Electrical Engineering and Computer Science

EECS 358 - INTRODUCTION TO PARALLEL COMPUTING

Lecture 12

Dist. Mem. Message Passing Programming - III

Outline

- Intermediate MPI concepts
- Solution of Poisson Problem using Jacobi Method
- Use of Topologies
- Use of Decompositions
- Use of Data Exchange Routines
- Use of Derived Datatypes
- READING: Foster, Chapter 8

The Poisson Problem

- The Poisson Problem is a simple partial differential equation (PDE)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{1}$$

  given u(x,y) = g(x,y) on the boundary.

- We may "solve" the Poisson equation numerically over a region by discretizing it in the x and y directions to obtain a grid of points and then computing the approximate solution values at these points.

- Assuming that the distances between neighboring points in the grid in the x and y directions is h

The Poisson Problem

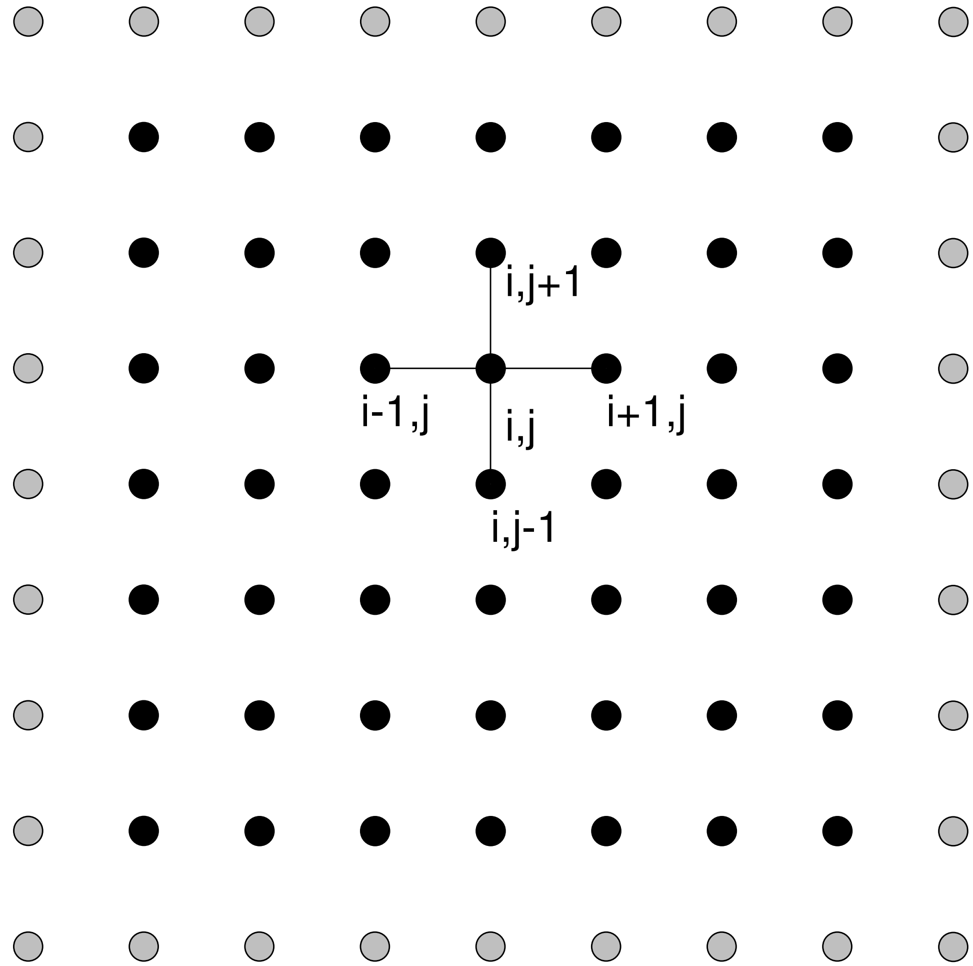
- We may replace the partial derivatives by numerical approximations involving first-order finite differences to get

$$u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)-4u(i,j)=f(i,j) \qquad (2)$$

- We can then write a Jacobi iteration as

$$u^{k+1}(i,j)=1/4(u^k(i-1,j)+u^k(i,j+1)+u^k(i,j-1)+u^k(i+1,j)-h^2 f(i,j)) \quad (3)$$

Poisson Problem



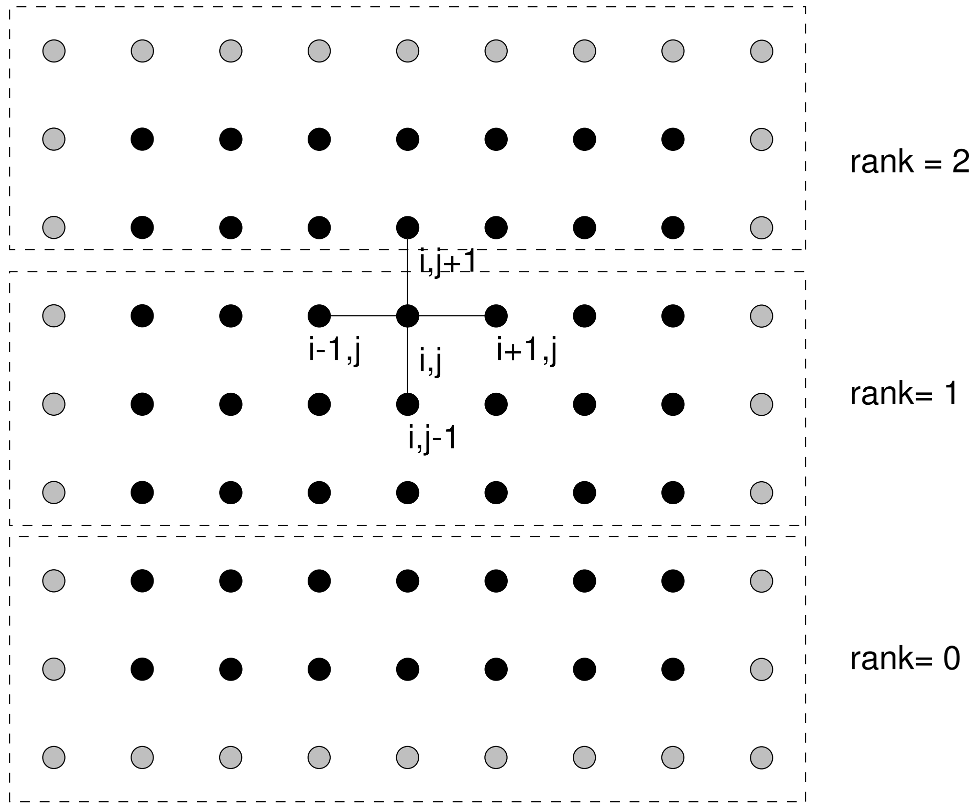
Finite Difference Algorithm

```
int finite_difference()
{
int i, j, n;
double u[n,n], unew[n,n];
for (k = 0; k <ITERS; k++) {
  for (i = 0; i < n; j++) {
    for (j = 0; j < n; i++) {
      unew[i,j] = 0.25 * (u[i-1,j] + u[i+1,j]
                  + u[i,j-1] + u[i, j+1]) - f[i,j];
    }
  }
  diffmax = 0.0;
  for (i = 1; i < n; i++) {
    for (j = 1; j < n; j++) {
      diff = abs(unew[i,j] - u[i,j]);
      if (diff > diffmax) diffmax = diff;
      u[i,j] = unew[i,j];
    }
   }
}
```

Basic Sweep Routine

```
int sweep()
{
int i, j, n;
double u[n,n], unew[n,n];
  for (i = 0; j < n; j++) {
    for (j = 0; i < n; i++)
      unew[i,j] = 0.25 * (u[i-1,j] + u[i+1,j]
                  + u[i,j-1] + u[i, j+1]) - f[i,j];
}
```

One-dimensional Decomposition of Domain



Jacobi Iteration for Slice of Domain

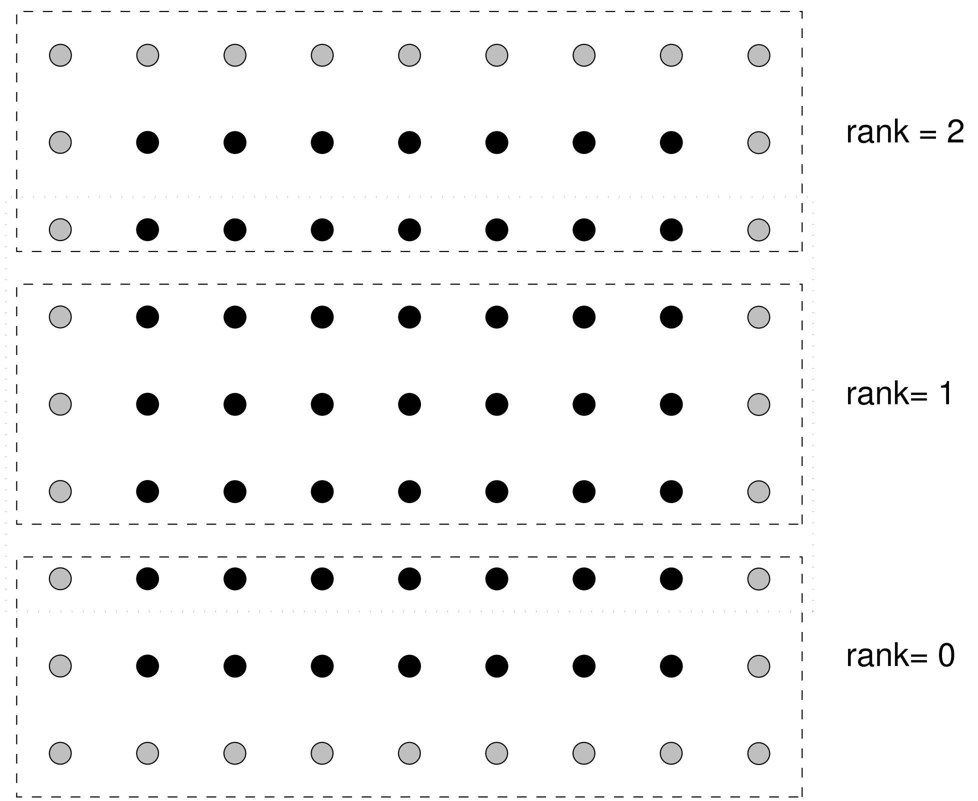
```
int i, j, n, m;
  /* assume P processors, partition array by rows */
double u[n/P,n], unew[n/P,n];

for (i = 0; i < n/P; i++)
  for (j = 0; j < n; j++) {
      unew[i,j] = 0.25 * (u[i-1,j] + u[i+1,j]
                 + u[i,j-1] + u[i, j+1]) - f[i,j];
}
```

Problem with Previous Code

- The loop will require elements such as u[i-1,j] or u[i+1,j] from a different process
- We will discuss how to get this data from other processes
- Define an overlap region to hold data from other regions

```
double u[n/P+2, n]
```

Computational Domain with Overlap Regions



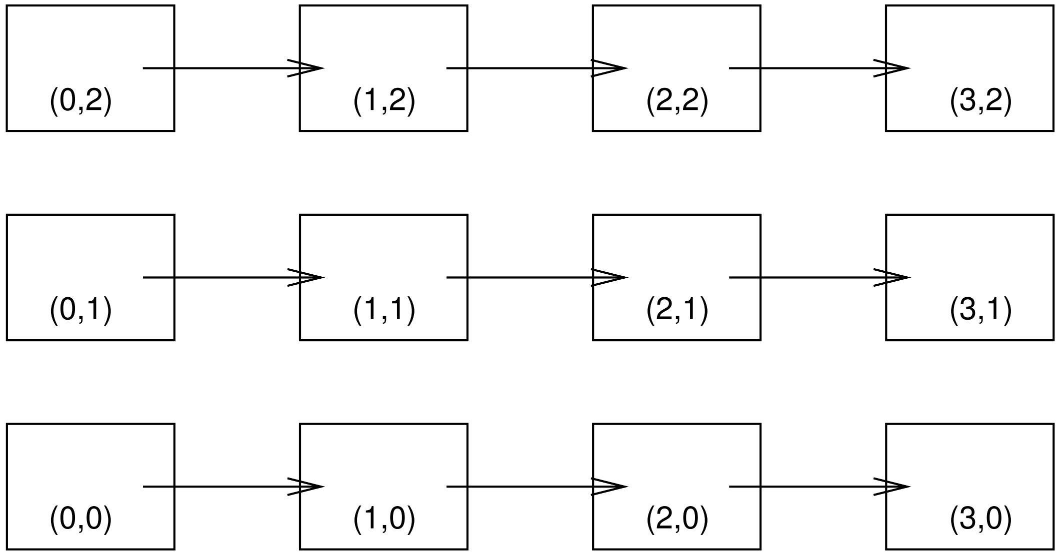
Defining Topologies

- Our next task is to define how to assign processes to each part of the decomposed domain
- MPI lets user specify various *application topologies*
- The routine MPI_Cart_create() creates a Cartesian decomposition of the processes, with the number of dimensions given by the $ndim$ argument

```
dims[0] = 4
dims[1] = 3
periods[0] = 0
periods[1] = 0
reorder = 1
ndim = 2
MPI_Cart_create(MPI_COMM_WORLD, ndim, *dims,
                     *periods, reorder, comm2d)
```

- This creates a new communicator "comm2d"
- The "periods" argument specifies if connection is with wrapround, i.e. if the processes at end are connected together.

Illustration of Cartesian Topology



Finding Neighbors of a Topology

- To determine the coordinates of a calling process

```
MPI_Cart_get(comm1d, 2, *dims, *periods, *coords);
printf('(', coords[0], ',' coords[1], ')');
```

- To determine rank

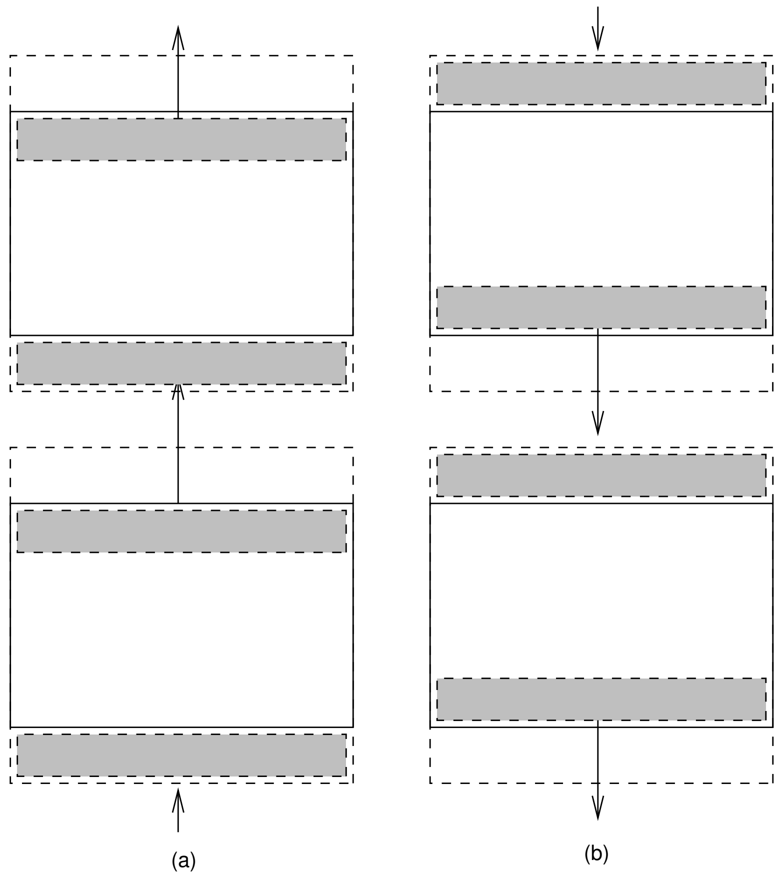
```
MPI_Cart_rank(comm1d, *coord, *myrank);
printf(myrank);
```

Using Topologies

- Performing shift operations

```
MPI_Cart_shift(comm1d, direction, shift, *src, *dest);
```

Data Exchange Operations



Data Exchange Routine

```
int exchng1(a, nx, comm1d, nbrbottom, nbrtop)
{
int nx; double a(n/P+2,n)
int comm1d, nbrbottom, nbrtop
int status(MPI_STATUS_SIZE), ierr

if (nbrtop != MPI_UNDEFINED) {
MPI_Send(&a(1,0), nx, MPI_DOUBLE,
             nbrtop, 0, comm1d);
MPI_Recv(&a(0,0), nx, MPI_DOUBLE,
             nbrtop, 0, comm1d);
}

if (nbrbottom != NPI_UNDEFINED) {
MPI_Send(&a(n/P,0), nx, MPI_DOUBLE,
             nbrbottom, 1, comm1d);
MPI_Recv(&a(n/P+1,0), nx, MPI_DOUBLE,
             nbrbottom, 1, comm1d);
}
}
```

Basic Jacobi Sweep Routine

```
int sweep1d( &a, &f, nx, &b)
int nx
double a(n/P+2,n), f(n/P+2,n), b(n/P+2,n)

int i, j
double h

h = 1.0/(nx+1);

for (i = 1; i < (n/P+1); i++)
   for (j = 0; j < n; j++)
      b[i,j] = 0.25 * (a[i-1,j] + a[i+1,j]
           + a[i,j-1] + a[i, j+1]) - h * h * f[i,j])
```

Overall Jacobi Implementation

```
main()
      #define maxn 128
      double a(maxn,maxn), b(maxn,maxn), f(maxn,maxn);
      int nx, ny;
      int myid, numprocs;
      int comm1d, nbrbottom, nbrtop, it;

      MPI_Init();
      MPI_Comm_rank( MPI_COMM_WORLD, myid);
      MPI_Comm_size( MPI_COMM_WORLD, numprocs);
      if (myid == 0) nx = 100;
      MPI_Bcast(nx,1,MPI_INT,0,MPI_COMM_WORLD);
      ny   = nx;
```

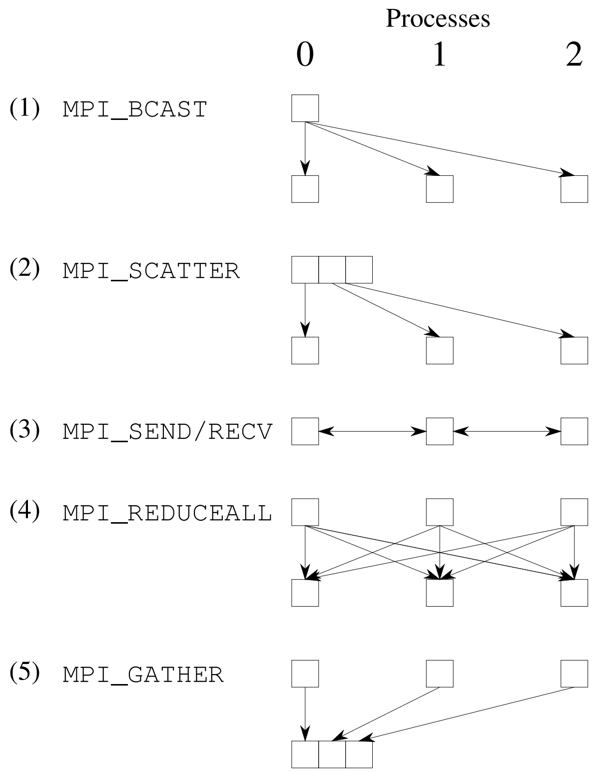
Jacobi Implementation-contd

```
/* Create a one dim cartesian mapping
      MPI_Cart_Create( MPI_COMM_WORLD, 1, numprocs, 0,1,comm1d);
/* Get my position in this communicator, and my neighbors
      MPI_Comm_Rank(comm1d, myid);
      MPI_Cart_shift( comm1d, 0,1, &nbrbottom, &nbrtop);
/* Initialize the right-hand-side (f) and the initial solution guess (a)
      onedinit( a, b, f, nx);
```

Jacobi Implementation-contd

```
      MPI_Barrier( MPI_COMM_WORLD);
      for (it=1; it < 100; it++) {
        exchng1( a, nx, comm1d, nbrbottom, nbrtop );
        sweep1d( a, f, nx, b );
        dwork = diff( a, b, nx, s, e );
        MPI_Allreduce( dwork, diffnorm, 1,MPI_DOUBLE,MPI_MAX,comm1d);
        if (diffnorm < 1.0e-5) break;
        exchng1( b, nx, comm1d, nbrbottom, nbrtop );
        sweep1d( b, f, nx, a );
        dwork = diff( a, b, nx, s, e );
        MPI_Allreduce( dwork, diffnorm, 1,MPI_DOUBLE,MPI_MAX,comm1d);
        if (diffnorm < 1.0e-5) break;
}
      MPI_Finalize();
```

Overall Communication Pattern of Jacobi



Use of MPI DataTypes

- One of MPI's novel features is the use of a datatype associated with every message.
- Specify the length of a message as a given count of occurrences of a given datatype.
- MPI has basic datatypes such as MPI_INT, MPI_FLOAT, etc.
- In MPI a datatype is an object that specifies a sequence of basic datatypes and displacements in bytes for each datatype.

MPI Datatypes in C

MPI Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Typemaps in MPI

- We represent a datatype as a sequence of pairs of basic types and displacements
- Call this sequence a *typemap*
- Typemap = $(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})$
- Example: (int,0), (char, 4) is a typemap describing a datatype which is an *integer* of 4 bytes, and then a *character* of 1 byte.
- Strictly, the total size is 5 bytes, but if we assume 4-byte alignment, then using padding, the extent of the previous datatype is 8 bytes.
- To add another *integer*, we have to specify (int,0), (char, 4), (int,8)

Derived Datatypes

- The typemap is a general way of describing an arbitrary datatype.
- However, need a convenient way to represent datatypes with large number of entries.
- CONTIGUOUS: This produces a new datatype by making *count* copies of an existing one.
- VECTOR: This creates a new datatype but allows regular gaps in the displacements
- INDEXED: In this datatype, an array of displacements of the input datatype is provided.

Example of CONTIGUOUS Datatype

- Assume an original datatype *oldtype* has typemap (int, 0), (double, 4), then

  ```
  MPI_Type_contiguous(2, oldtype, & newtype);
  ```

- Creates a datatype *newtype* with a typemap:

  (int, 0), (double, 4),(int, 12), (double, 16) with an extent of 24 bytes.

- To actually send such a data use the sequence of calls:

  ```
  MPI_Type_contiguous(count, datatype, &newtype);
  MPI_Type_commit(&newtype);
  MPI_Send(buffer, 1, newtype, dest, tag, comm);
  MPI_Type_free (&newtype);
  ```

Example of VECTOR Datatype

```
MPI_Type_Vector(count, blocklength, stride,
         oldtype, stridetype);
MPI_Type_Commit(stridetype);
MPI_Send();
```

- Specifically, for the Jacobi 2-D decomposition

```
MPI_Type_Vector(ey+1 - (sy-1), 1, ex+1 - (sx-1),
         MPI_DOUBLE, stridetype)
MPI_Type_Commit(stridetype);
MPI_SEND();
```

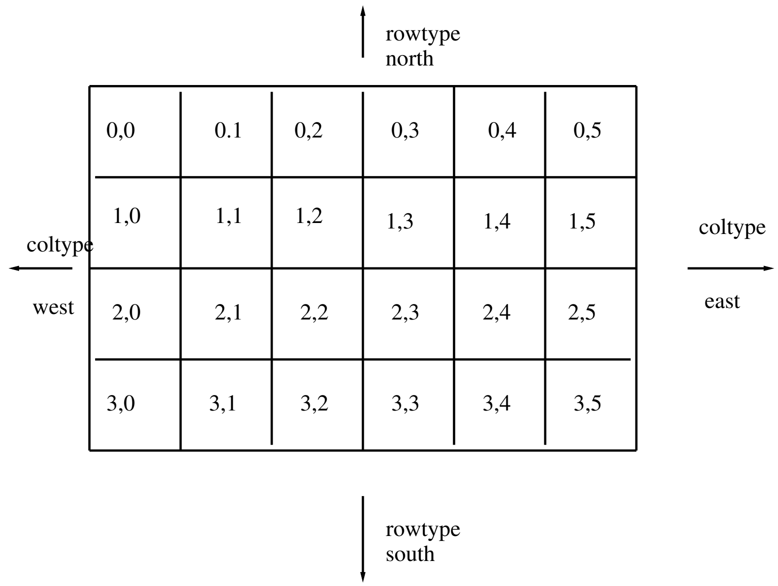
Example of VECTOR Datatype

29	30	31	32	33	34	35
22	23	24	25	26	27	28
15	16	17	18	19	20	21
8	9	10	11	12	13	14
1	2	3	4	5	6	7

MPI_Type_Vector(5, 1, 7, MPI_DOUBLE, newtype);

More Examples of Data Types

- Consider program to communicate the north and south rows and east and west columns of a 4 X 6 C array for a finite difference code



Finite Difference Code Example

```
integer coltype, rowtype, comm, ierr;
/*  The derived type rowtype is 6 contiguous reals.   */
MPI_Type_Contiguous(6, MPI_FLOAT, rowtype);
MPI_Type_Commit(rowtype) ;
/* The derived type coltype is 4 reals, located 6 apart. */
MPI_Type_Vector(4, 1, 6, MPI_FLOAT, coltype, ierr);
MPI_Type_Commit (coltype, ierr);
...
MPI_Send(&array[0,0], 1, coltype, west, 0, comm);
MPI_Send(&array[0,5], 1, coltype, east, 0, comm);
MPI_Send(&array[0,0], 1, rowtype, north, 0, comm);
MPI_Send(&array[3,0], 1, rowtype, south, 0, comm);
...
MPI_Type_Free(rowtype, ierr)
MPI_Type_Free(coltype, ierr)
```

Summary

- Intermediate MPI concepts
- Solution of Poisson Problem using Jacobi Method
- Use of Topologies
- Use of Decompositions
- Use of Data Exchange Routines
- Use of Derived Datatypes