Lecture 9: Numerical Partial Differential Equations(Part 1)

Finite Difference Method to Solve 2D Diffusion Equation

Consider to solve $\begin{cases} \dfrac{\partial u}{\partial t} = u_{xx} + u_{yy} + f(x,y) & in\ \Omega \\ u = 0 & on\ \ \partial\Omega \end{cases}$

by using an forward in time and backward in space (FTCS or explicit) finite difference scheme.

Here $\Omega = [0,a] \times [0,b], f(x,y) = xy$. a and b are constants and > 0.

Finite Differences

- Spatial Discretization: $0 = x_0 < \cdots < x_M = a$ with $x_i = \frac{i}{M} a$ and $0 = y_0 < \cdots < y_N = b$ with $y_j = \frac{j}{N} b$. Define $\Delta x = \frac{a}{M}$ and $\Delta y = \frac{b}{N}$.
- Differential quotient:

$$u_{xx}(x_i, y_j, t) \sim \frac{u(x_{i-1}, y_j, t) - 2u(x_i, y_j, t) + u(x_{i+1}, y_j, t)}{\Delta x^2}$$

$$u_{yy}(x_i, y_j, t) \sim \frac{u(x_i, y_{j-1}, t) - 2u(x_i, y_j, t) + u(x_i, y_{j+1}, t)}{\Delta y^2}$$

$$u_t(x_i, y_j, t_n) \sim \frac{u(x_i, y_j, t_{n+1}) - u(x_i, y_j, t_n)}{\Delta t}$$

Insert quotients into PDE yields:

$$\begin{aligned} v(x_i, y_j, t_{n+1}) &= v(x_i, y_j, t_n) \\ &+ \Delta t \left(\frac{v(x_{i-1}, y_j, t_n) - 2v(x_i, y_j, t_n) + v(x_{i+1}, y_j, t_n)}{\Delta x^2} \right. \\ &\left. + \frac{v(x_i, y_{j-1}, t_n) - 2v(x_i, y_j, t_n) + v(x_i, y_{j+1}, t_n)}{\Delta y^2} \right) + \Delta t f(x_i, y_j) \end{aligned}$$

Or in short notation

$$\begin{aligned} v_{i,j}^{n+1} = v_{i,j}^n + \Delta t \left(\frac{v_{i-1,j}^n - 2v_{i,j}^n + v_{i+1,j}^n}{\Delta x^2} + \frac{v_{i,j-1}^n - 2v_{i,j}^n + v_{i,j+1}^n}{\Delta y^2} \right) \\ + \Delta t f(x_i, y_j) \end{aligned}$$

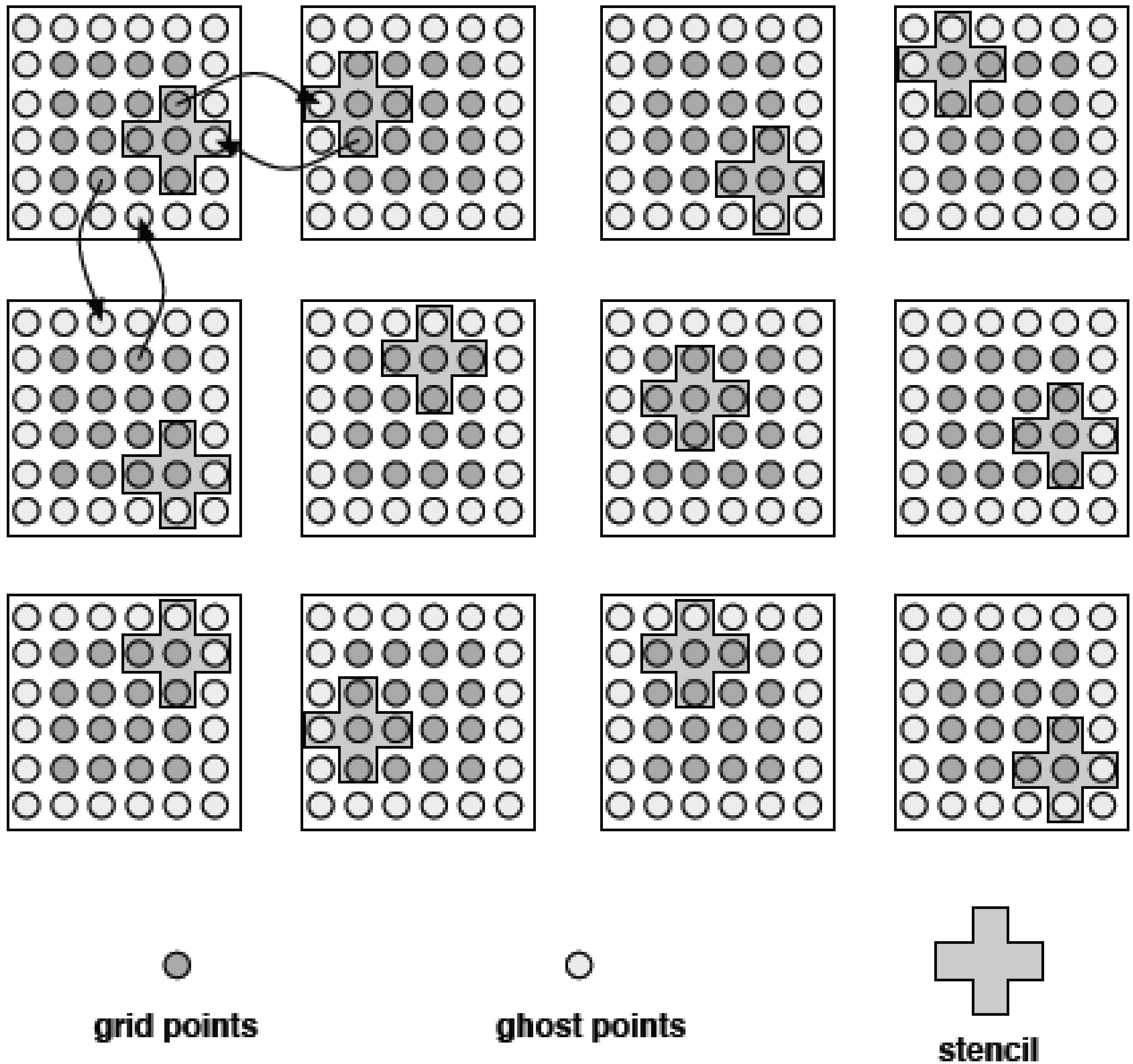
Boundary conditions:

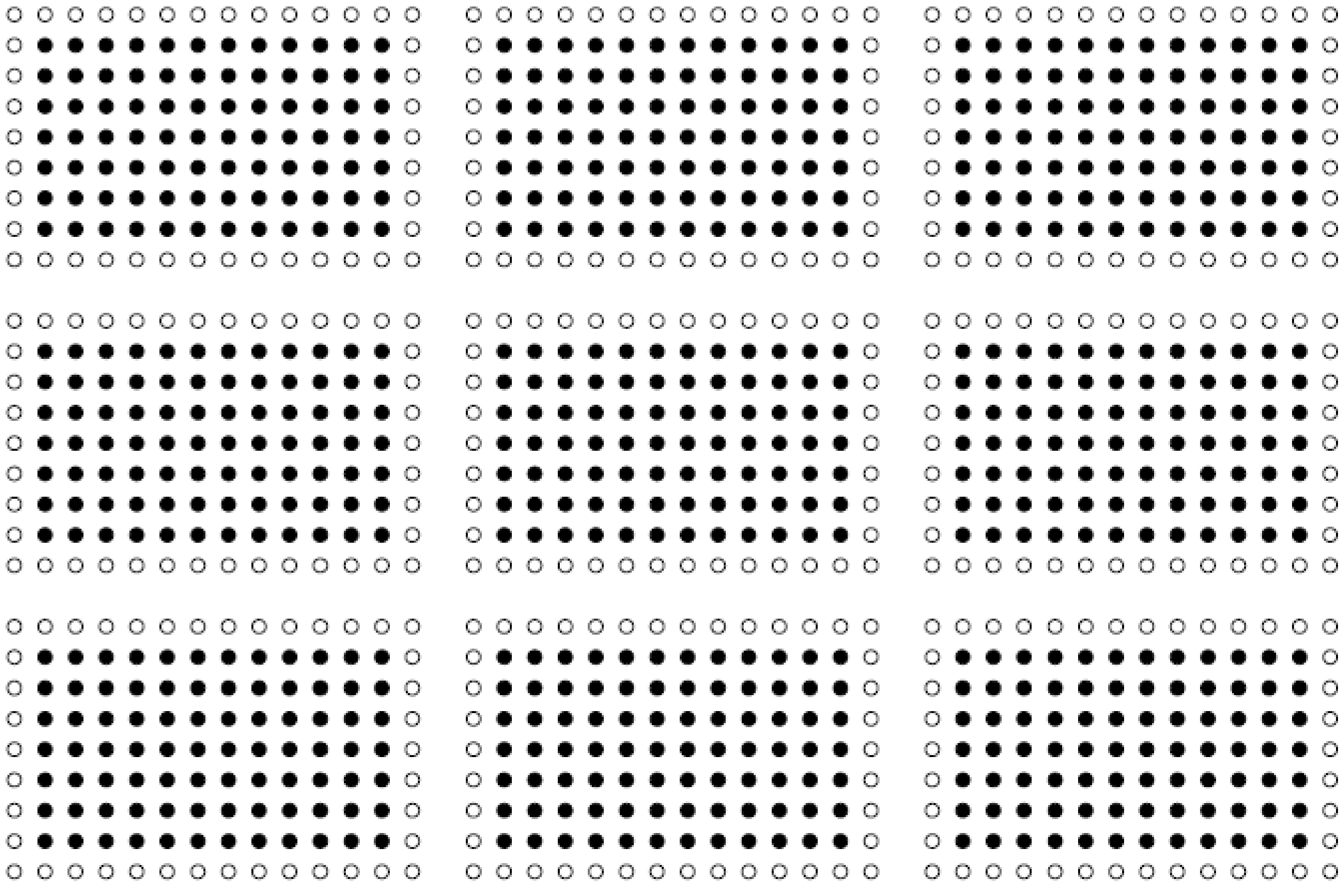
$$v_{0,j}^{n+1} = 0; \quad v_{M,j}^{n+1} = 0; v_{i,0}^{n+1} = 0; v_{i,N}^{n+1} = 0.$$

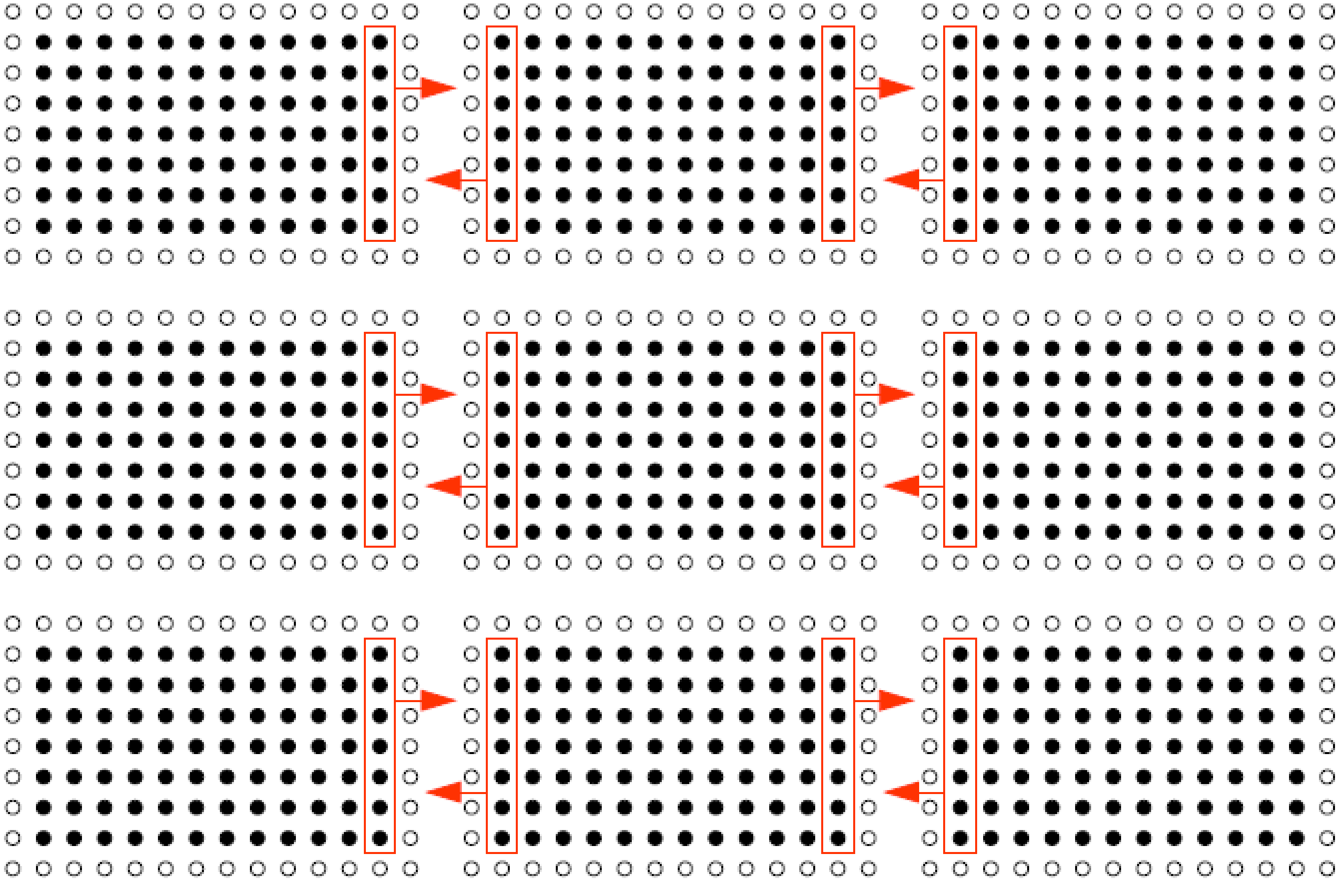
Parallel Computation with Grids

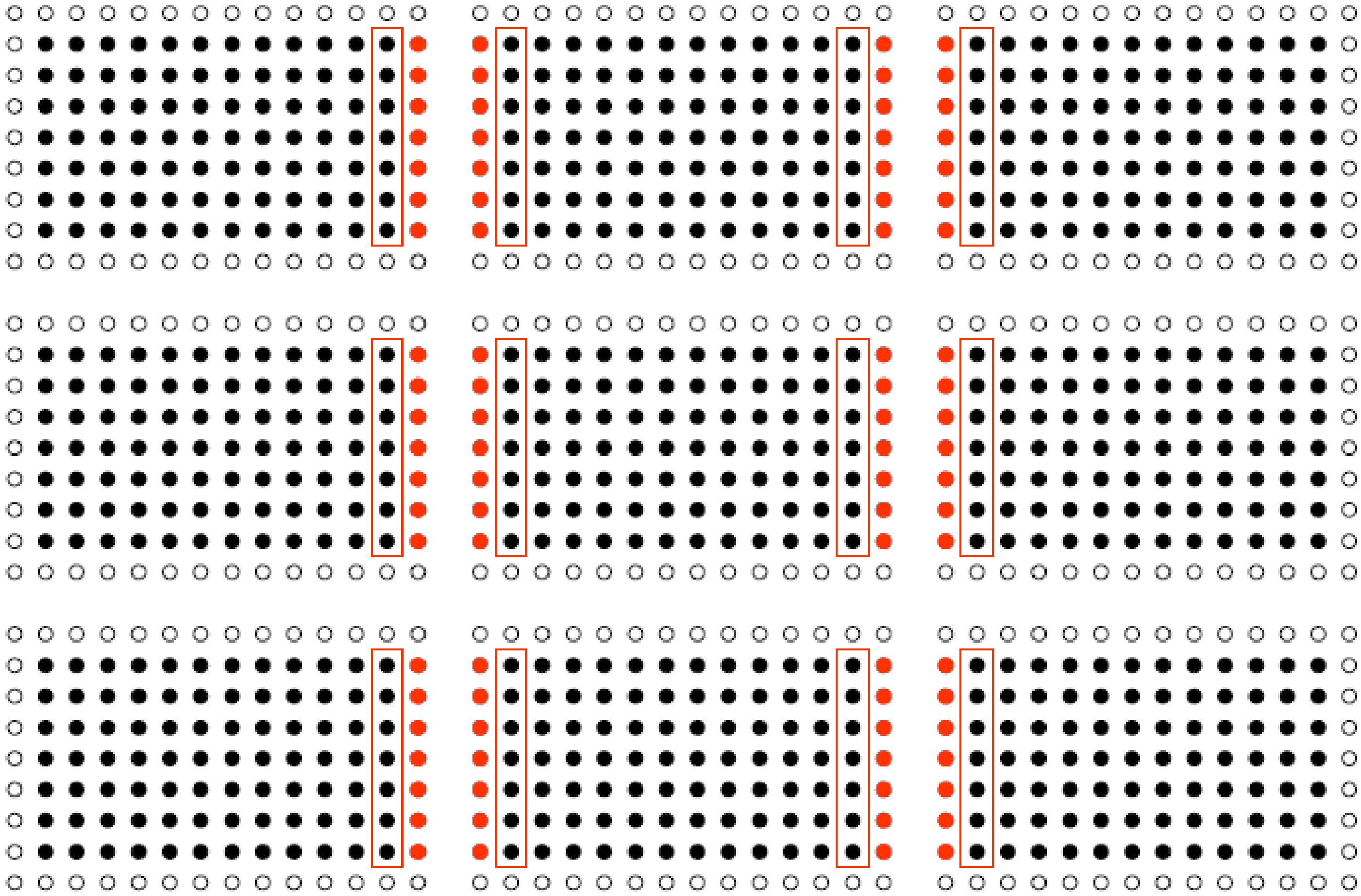
- Partition solution domain into subdomains.
- Distribute subdomains across processors
- Communication between processors is needed to provide interface between subdomains.
  - Communication is needed when stencil for given grid point includes points on another processor
  - For efficiency, ghost points are used for message passing at the end (or begin) of each iteration. Ghost points overlap between two subdomains, so as subgrids.

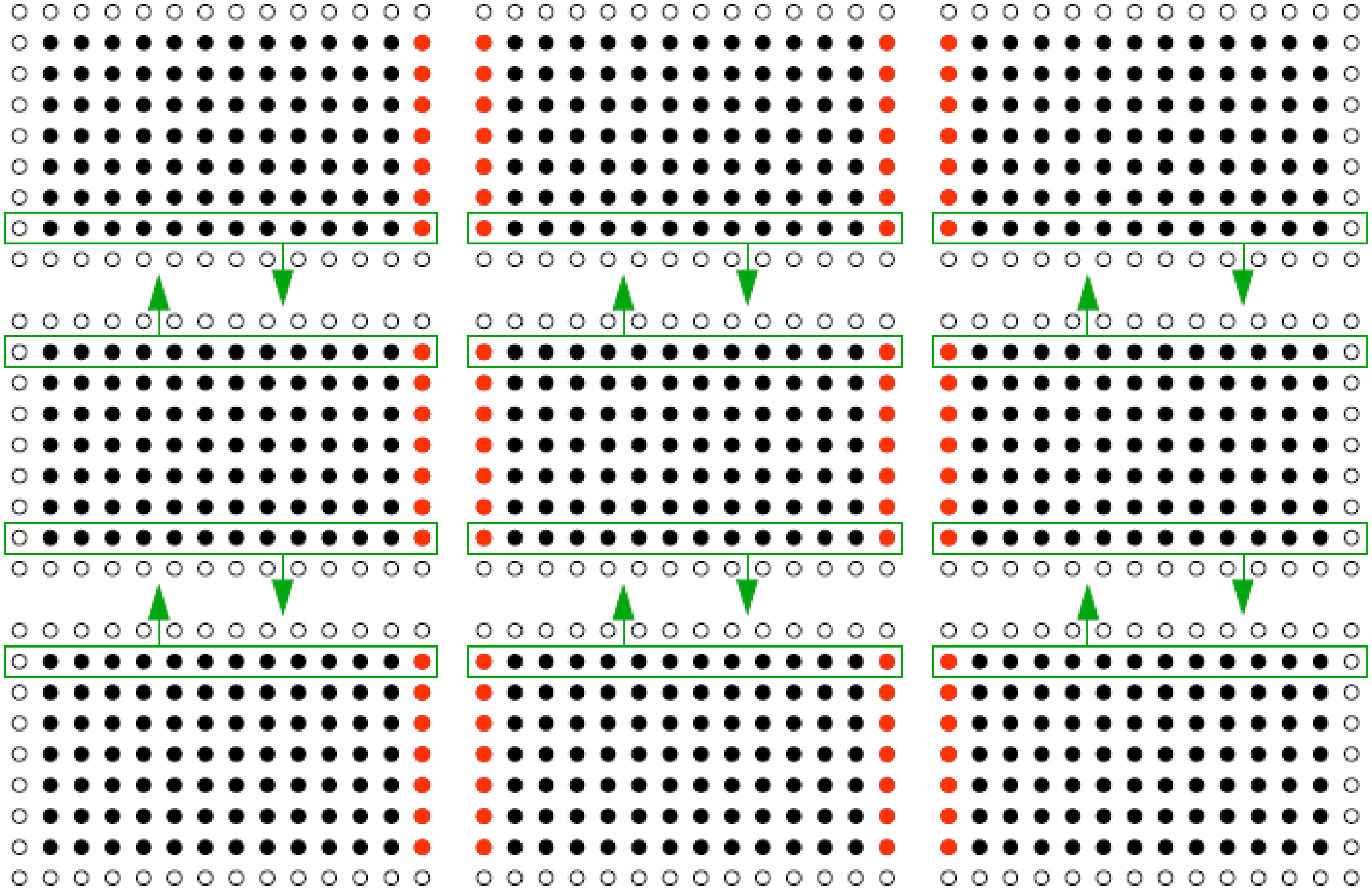
Ghost Points

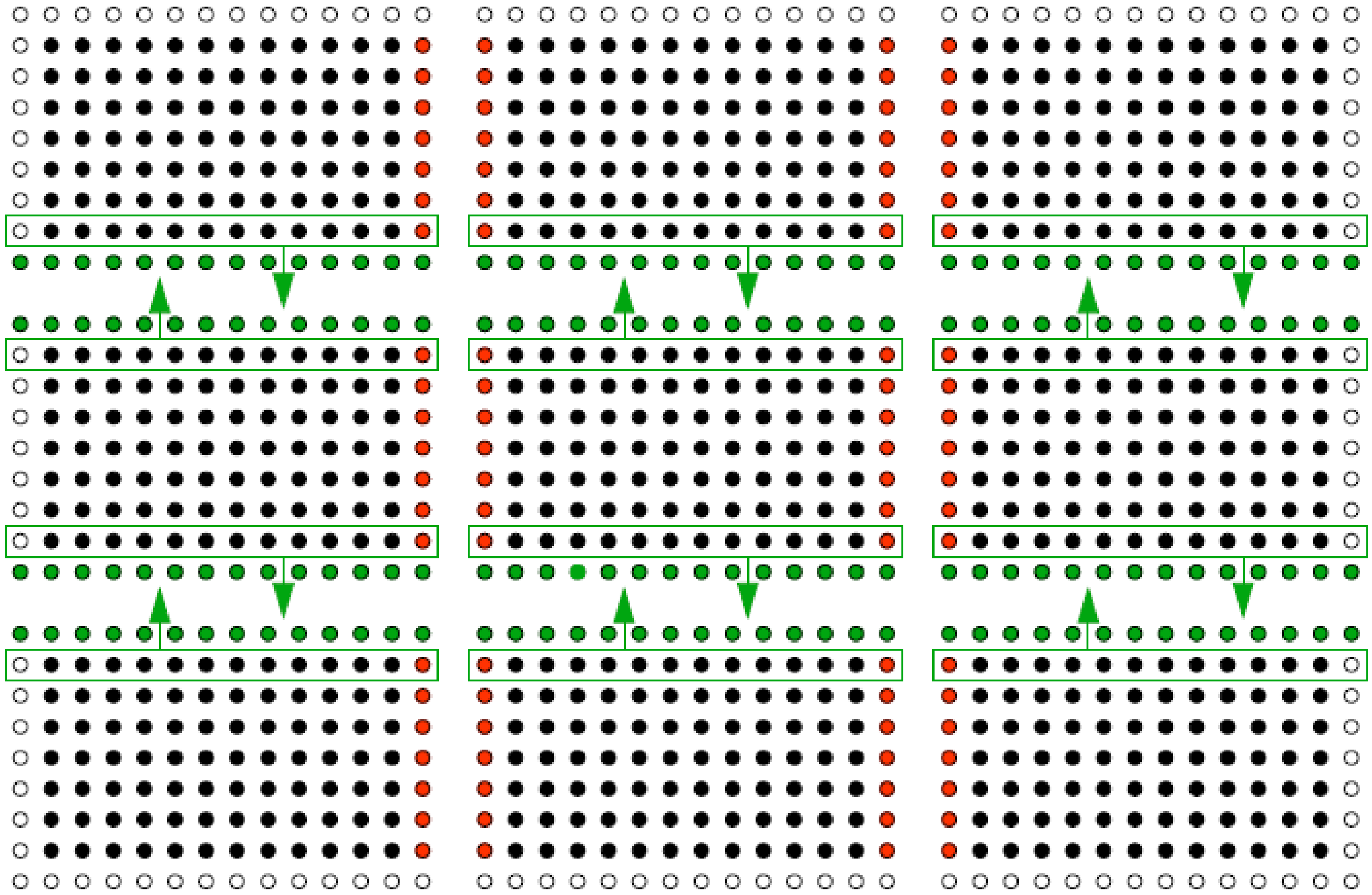












Grid Structure

```
struct _RECT_GRID {
      double L[3];      /* Lower corner of rectangle containing grid */
      double U[3];      /* Upper corner of rectangle containing grid */
      double h[3];      /* Average grid spacings in the grid        */
      int   gmax[3];    /* Number of grid blocks                    */
      int   dim;        /* Dimension of Grid */

            /* Specifications for virtual domains and variable grids */

      double GL[3];     /* Lower corner of global grid */
      double GU[3];     /* Upper corner of global grid */
      double VL[3];     /* Lower corner of virtual domain */
      double VU[3];     /* Upper corner of virtual domain */
      int   lbuf[3];    /* Lower buffer zone width */
      int   ubuf[3];    /* Upper buffer zone width */
};
typedef struct _RECT_GRID RECT_GRID;
```

Solution Storage

```
#define  soln(u, ic, gr)     (u[n_indx((ic), (gr))])
#define  n_indx(ic, gr)     ((ic)[1]*((gr)->gmax[0]+(gr)->lbuf[0]+(gr)->ubuf[0]) + (ic)[0])


double  *u_store, *u;
int         x_size, y_size, i, ic[2];
RECT_GRID *gr;
….
// properly initialize gr.
….
x_size = gr->gmax[0]+gr->lbuf[0]+gr->ubuf[0];
y_size = gr->gmax[1]+gr->lbuf[1]+gr->ubuf[1];

u_store = new double [x-size*y-size];
u = u_store + gr->lbuf[1]*x_size + gr->lbuf[0];


// show state at the first row of the grid
ic[1] = 0;
for(i = -gr->lbuf[0]; i < gr->lbuf[0]+gr->ubuf[0]; i++)
{
   ic[0] = i;
   cout << “state = “ << soln(u,ic,gr) <<endl;
}
```

Communication of Rectangular Lattice States

```
*     i = 0
*
*                l[0]+lbuf[0]     u[0]-ubuf[0]
*                   |              |
*                  \|/            \|/
*     l[1] |------|--------------------------|------|
*          |      |      |           |       |      |
*          |  R   |      |           |       |  R   |
*          |  E   |      |           |       |  E   |
*          |  C   |  S   |           |   S   |  C   |
*          |  E   |  E   |           |   E   |  E   |
*          |  I   |  N   |           |   N   |  I   |
*          |  V   |  D   |           |   D   |  V   |
*          |  E   |      |           |       |  E   |
*          |      |      |           |       |      |
*          |      |      |           |       |      |
*          |      |      |           |       |      |
*     l[1] |------|--------------------------|------|
*     Vl[0]     l[0]                      u[0]  Vu[0]
```

```
*     i = 1
*
*       Vu[1]--------------------------------------------
*           |                                            |
*           |          RECEIVE                           |
*       l[1] |------------------------------------------|
*           |                                            |
*           |            SEND                            |
*           |-------------------------------------------|
*           |                                            |
*           |                                            |
*           |                                            |
*           |                                            |
*           |                                            |
*           |                                            |
*           |-------------------------------------------|
*           |                                            |
*           |            SEND                            |
*        l[1]|------------------------------------------|
*           |                                            |
*           |          RECEIVE                           |
*       Vl[1]--------------------------------------------
*       Vl[0]                                Vu[0]
```

```
// Assume we have created a Cartesian grid topology with communicator
// grid_comm

void scatter_states(
double        *u,
RECT_GRID *gr)
{
    int      my_id, side, dim = 2, i;
    int      me[2];

    MPI_Comm_rank(grid_comm , &my_id);
    MPI_Cart_coords(grid_comm, my_id, 2, me);
    for(i = 0; i < dim;  i++)
    {
        for(side = 0; side < 2; side++)
        {
            MPI_Barrier(MPI_Comm);
            pp_send_interior_states(me, i, side, u);
            pp_receive_interior_states(me, i, (side+1)%2, u);
        }
    }
}
```

```
// Assume G[2] stores orders of process grid
void pp_send_interior_states(
int  *me,
int   dir,
int   side,
double *u)
{
    int   him[2], i, dim = 2;
    int   dst_id;
    int   L[3], U[3];
    double     *storage;

    for (i = 0; i < dim; i++)
          him[i] = me[i];
     him[dir] = (me[dir] + 2*side - 1);
     if (him[dir] < 0)
          him[dir] = G[dir] - 1;
       if (him[dir] >= G[dir])
          him[dir] = 0;
       MPI_Cart_rank(grid_comm, him, &dst_id);

       /// figure out region in which the data need to be sent
       set_send_domain(L,U,dir,side,gr);

       storage = new double [(U[0]-L[0])*(U[1]-L[1])];
       // collect data and put into storage
        …
      //
      MPI_Bsend(storage, (U[0]-L[0])*(U[1]-L[1]), MPI_DOUBLE, dst_id, 100, MPI_COMM);
}
```

```
set_send_domain(int *L, int *U,int dir, int side,RECT_GRID *gr)
{
	int		dim = gr->dim;
	int		*lbuf = gr->lbuf;
	int		*ubuf = gr->ubuf;
	int		*gmax = gr->gmax;
	int		j;
	for (j = 0; j < dir; ++j)
	{
	   L[j] = -lbuf[j];
	   U[j] = gmax[j] + ubuf[j];
	}
	if (side == 0)
	{
	   L[dir] =  0;
	   U[dir]] = lbuf[dir];
	}
	else
	{
	   L[dir] = gmax[dir] - ubuf[dir];
	   U[dir]] = gmax[dir];
	}
	for (j = dir+1; j < dim; ++j)
	{
	   L[j] = -lbuf[j];
	   U[j] = gmax[j] + ubuf[j];
	}
}
```

```
void pp_receive_interior_states(
int  *me,
int   dir,
int   side,
double *u)
{
    int   him[2], i, dim = 2;
    int   src_id;
    int   L[3], U[3];
    double     *storage;
     MPI_Status   *status;

    for (i = 0; i < dim; i++)
          him[i] = me[i];
     him[dir] = (me[dir] + 2*side - 1);
     if (him[dir] < 0)
          him[dir] = G[dir] - 1;
      if (him[dir] >= G[dir])
          him[dir] = 0;
      MPI_Cart_rank(grid_comm, him, &src_id);

      /// figure out region in which the data need to be sent
      set_receive_domain(L,U,dir,side,gr);

      storage = new double [(U[0]-L[0])*(U[1]-L[1])];

      MPI_Recv(storage, (U[0]-L[0])*(U[1]-L[1]), MPI_DOUBLE, src_id, 100, MPI_COMM,&status);
     // Put received data into proper places of u
}
```

```
set_receive_domain(int  *L,int *U,int dir,int side, RECT_GRID *gr)
{
      int           dim = gr->dim;
      int           *lbuf = gr->lbuf;
      int           *ubuf = gr->ubuf;
      int           *gmax = gr->gmax;
      int           j;
      for (j = 0; j < dir; ++j)
      {
         L[j] = -lbuf[j];
         U[j] = gmax[j] + ubuf[j];
      }
      if (side == 0)
      {
         L[dir] = -lbuf[dir];
         U[dir] = 0;
      }
      else
      {
         L[dir] = gmax[dir];
         U[dir] = gmax[dir] + ubuf[dir];
      }
      for (j = dir+1; j < dim; ++j)
      {
         L[j] = -lbuf[j];
         U[j] = gmax[j] + ubuf[j];
      }
}
```

Putting Together

```
int main()
{
    int      i, j, k, Max_steps = 10000, ic[2];
    RECT_GRID     *gr;
    double            *u, *u_prev, *tmp;

    // initialize lattice grid: gr
    // initialize storage: *u;
    // initialize state
    // computation
    for(i = 0; i < Max_steps; i++)
    {
       /// time stepping
       for(j = 0; j < gr->gmax[0]; j++)
       {
           ic[0] = j;
           for(k = 0; k < gr->gmax[1]; k++)
           {
               ic[1] = k;
                // update soln:  soln(u, ic, gr) = soln(u_prev, ic, gr) + … ;
            }
       }
       // communication to update ghost points
       scatter_states( u, gr);

       // swap storage for next step
       tmp = u;          u = u_prev;
       u_prev = tmp;
    }
}
```