aint address;
21
22     block_lengths[0] = block_lengths[1]
23                     = block_lengths[2] = 1;
24
25     /* Build a derived datatype consisting of  */
26     /* two floats and an int                   */
27     typelist[0] = MPI_FLOAT;
28     typelist[1] = MPI_FLOAT;
29     typelist[2] = MPI_INT;
```

```c
31      /* First element, a, is at displacement 0      */
32      displacements[0] = 0;
33
34      /* Calculate other displacements relative to a */
35      MPI_Address(a_ptr, &start_address);
36
37      /* Find address of b and displacement from a   */
38      MPI_Address(b_ptr, &address);
39      displacements[1] = address - start_address;
40
41      /* Find address of n and displacement from a   */
42      MPI_Address(n_ptr, &address);
43      displacements[2] = address - start_address;
44
45      /* Build the derived datatype */
46      MPI_Type_struct(3, block_lengths, displacements,
47          typelist, mesg_mpi_t_ptr);
48
49      /* Commit it -- tell system we'll be using it for */
50      /* communication.                                 */
51      MPI_Type_commit(mesg_mpi_t_ptr);
52  } /* Build_derived_type */
53
```

Details Derived Datatypes

int MPI_Type_struct (int count, int* blocklengths, MPI_Aint *displacements, MPI_Datatype* array_of_types, MPI_Datatype* newtype)

This is the most general way of building an MPI type.

int MPI_Type_vector()

- For equally spaced elements of arrays

int MPI_Type_contiguous()

- For contiguous entries of an array

int MPI_Type_indexed()

- For arbitrary entries in an array

MPI_Pack: An alternative

May explicitly **copy** noncontiguous data into a contiguous block of memory

```
 1  /* Now pack the data into buffer.  Position = 0 */
 2  /* says start at beginning of buffer.             */
 3  position = 0;
 4
 5  /* Position is in/out */
 6  MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100,
 7      &position, MPI_COMM_WORLD);
 8  /* Position has been incremented: it now refer- */
 9  /* ences the first free location in buffer.     */
10
11  MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100,
12      &position, MPI_COMM_WORLD);
13  /* Position has been incremented again. */
14
15  MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100,
16      &position, MPI_COMM_WORLD);
17  /* Position has been incremented again. */
18
19  /* Now broadcast contents of buffer */
20  MPI_Bcast(buffer, 100, MPI_PACKED, 0,
21      MPI_COMM_WORLD);
```

MPI_Pack: An alternative

May explicitly **copy** noncontiguous data into a contiguous block of memory

```
/* Now pack the data into buffer.  Position = 0 */
/*
pos

/*
MPI

/* Position has been incremented: It now refer- */
/* ences the first free location in buffer.      */

MPI_

/* P

MPI_
    &position, MPI_COMM_WORLD);
/* Position has been incremented again. */

/* Now broadcast contents of buffer */
MPI_Bcast(buffer, 100, MPI_PACKED, 0,
    MPI_COMM_WORLD);
```

```
MPI_Pack(void* in, int incount, MPI_Datatype datatype,
         void* out, int outsize, int* position,
         MPI_Comm comm)
```

```
MPI_Unpack(void* in, int incount, int* position,
           void* out, int outsize, MPI_Datatype datatype,
           MPI_Comm comm)
```

Which do I use?

- Usually better to use derived datatypes.
- Do experiments.