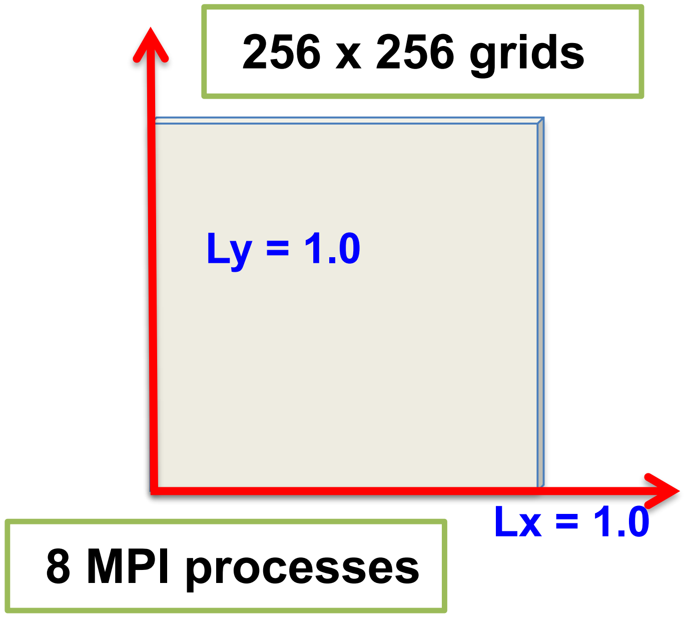
Domain decomposition:

An example of 2D-Poisson equation

Ji-Hoon Kang
(jhkang@kisti.re.kr)

Problem summary

- Two- dimensional Poisson equation



$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)p(x,y)=f(x,y)$$

$$f(x,y)=-2\cos(2\pi y)-4\pi^2 x(1-x)\cos(2\pi y)$$

Boundary condition

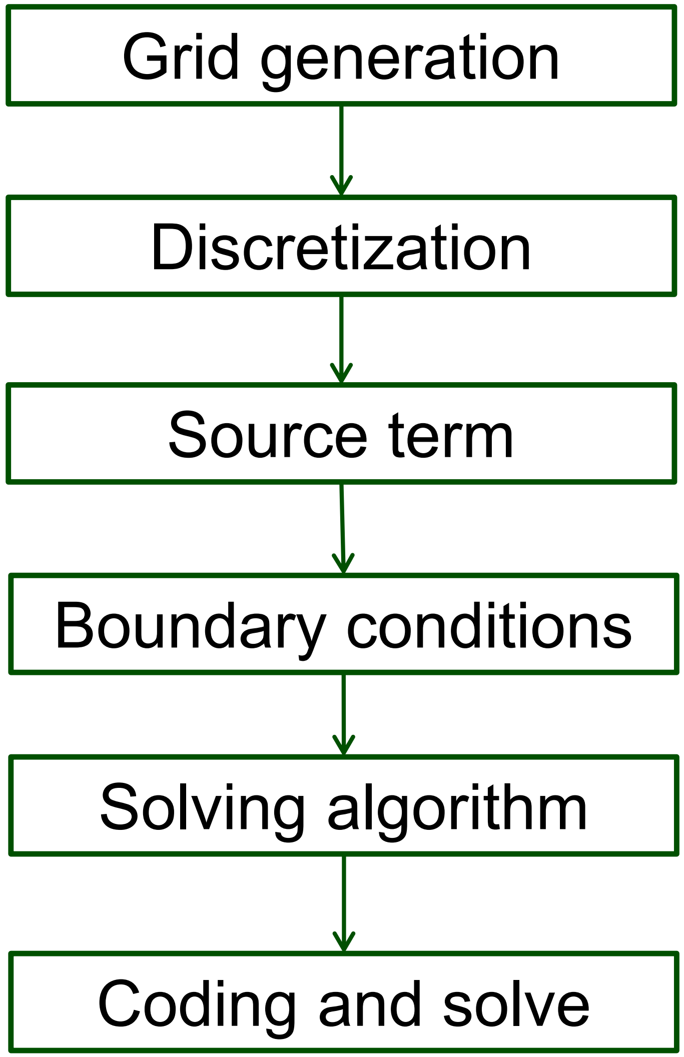
$p(0,y)=p(1,y)=0$	Dirichlet B.C. in x
$p(x,0)=p(x,1)$	Periodic B.C. in y

Exact solution

$$p(x,y)=x(1-x)\cos(2\pi y)$$

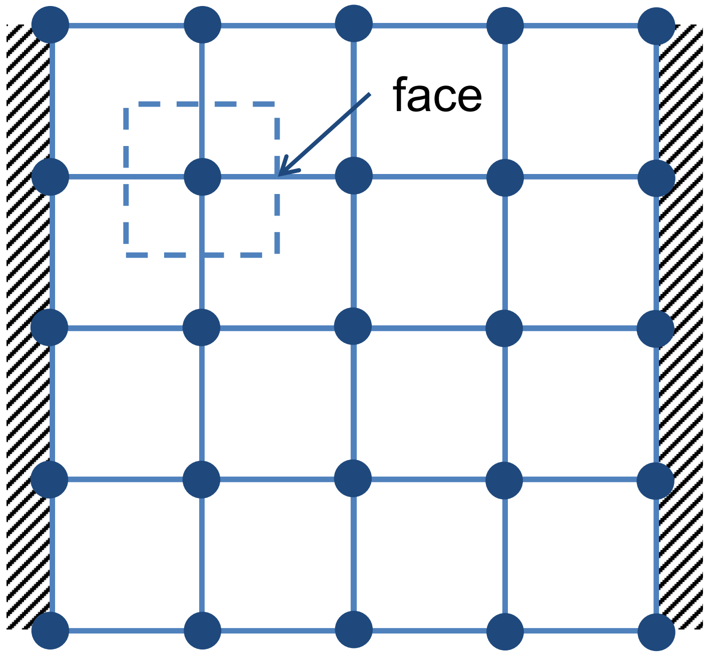
- Numerical method
  - Cell-centered grid
  - Second-order five point discretization
  - Ghost-cell for boundary treatment
  - SOR Red-Black Gauss-Seidel iteration

Solving procedure

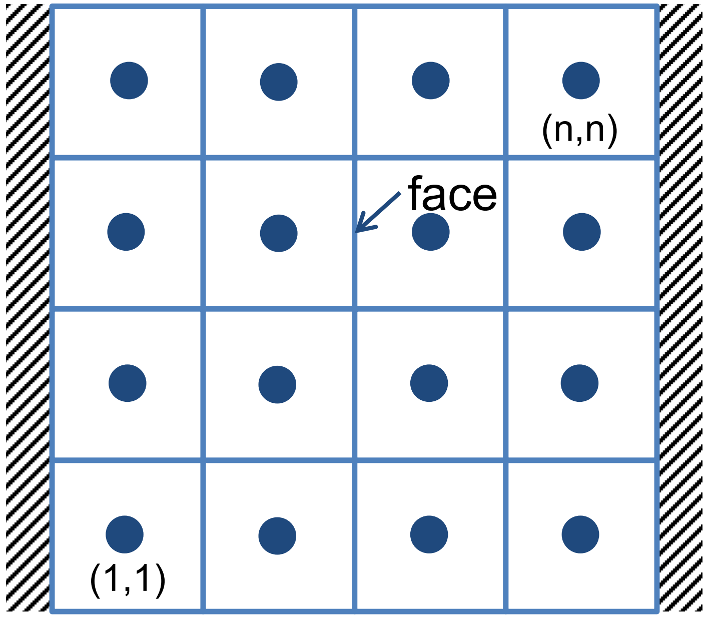


Grid

- **Two representative grid system**



Vertex-centered grid



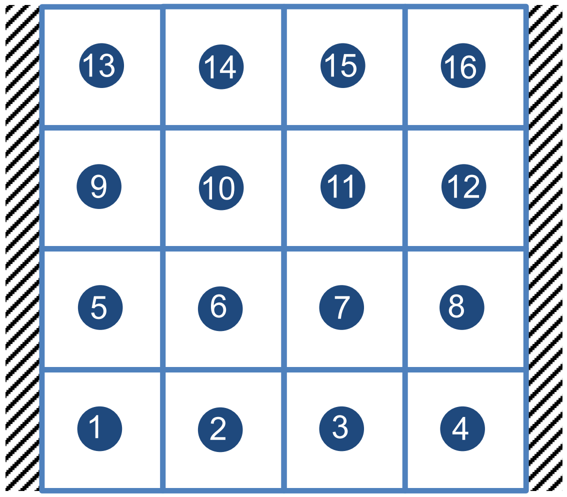
Cell-centered grid

- **Grid type in this tutorial**
  - Cell-centered
  - Uniform

Discretization form

- Matrix form and direct form

$$\frac{p_{i+1,j}-2p_{i,j}+p_{i-1,j}}{\Delta x^2}+\frac{p_{i,j+1}-2p_{i,j}+p_{i,j-1}}{\Delta y^2}=f_{i,j}$$

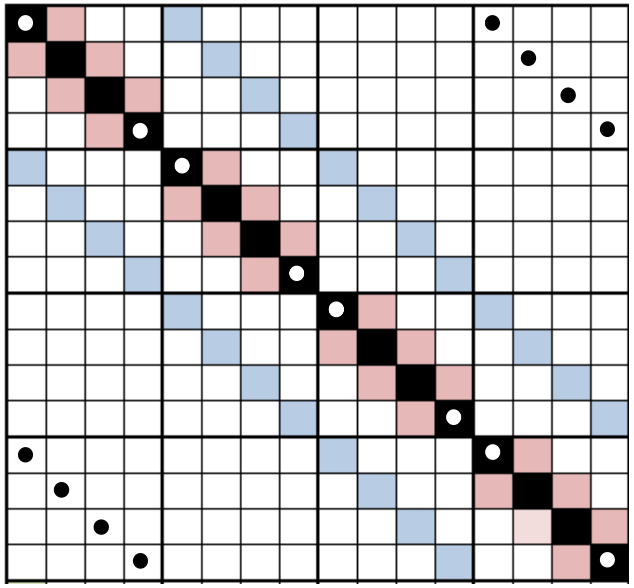


Left b.c. $$\frac{p_{2,j}-3p_{1,j}}{\Delta x^2}+\frac{p_{1,j+1}-2p_{1,j}+p_{1,j-1}}{\Delta y^2}=f_{1,j}$$

Right b.c. $$\frac{3p_{n,j}+p_{n-1,j}}{\Delta x^2}+\frac{p_{n,j+1}-2p_{n,j}+p_{n,j-1}}{\Delta y^2}=f_{n,j}$$

Upper b.c. $$\frac{p_{i+1,n}-2p_{i,n}+p_{i-1,n}}{\Delta x^2}+\frac{p_{i,2}-2p_{i,1}+p_{i,n}}{\Delta y^2}=f_{i,n}$$

Lower b.c. $$\frac{p_{i+1,1}-2p_{i,1}+p_{i-1,1}}{\Delta x^2}+\frac{p_{i,1}-2p_{i,n}+p_{i,n-1}}{\Delta y^2}=f_{i,1}$$



Discretization –node numbering change

- Matrix form and direct form

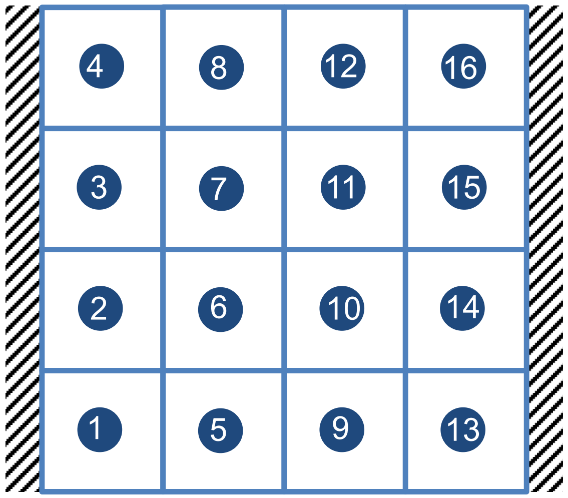
$$\frac{p_{i+1,j}-2p_{i,j}+p_{i-1,j}}{\Delta x^2}+\frac{p_{i,j+1}-2p_{i,j}+p_{i,j-1}}{\Delta y^2}=f_{i,j}$$

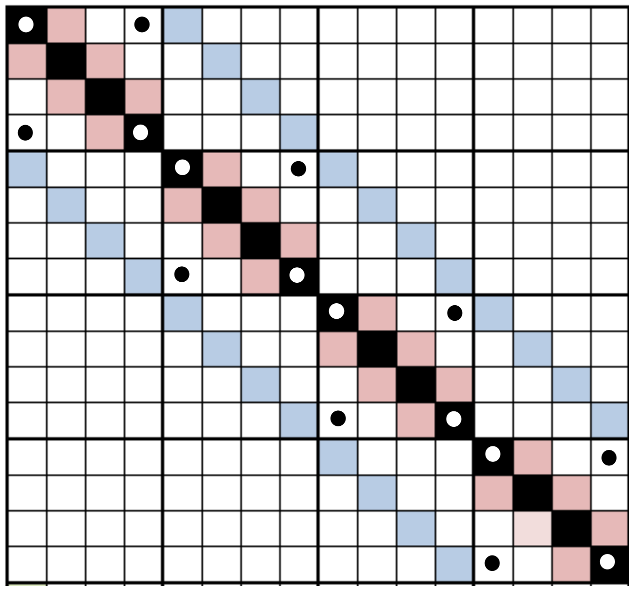
Left b.c. $$\frac{p_{2,j}-3p_{1,j}}{\Delta x^2}+\frac{p_{1,j+1}-2p_{1,j}+p_{1,j-1}}{\Delta y^2}=f_{1,j}$$

Right b.c. $$\frac{3p_{n,j}+p_{n-1,j}}{\Delta x^2}+\frac{p_{n,j+1}-2p_{n,j}+p_{n,j-1}}{\Delta y^2}=f_{n,j}$$

Upper b.c. $$\frac{p_{i+1,n}-2p_{i,n}+p_{i-1,n}}{\Delta x^2}+\frac{p_{i,2}-2p_{i,1}+p_{i,n}}{\Delta y^2}=f_{i,n}$$

Lower b.c. $$\frac{p_{i+1,1}-2p_{i,1}+p_{i-1,1}}{\Delta x^2}+\frac{p_{i,1}-2p_{i,n}+p_{i,n-1}}{\Delta y^2}=f_{i,1}$$



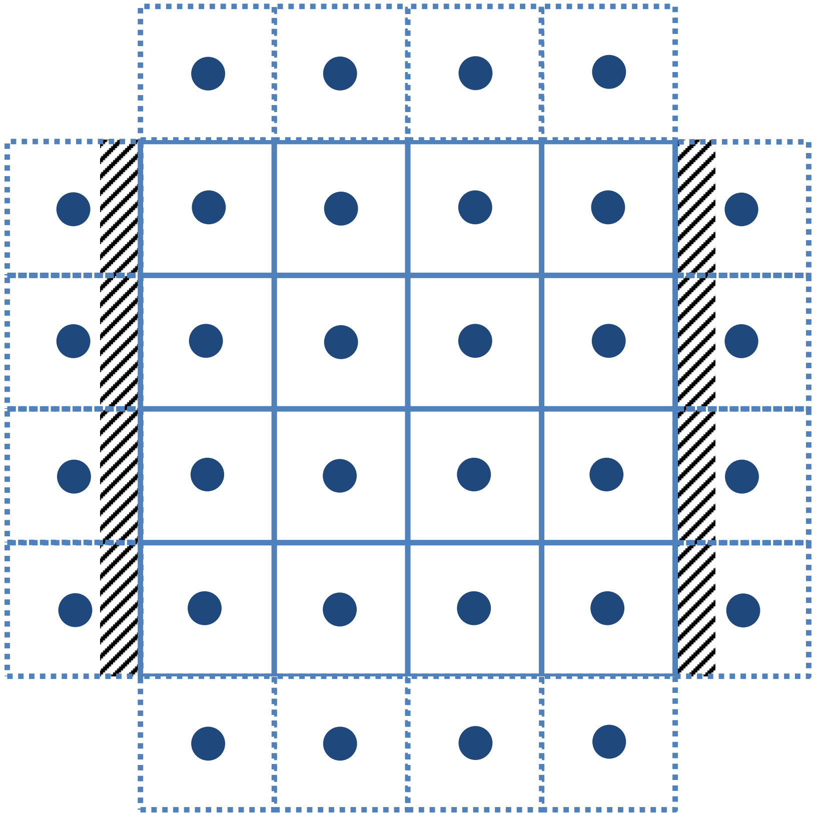


Ghost cell for boundary condition

- **In every step, update ghost cell under the given boundary condition**
  - All grid point inside the domain has the discretized equation

$$p_{i,n+1} = p_{i,1}$$

$$p_{0,j} = -p_{1,j}$$



$$p_{n+1,j} = -p_{n,j}$$

$$p_{i,0} = p_{i,n}$$

Discrized form

- **Iterative method**

$$\frac{p_{i+1,j}-2p_{i,j}+p_{i-1,j}}{\Delta x^2}+\frac{p_{i,j+1}-2p_{i,j}+p_{i,j-1}}{\Delta y^2}=f_{i,j} \qquad \Delta x=\Delta y=\Delta h$$

$$\Rightarrow\quad p_{i,j}=-\frac{1}{4}\Delta h^2 f_{i,j}+\frac{1}{4}\left(p_{i+1,j}+p_{i-1,j}+p_{i,j+1}+p_{i,j-1}\right)$$

Left b.c. $p_{0,j}=-p_{1,j}$

$$p_{1,j}=-\frac{1}{4}\Delta h^2 f_{1,j}+\frac{1}{5}\left(p_{2,j}+p_{0,j}+p_{1,j+1}+p_{1,j-1}\right)$$

Right b.c. $p_{n+1,j}=-p_{n,j}$

$$p_{n,j}=-\frac{1}{4}\Delta h^2 f_{n,j}+\frac{1}{4}\left(p_{n+1,j}+p_{n-1,j}+p_{n,j+1}+p_{n,j-1}\right)$$

Upper b.c. $p_{i,n+1}=p_{i,1}$

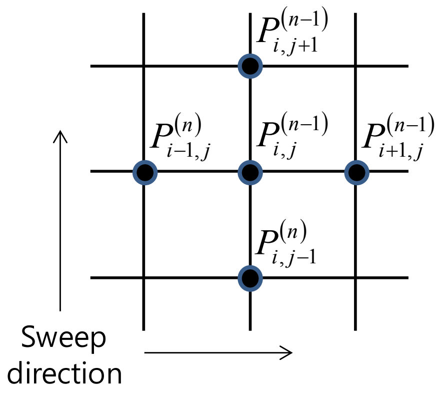
$$p_{i,n}=-\frac{1}{4}\Delta h^2 f_{i,n}+\frac{1}{4}\left(p_{i+1,n}+p_{i-1,n}+p_{i,1}+p_{i,n-1}\right)$$

Lower b.c. $p_{i,0}=p_{i,n}$

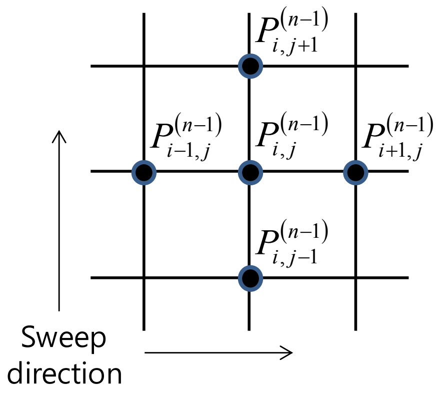
$$p_{i,1}=-\frac{1}{4}\Delta h^2 f_{i,1}+\frac{1}{4}\left(p_{i+1,1}+p_{i-1,1}+p_{i,n}+p_{i,2}\right)$$

Solving algorithm

- **General iterative method**
  - Jacobi (Red-Black Gauss-Siedel)
  - Gauss-Siedel
  - Conjugate-gradient
  - Etc.

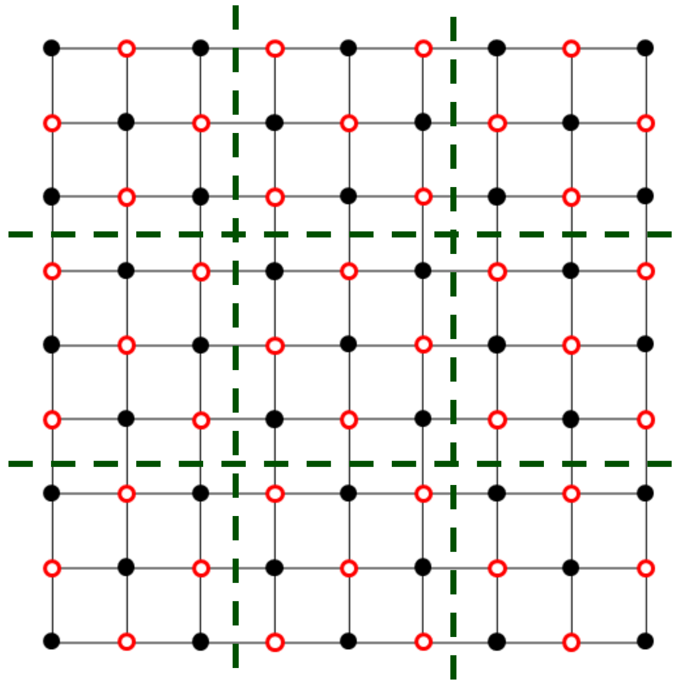


Gauss-Seidel



Jacobi

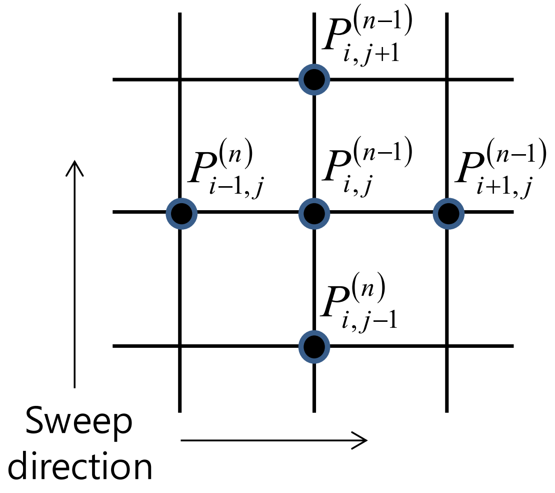
- **Red-Black Gauss-Seidel iteration (or extrapolated Jacobi, EJ)**
  - Good for parallelization
  - 1st pass
    - All red nodes are updated using old values of black nodes
  - 2nd pass
    - All black nodes are updated using updated values of red nodes
  - Calculation in each pass is completely independent.



SOR

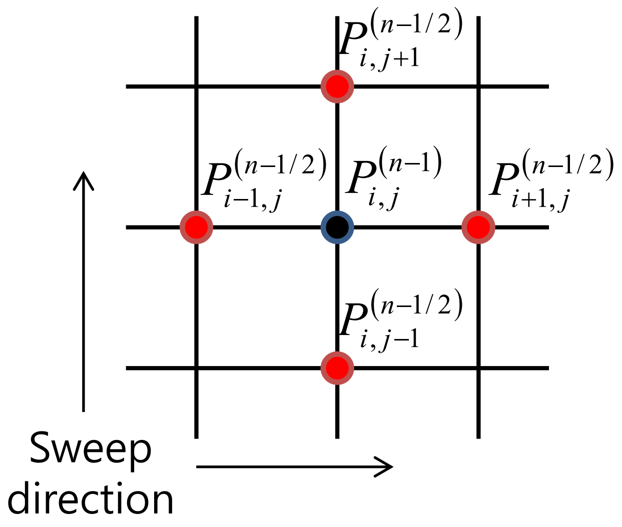
- **Gauss-Seidel**

$$p_{i,j}^{(n)} = \alpha\left[-\frac{1}{4}\Delta h^2 f_{i,j} + \frac{1}{4}\left(p_{i+1,j}^{(n-1)} + p_{i-1,j}^{(n)} + p_{i,j+1}^{(n-1)} + p_{i,j-1}^{(n)}\right)\right] + (1-\alpha)p_{i,j}^{(n-1)}$$

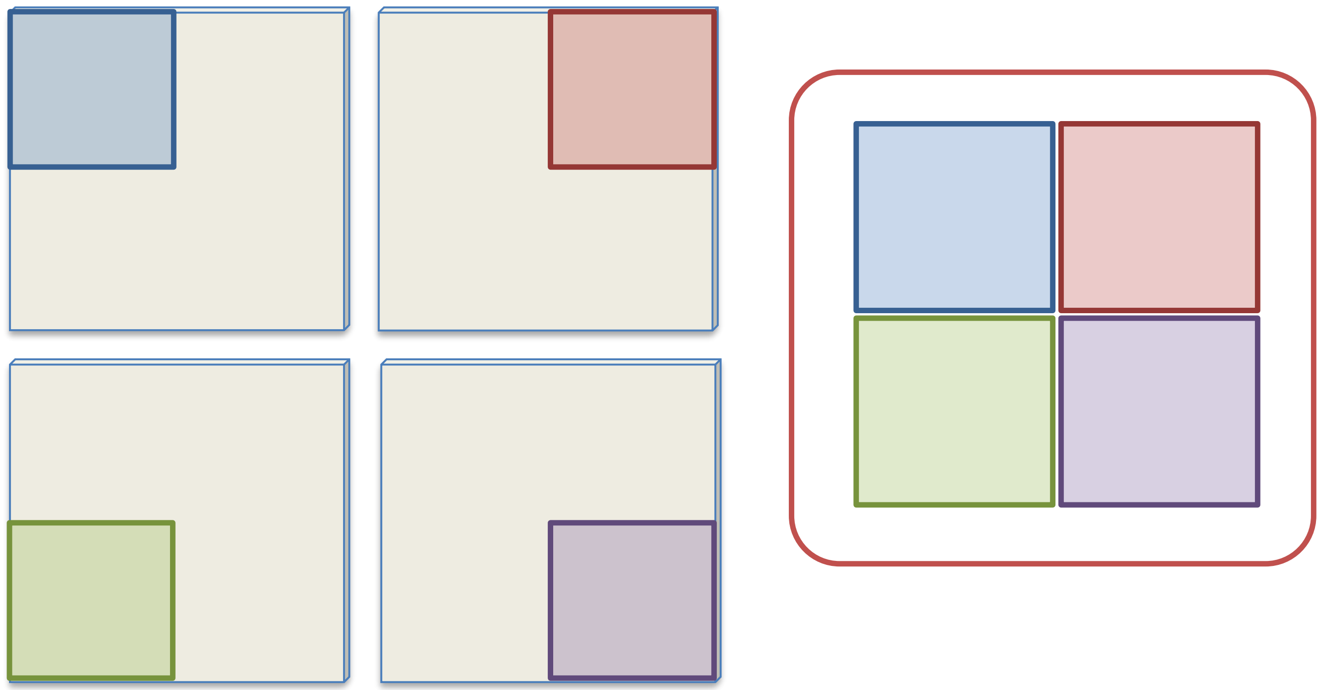


- **Red-black Gauss-Seidel**

$$p_{i,j}^{(n)} = \alpha\left[-\frac{1}{4}\Delta h^2 f_{i,j} + \frac{1}{4}\left(p_{i+1,j}^{(n-1/2)} + p_{i-1,j}^{(n-1/2)} + p_{i,j+1}^{(n-1/2)} + p_{i,j-1}^{(n-1/2)}\right)\right] + (1-\alpha)p_{i,j}^{(n-1)}$$

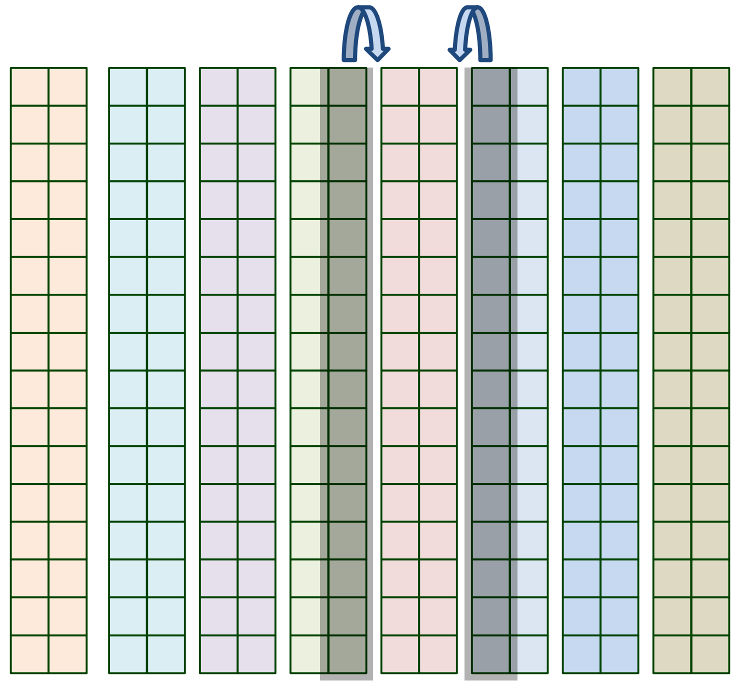


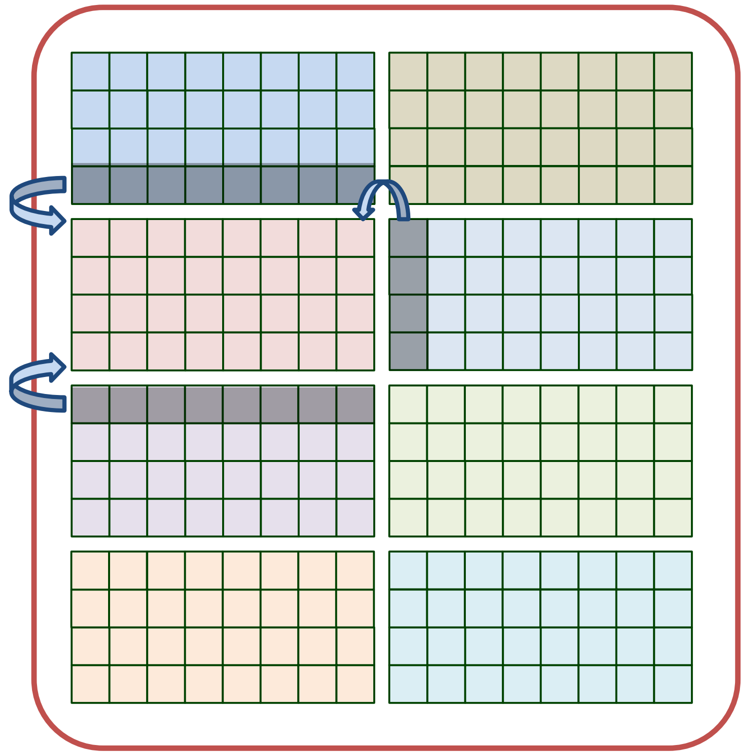
Two approaches of parallelization



	Shared data decomp.	Domain decomp.
Memory space	All data	Its own data
Calculation	Its own data	Its own data
Communication pattern	Update its own data to shared data	Exchange necessary data
Implementation	Simple (It is like OpenMP)	Complicated

Decomposition type





	1-D decomposition	2-D decomposition
Communication pattern	One boundary cells	Two boundary cells
Implementation	Relatively simple	Relatively complicated
Available MPI processes	Nx (or Ny)	Nx × Ny
Communication amount	2 Ny (or 2 Nx)	~ 2(Nx+Ny)/sqrt(p)

Let's start from serial code

- Parallelization steps

1. Break up the domain into blocks. (domain)
2. Assign blocks to MPI processes one-to-one.
3. Provide a "map" of neighbors to each process.

MPI setup

4. Insert communication subroutine calls where needed.

Communication

5. Write or modify your code so it only updates a single block.
6. Adjust the boundary conditions code.

0. Initialize and finalize MPI

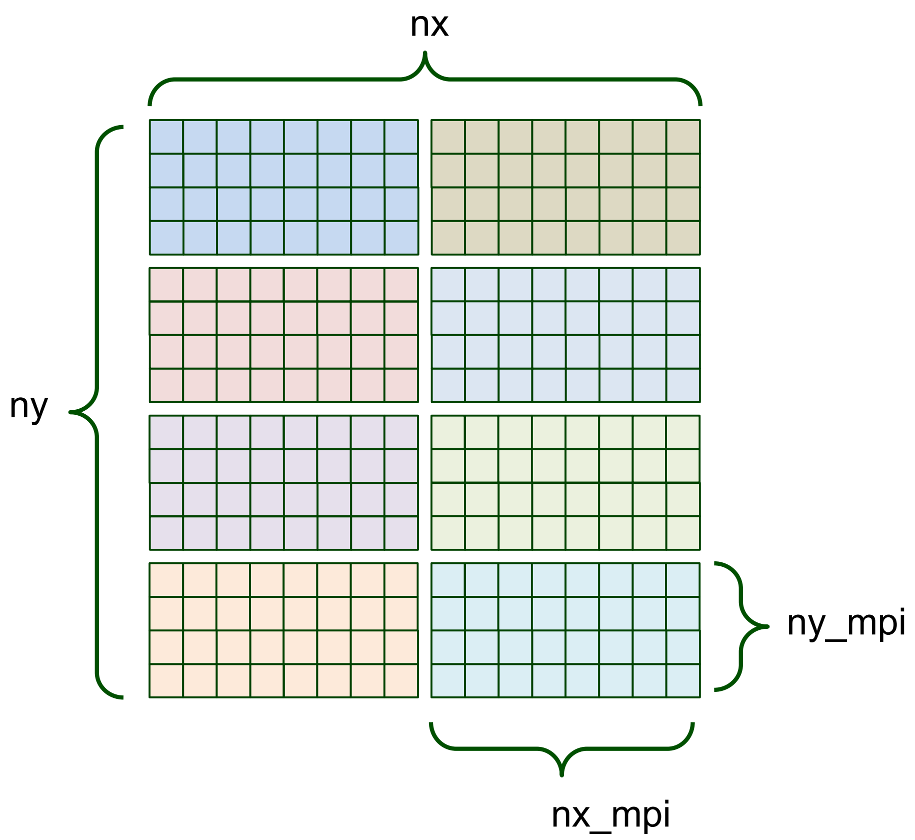
Code fragments in green box

```
typedef struct mympi {
    int nprocs;
    int myrank;
} MYMPI;

void mpi_setup(int nx, int ny, MYMPI *mpi_info) {
}

int main(int argc, char **argv)
{
    int nx = 64, ny = 64;
    int grid_size;
    double length_x = 1.0, length_y = 1.0;
    double PI = atan(1.0)*4.0;

    double dx, dy, x_val, y_val;
    double *pos_x, *pos_y; double *u_exact, *u_solve, *rhs;
    int i,j;

    MYMPI mpi_info;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&mpi_info.nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&mpi_info.myrank);
    ....

    MPI_Finalize();

    return 0;
}
```

1. Break up the domain into blocks

- **Number of meshes**
  - (nx, ny) = (256,256)
- **Number of MPI processes**
  - nprocs = 8
- **Domain decomposition**
  - (mpisize_x, mpisize_y) = (2, 4)
- **Number of meshes in each domain**
  - nx_mpi = nx / mpi_xsize
  - ny_mpi = ny / mpi_ysize

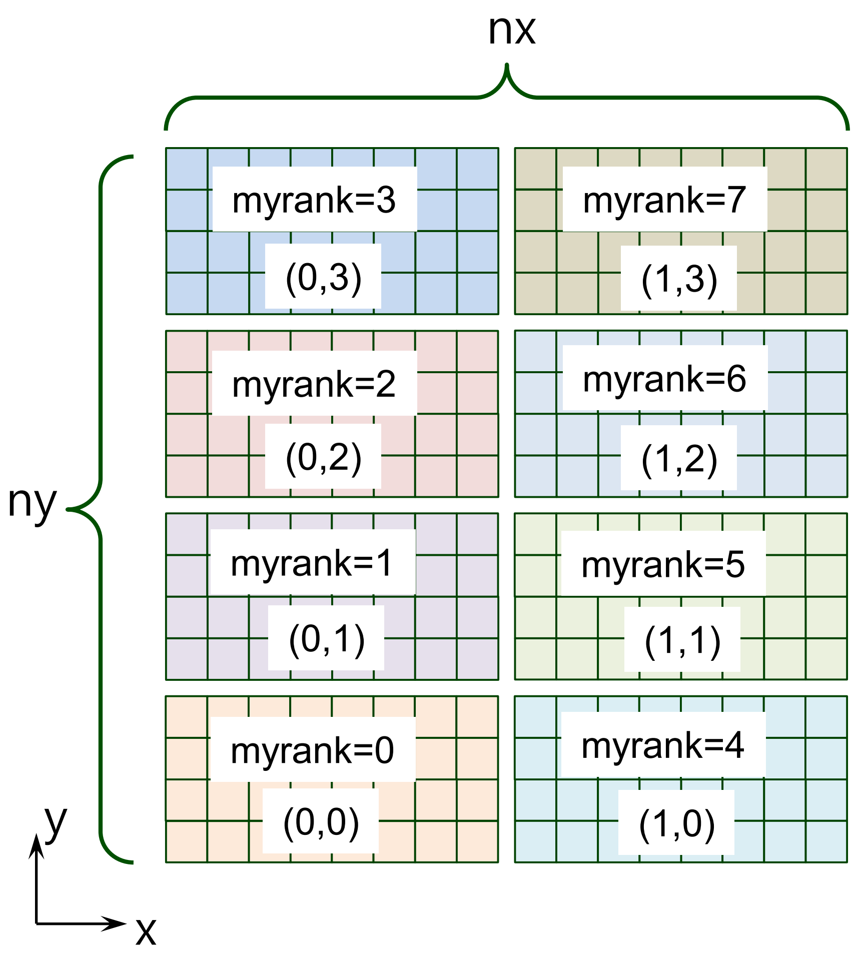
```
typedef struct mympi {
    int nprocs;
    int myrank;
    int nx_mpi, ny_mpi;
    int mpisize_x, mpisize_y;
} MYMPI;
void mpi_setup(int nx, int ny, MYMPI *mpi_info) {
    mpi_info->mpisize_x=2;
    mpi_info->mpisize_y=4;
    mpi_info->nx_mpi=nx/mpi_info->mpisize_x;
    mpi_info->ny_mpi=ny/mpi_info->mpisize_y;
}
```



2. Assign blocks to MPI processes one-to-one (I)

- **Assign MPI rank to decomposed block**
  - x-direction first
  - y-direction first
- **Add decomposed block coordinate to MPI rank map**
  - For process of mpi_rank =5, mpi_xrank=mpi_yrank =1

```
typedef struct mympi {
    int nprocs;
    int myrank;
    int nx_mpi, ny_mpi;
    int mpisize_x, mpisize_y;
    int mpirank_x, mpirank_y;
} MYMPI;
void mpi_setup(int nx, int ny, MYMPI *mpi_info) {
    mpi_info->mpisize_x=2;
    mpi_info->mpisize_y=4;
    mpi_info->nx_mpi=nx/mpi_info->mpisize_x;
    mpi_info->ny_mpi=ny/mpi_info->mpisize_y;
    mpi_info->mpirank_x=mpi_info->myrank/mpi_info->mpisize_y;
    mpi_info->mpirank_y=mpi_info->myrank%mpi_info->mpisize_y;
}
```



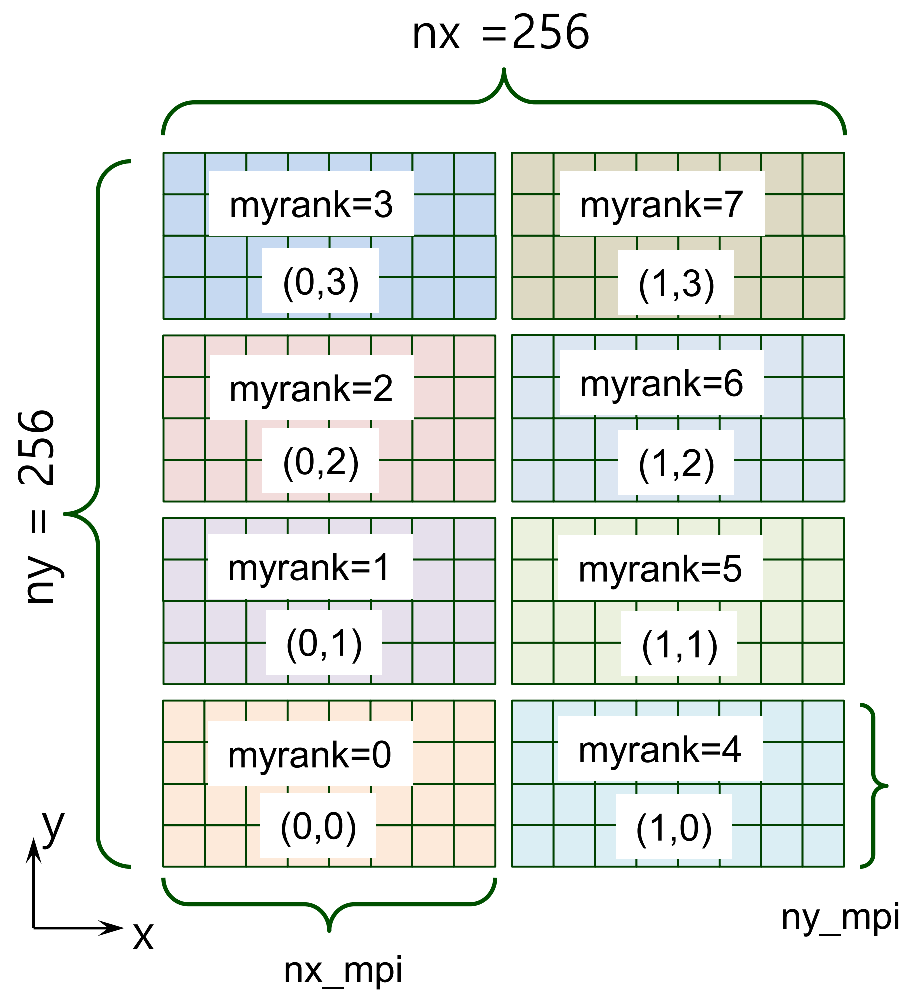
2. Assign blocks to MPI processes one-to-one (II)

Assign global array index to each MPI process

- Search para_range in google
- For process of mpi_rank=5,
  starting index of global array is (129, 65)
  and ending index is (256,128)

```
// struct variable
int ista,iend,jsta,jend
```

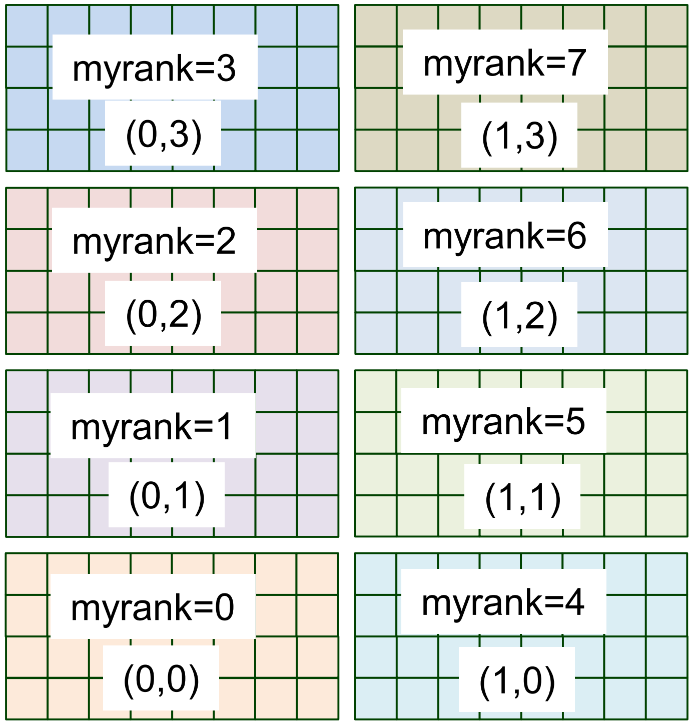
```
// mpi_setup
 ista = mpirank_x * nx_mpi + 1
 iend = mpirank_x * nx_mpi + nx_mpi
 jsta = mpirank_y * ny_mpi + 1
 jend = mpirank_y * ny_mpi + ny_mpi
```



3. Provide a "map" of neighbors to each process. (1)

- **Assigning east, west, south, north process**
  - For the process of myrank=5,
    w_rank=1
    e_rank=??.
    s_rank=4
    n_rank=6
  - Except boundary block,

```
typedef struct mympi {
    int nprocs;
    int myrank;
    int nx_mpi, ny_mpi;
    int mpisize_x, mpisize_y;
    int mpirank_x, mpirank_y;
    int w_rank, e_rank, s_rank, n_rank;
} MYMPI;
void mpi_setup(int nx, int ny, MYMPI *mpi_info) {
    mpi_info->mpisize_x=2;
    mpi_info->mpisize_y=4;
    mpi_info->nx_mpi=nx/mpi_info->mpisize_x;
    mpi_info->ny_mpi=ny/mpi_info->mpisize_y;
    mpi_info->mpirank_x=mpi_info->myrank/mpi_info->mpisize_y;
    mpi_info->mpirank_y=mpi_info->myrank%mpi_info->mpisize_y;
}
```

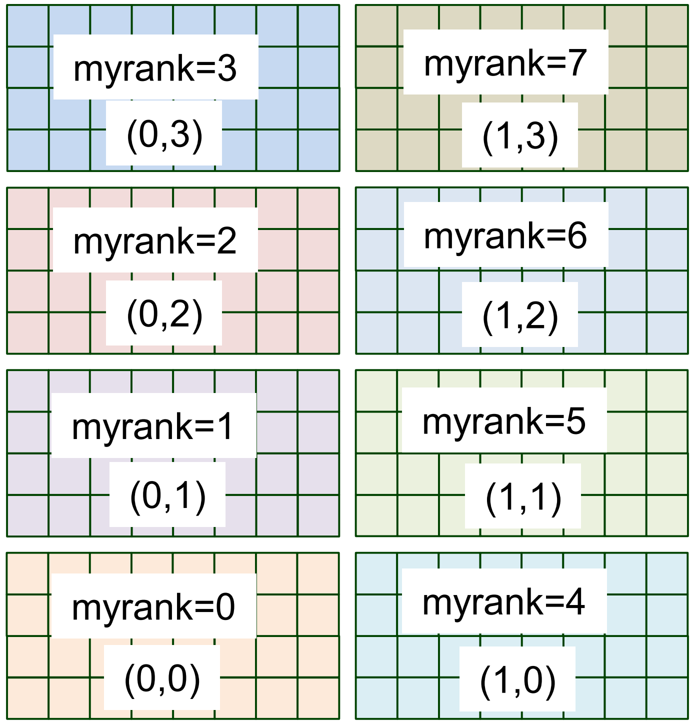


3. Provide a "map" of neighbors to each process. (II)

Treatment of boundary block

- Assign negative value for empty neighbor (or MPI_PROC_NULL)
  - For the process of mpi_rank=5, rankE=-1
- Use periodicity in y-direction
  - For the process of mpi_rank=7, rankN=4

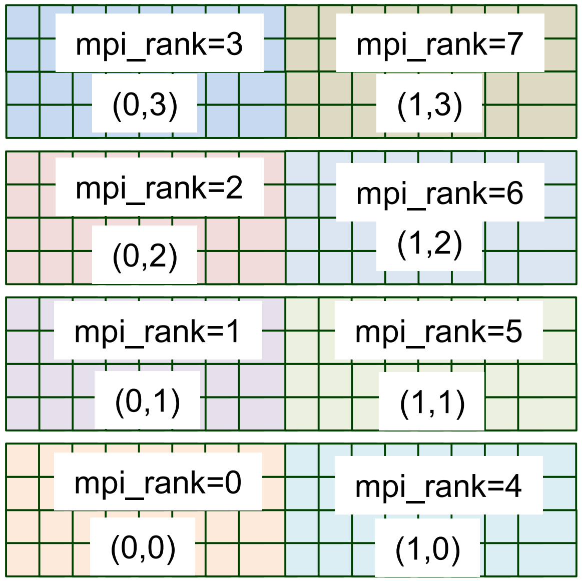
```
void mpi_setup(int nx, int ny, MYMPI *mpi_info){
    if(mpi_info->mpirank_x==mpi_info->mpisize_x-1) {
        mpi_info->e_rank=-1; }
    else {
        mpi_info->e_rank=mpi_info->myrank+mpi_info->mpisize_y; }

    if(mpi_info->mpirank_x==0) {
        mpi_info->w_rank=-1;}
    else {
        mpi_info->w_rank=mpi_info->myrank-mpi_info->mpisize_y;}

    if(mpi_info->mpirank_y==mpi_info->mpisize_y-1) {
        mpi_info->n_rank=mpi_info->myrank+1-mpi_info->mpisize_y; }
    else {
        mpi_info->n_rank=mpi_info->myrank+1; }

    if(mpi_info->mpirank_y==0) {
        mpi_info->s_rank=mpi_info->myrank-1+mpi_info->mpisize_y; }
    else {
        mpi_info->s_rank=mpi_info->myrank-1; }
}
```



MPI setup results

```
========== mpi rank information ==========
 (mpi rank, mpi size) = (    0,    8)
 (  x rank,   y rank) = (    0,    0)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (   -1,    4)
 (  s rank,   n rank) = (    3,    1)
 (    ista,     iend) = (    1,  128)
 (    jsta,     jend) = (    1,   64)
========== mpi rank information ==========
 (mpi rank, mpi size) = (    1,    8)
 (  x rank,   y rank) = (    0,    1)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (   -1,    5)
 (  s rank,   n rank) = (    0,    2)
 (    ista,     iend) = (    1,  128)
 (    jsta,     jend) = (   65,  128)
========== mpi rank information ==========
 (mpi rank, mpi size) = (    2,    8)
 (  x rank,   y rank) = (    0,    2)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (   -1,    6)
 (  s rank,   n rank) = (    1,    3)
 (    ista,     iend) = (    1,  128)
 (    jsta,     jend) = (  129,  192)
========== mpi rank information ==========
 (mpi rank, mpi size) = (    3,    8)
 (  x rank,   y rank) = (    0,    3)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (   -1,    7)
 (  s rank,   n rank) = (    2,    0)
 (    ista,     iend) = (    1,  128)
 (    jsta,     jend) = (  193,  256)
```

```
========== mpi rank information ==========
 (mpi rank, mpi size) = (    4,    8)
 (  x rank,   y rank) = (    1,    0)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (    0,   -1)
 (  s rank,   n rank) = (    7,    5)
 (    ista,     iend) = (  129,  256)
 (    jsta,     jend) = (    1,   64)
========== mpi rank information ==========
 (mpi rank, mpi size) = (    5,    8)
 (  x rank,   y rank) = (    1,    1)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (    1,   -1)
 (  s rank,   n rank) = (    4,    6)
 (    ista,     iend) = (  129,  256)
 (    jsta,     jend) = (   65,  128)
========== mpi rank information ==========
 (mpi rank, mpi size) = (    6,    8)
 (  x rank,   y rank) = (    1,    2)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (    2,   -1)
 (  s rank,   n rank) = (    5,    7)
 (    ista,     iend) = (  129,  256)
 (    jsta,     jend) = (  129,  192)
========== mpi rank information ==========
 (mpi rank, mpi size) = (    7,    8)
 (  x rank,   y rank) = (    1,    3)
 (  x size,   y size) = (    2,    4)
 (  w rank,   e rank) = (    3,   -1)
 (  s rank,   n rank) = (    6,    4)
 (    ista,     iend) = (  129,  256)
 (    jsta,     jend) = (  193,  256)
```



Communication implementation

- Parallelization steps

1. Break up the domain into blocks (domain).
2. Assign blocks to MPI processes one-to-one.
3. Provide a "map" of neighbors to each process.

MPI setup

4. Insert communication subroutine calls where needed.

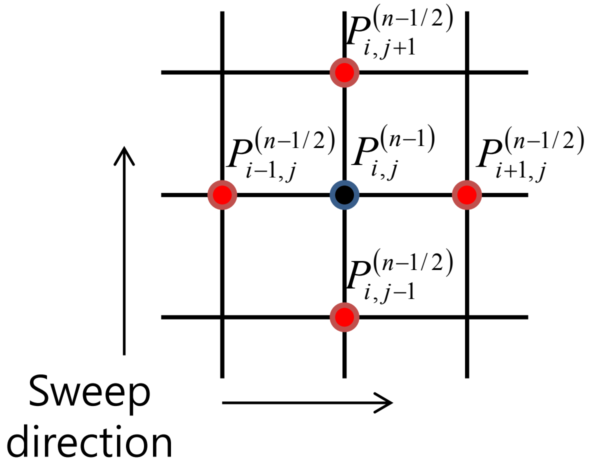
Communication

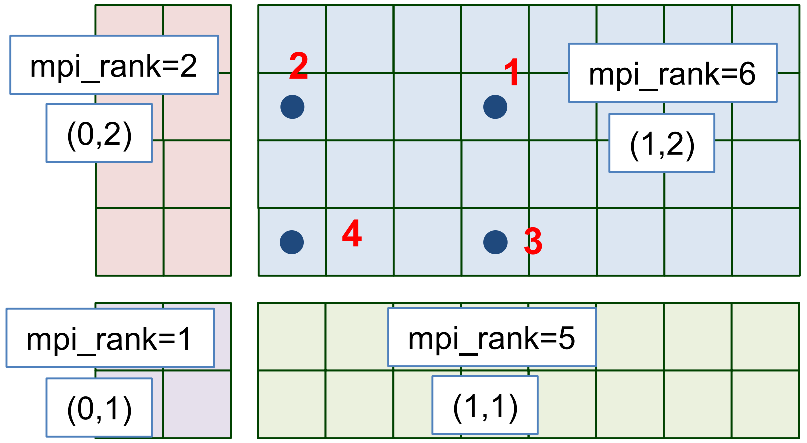
5. Write or modify your code so it only updates a single block.
6. Adjust the boundary conditions code.

4. Insert communication call

Where do we need communication?

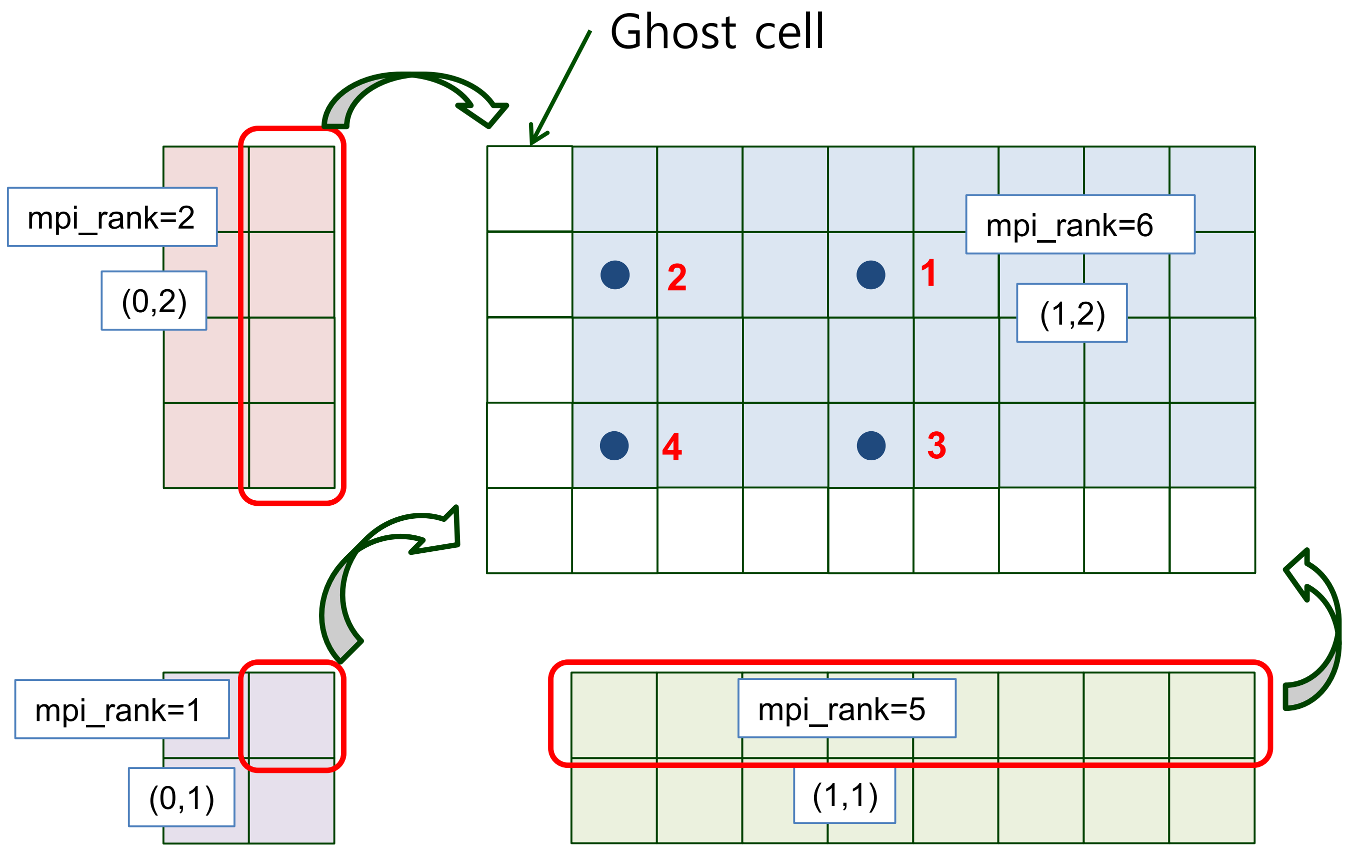
- In Red-Black Gauss-Seidel iteration, the values of neighboring grids are required.
- For the boundary cell of each decomposed block, the value in neighboring domain is required.
  - Point 1: No communication
  - Point 2: Value from west (mpi_rank=2) is required
  - Point 3: Value from south (mpi_rank=5) is required
  - Point 4: Values from west and south are required





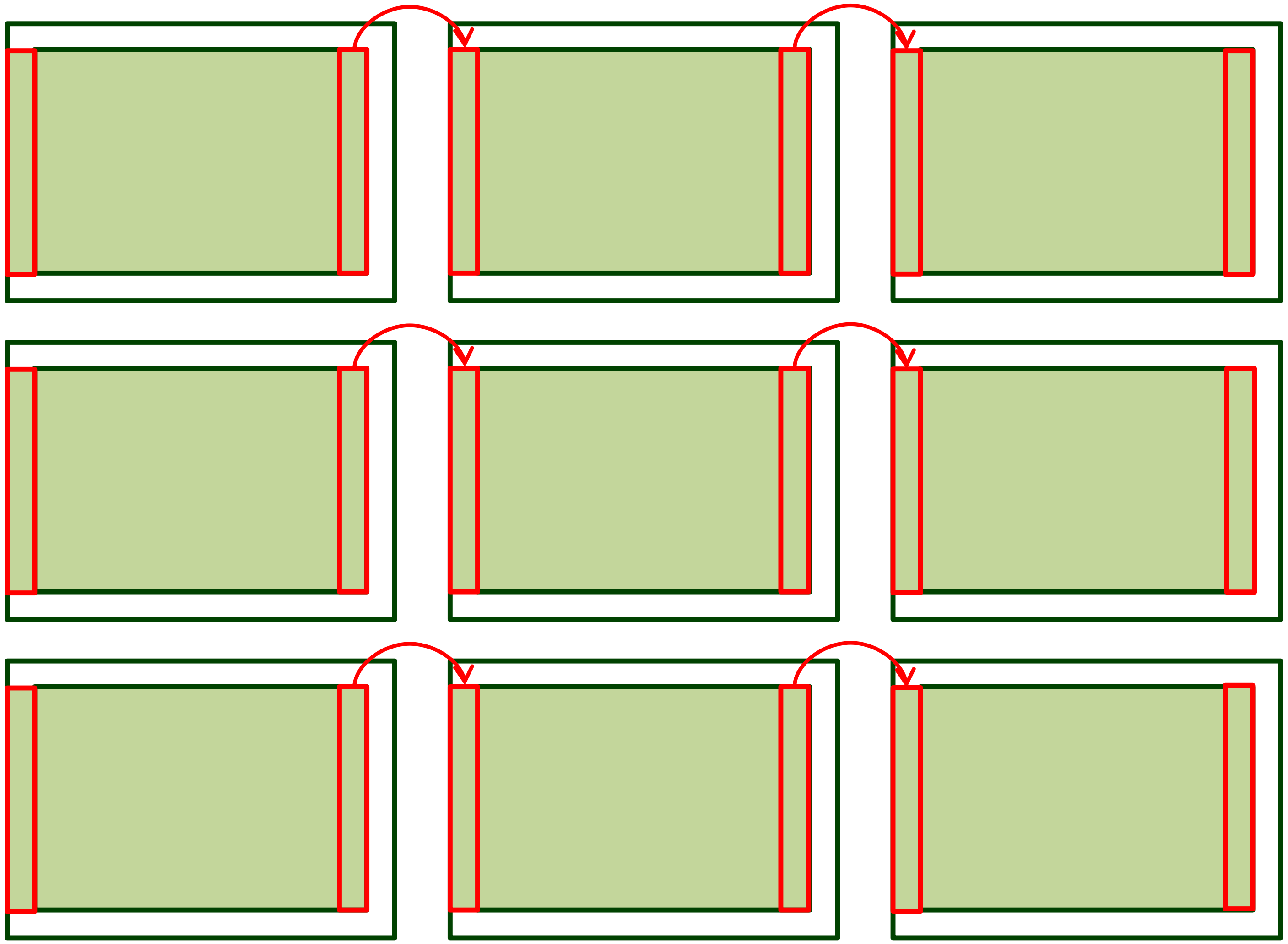
Solution: Ghost cell

- Update the value of ghost cell by using MPI_Send & MPI_Recv before iteration.



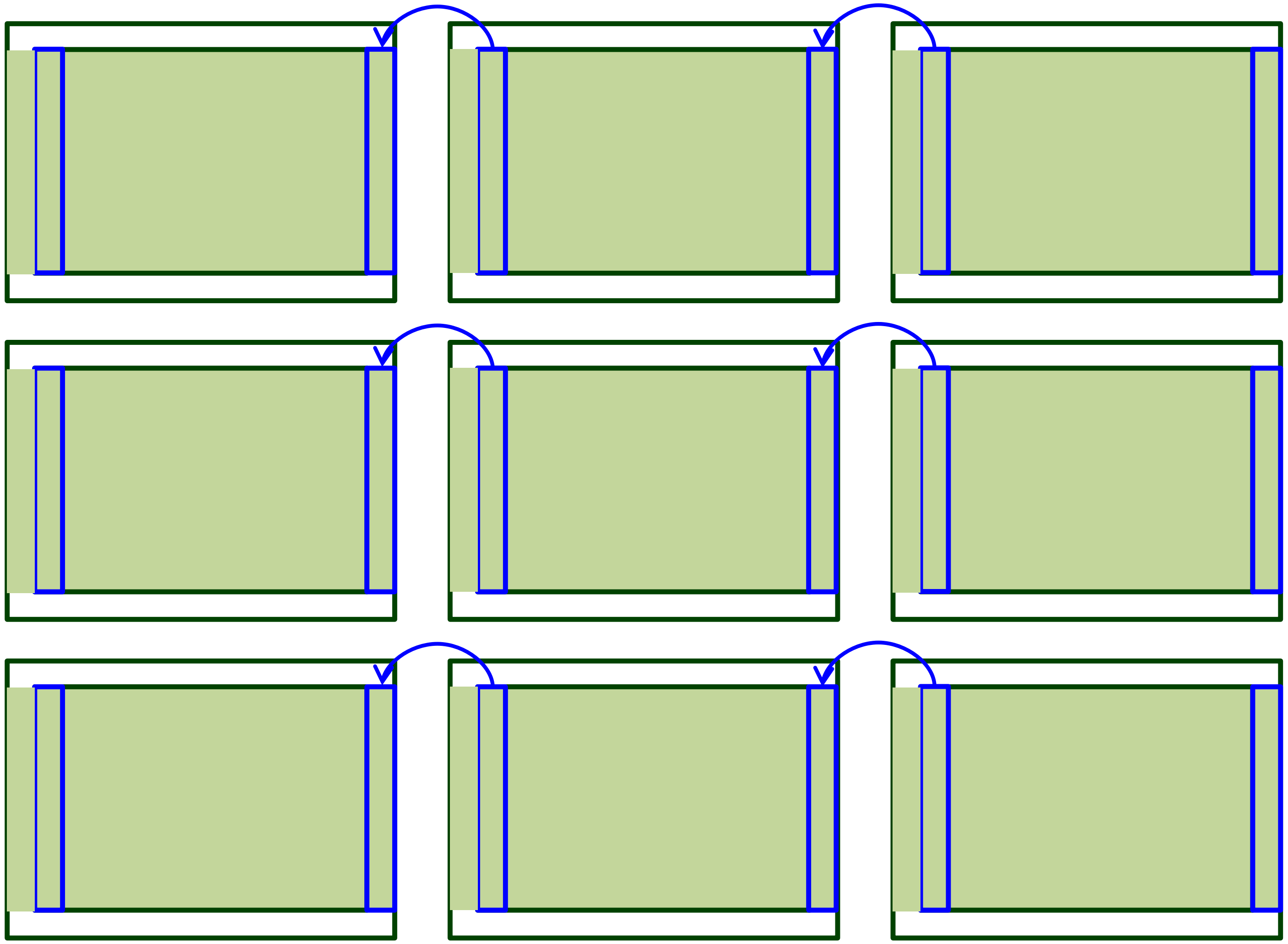
Communication process design

- 1. Update the west ghost cell from west neighboring domain



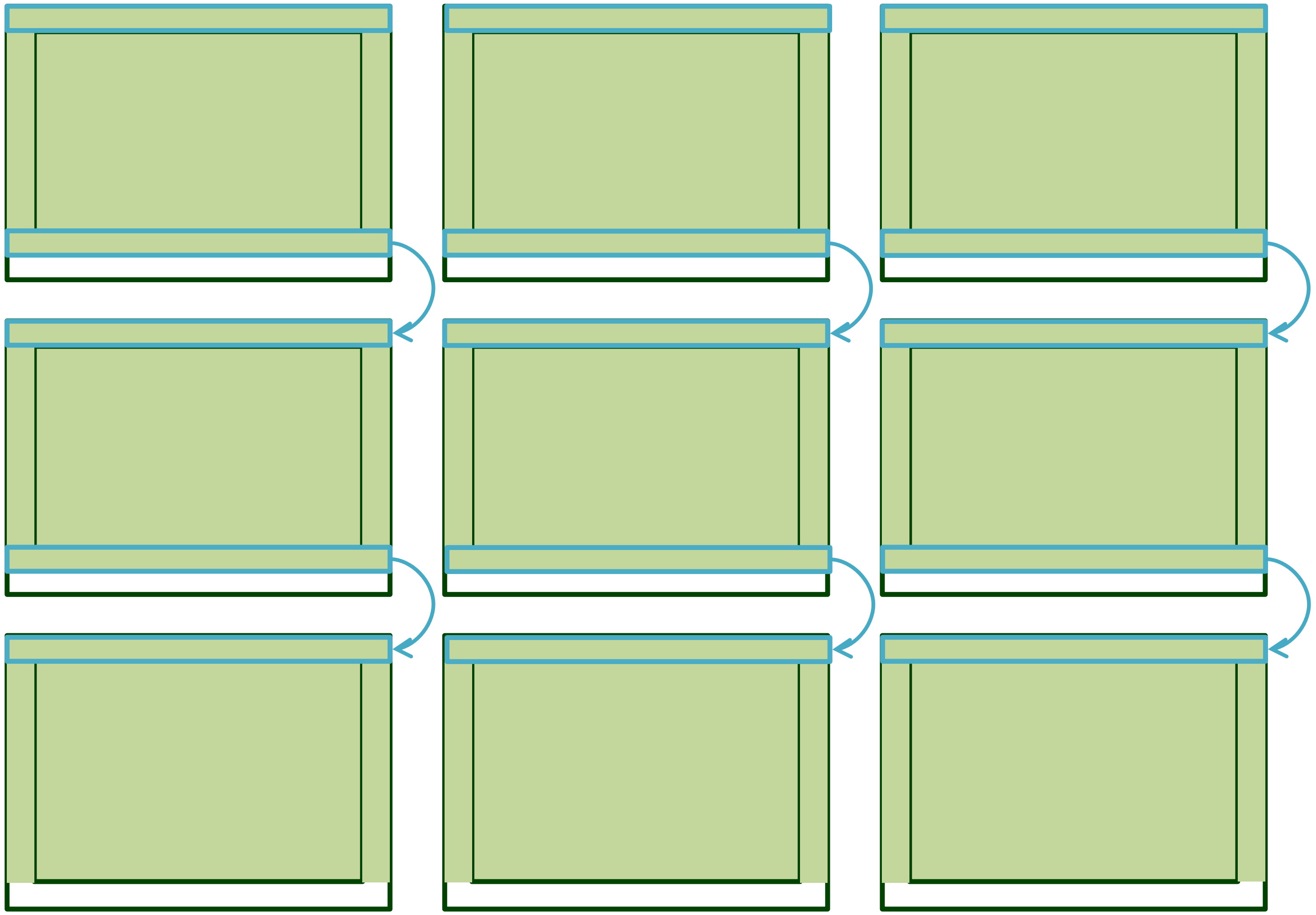
Communication process design

- 1. Update the east ghost cell from east neighboring domain



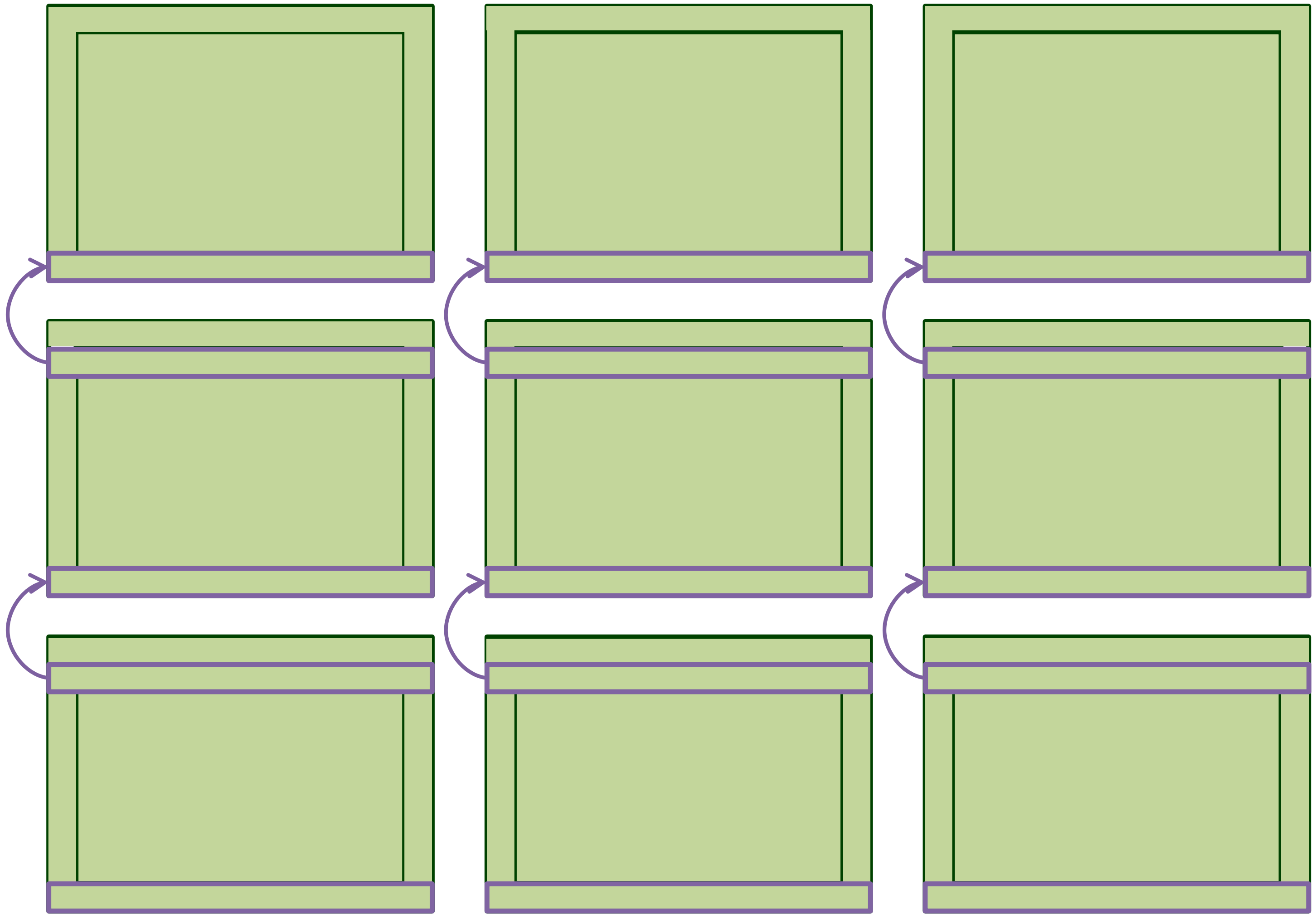
Communication process design

- 3. Update the north ghost cell from north neighboring domain



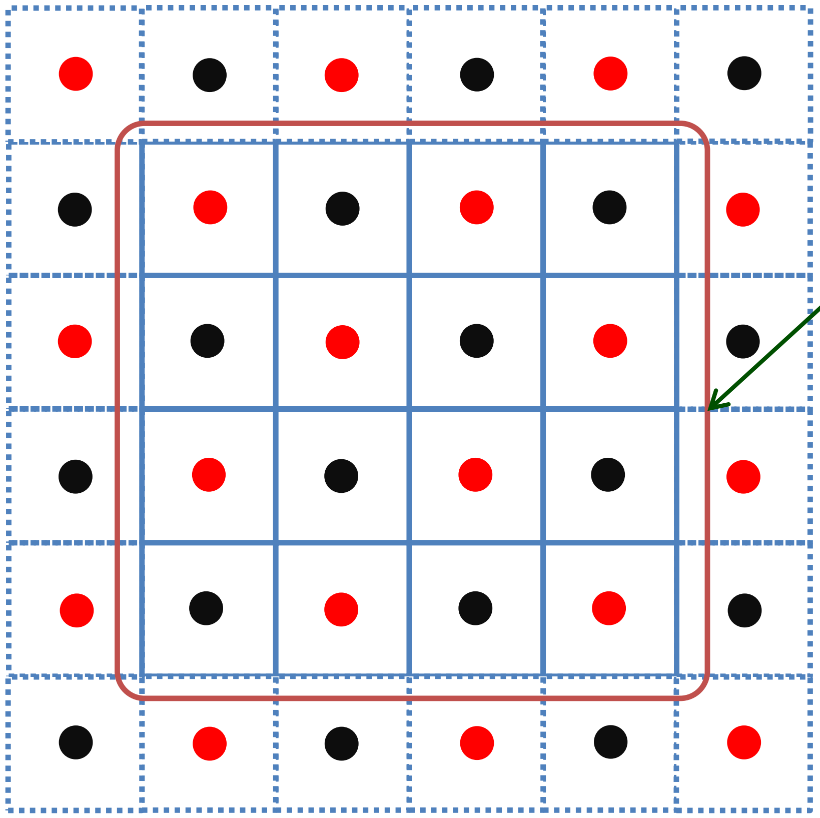
Communication process design

- 4. Update the south ghost cell from south neighboring domain



Communication process design

- **Now, all domain has updated ghost cells**
  - A single step of RB-GS is calculated except ghost cells
  - We don't need the calculation of ghost cells
  - Strictly, we need to know the values of every other cells



Single step of red-cell update

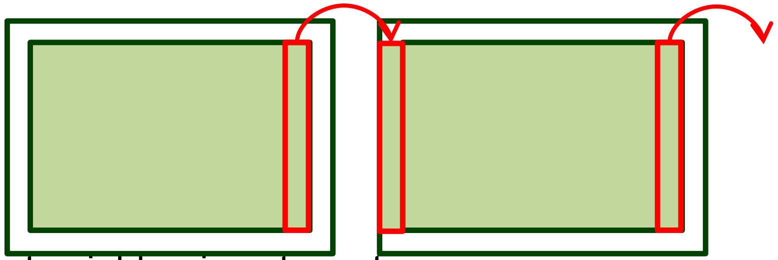
For red-cell grid update, we require black-cell only

Designing communication subroutines

Send east neighboring domain and update the west ghost cell

1. Prepare the east-edge data to send
➔ packing the data to 1-d array

```
for(j=1;j<=ny_mpi;j++)
      sendbuf[j-1] = [nx_mpi*(ny_mpi+2)+j];
```



2. Send the packed data to east neighbor when east neighbor is not empty

```
if(mpi_info->e_rank >= 0)  {
   for(j=1;j<=ny_mpi;j++)
      sendbuf[j-1] = [nx_mpi*(ny_mpi+2)+j];
   MPI_Isend(sendbuf,ny_mpi,MPI_DOUBLE,mpi_info->e_rank,101,MPI_COMM_WORLD,&req1);
}
```

3. Receive the packed data from west neighbor when west neighbor is not empty

```
if(mpi_info->w_rank >= 0)  {
   MPI_Irecv(recvbuf,ny_mpi,MPI_DOUBLE,mpi_info->w_rank,101,MPI_COMM_WORLD,&req2);
}
```

4. Waiting for the end of communication

```
if(mpi_info->e_rank >= 0)
   MPI_Wait(&req1,&status1);
if(mpi_info->w_rank >= 0)
   MPI_Wait(&req2,&status2);
```

5. Restore the ghost-cell from the transferred data

```
if(mpi_info->w_rank >= 0)  {
   MPI_Wait(&req2,&status2);
   for(j=1;j<=ny_mpi;j++)
      u[0*(ny_mpi+2)+j] = recvbuf[j-1];
}
```

Example of sending north domain

- We don’t need check whether the north domain is empty of not

```
void send_north(double *u, int nx_mpi, int ny_mpi, MYMPI *mpi_info)
{
    int i;
    double *sendbuf, *recvbuf;
    MPI_Request req1, req2;
    MPI_Status status1, status2;
    sendbuf = (double*)malloc(nx_mpi*sizeof(double));
    recvbuf = (double*)malloc(nx_mpi*sizeof(double));

    for(i=1;i<=nx_mpi;i++) {
        sendbuf[i-1] = u[i*(ny_mpi+2)+ny_mpi];
    }

    MPI_Isend(sendbuf,nx_mpi,MPI_DOUBLE,mpi_info->n_rank,103,MPI_COMM_WORLD,&req1);
    MPI_Irecv(recvbuf,nx_mpi,MPI_DOUBLE,mpi_info->s_rank,103,MPI_COMM_WORLD,&req2);
    MPI_Wait(&req1,&status1);
    MPI_Wait(&req2,&status2);

    for(i=1;i<=nx_mpi;i++)    {
        u[i*(ny_mpi+2)+0] = recvbuf[i-1];
    }

    free(sendbuf);
    free(recvbuf);
}
```

Header file design

```
typedef struct mympi {
                int nprocs;
                int myrank;
                int nx_mpi;
                int ny_mpi;
                int mpisize_x;
                int mpisize_y;
                int mpirank_x;
                int mpirank_y;
                int w_rank;
                int e_rank;
                int n_rank;
                int s_rank;
} MYMPI;

void mpi_setup(int nx, int ny, MYMPI *mpi_info);
void send_east(double *u, int nx_mpi, int ny_mpi, MYMPI *mpi_info);
void send_west(double *u, int nx_mpi, int ny_mpi, MYMPI *mpi_info);
void send_north(double *u, int nx_mpi, int ny_mpi, MYMPI *mpi_info);
void send_south(double *u, int nx_mpi, int ny_mpi, MYMPI *mpi_info);
```

5. Write or modify your code for single domain (I)

- **REMEMBER: Each process will run the same code to update its domain!**
  - Adjust array dimensions to decomposed domain size. e.g.:

```
grid_size = (nx+2) * (ny+2);
grid_size_mpi = (mpi_info.nx_mpi + 2) * (mpi_info.ny_mpi + 2);

pos_x = (double*)malloc((mpi_info.nx_mpi+2)*sizeof(double));
pos_y = (double*)malloc((mpi_info.ny_mpi+2)*sizeof(double));
u_solve = (double*)malloc(grid_size_mpi*sizeof(double));
rhs = (double*)malloc(grid_size_mpi*sizeof(double));
```

  - Code explicitly for specific blocks (i.e. processes) where necessary. e.g.

```
for(i=0;i<=mpi_info.nx_mpi+1;i++)
    pos_x[i]=(i-0.5 + mpi_info.mpirank_x*mpi_info.nx_mpi)*dx;
for(j=0;j<=mpi_info.ny_mpi+1;j++)
    pos_y[j]=(j-0.5 + mpi_info.mpirank_y*mpi_info.ny_mpi)*dy;
for(i=1;i<=mpi_info.nx_mpi;i++) {
    x_val = pos_x[i]*(1.0-pos_x[i]);
    for(j=1;j<=mpi_info.ny_mpi;j++) {
        y_val = cos(2.0*PI*pos_y[j]);
        u_solve[i*(mpi_info.ny_mpi+2)+j] = 0.0;
        rhs[i*(mpi_info.ny_mpi+2)+j] = -2.0*y_val-4.0*PI*PI*x_val*y_val;
    }
}
```

➔ We can maintain x_pos and y_pos without decomposition because its dimension is lower than main variable

➔ We need to apply domain decomposition for variables in main equations with the size of grids

5. Write or modify your code for single domain (II)

- In RB-GS solver subroutine

```
for(walk=1;walk<3;walk++)           {
…
   send_east(u_solve, nx, ny, mpi_info);
   send_west(u_solve, nx, ny, mpi_info);
   send_north(u_solve, nx, ny, mpi_info);
   send_south(u_solve, nx, ny, mpi_info);
   js = 3 - walk;
   for(i=1;i<=nx;i++) {
      js=3-js;
      for(j=js;j<=ny;j+=2) {
         …
```

Decomposed domain

6. Adjust boundary condition

- **We can decide boundary domain from whether neighbor is empty or not**
  - If west neighbor is empty, it is left boundary domain
  - If east neighbor is empty, it is right boundary domain

```
if(mpi_info->w_rank < 0 ) {
       for(j=1;j<=ny;j++) {
           u_solve[0*(ny+2)+j]=-u_solve[1*(ny+2)+j];
       }
   }
   if(mpi_info->e_rank < 0 ) {
       for(j=1;j<=ny;j++) {
           u_solve[(nx+1)*(ny+2)+j]=-u_solve[nx*(ny+2)+j];
       }
   }
```

Another issues - convergence check

- Error_sum and u_sum is calculated in each MPI process
- We should sum error_sum and u_sum from all MPI process

```
error_sum += fabs(uij_new-uij_old);
u_sum += fabs(uij_new);
```

```
if(error_sum_global/u_sum_global<tolerance) break;
```

⇩

```
MPI_Allreduce(&error_sum,&error_sum_global,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
MPI_Allreduce(&u_sum,&u_sum_global,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);

if(error_sum_global/u_sum_global<tolerance) break;
```

MPI Timing

- **MPI Wtime() function returns the wall clock time**
  - double s_time, e_time, t_time
  - MPI_Barrier(MPI_COMM_WORLD)
  - s_time = MPI_Wtime()
  - ..
  - ..
  - MPI_Barrier(MPI_COMM_WORLD)
  - e_time = MPI_Wtime()
  - t_time = e_time - s_time