Introduction to Parallel Programming with MPI

Lecture #6: *Solution of 2D Laplace Equation*

Andrea Mignone[1]

[1]Dipartimento di Fisica- Turin University, Torino (TO), Italy

Laplace Equation

- We now wish to solve the Laplace equation on a 2D Cartesian domain **Ω**:

$$\nabla^2\varphi = \frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} = 0 \qquad \text{with} \qquad \varphi|_{\partial\Omega} = f(x,y)$$

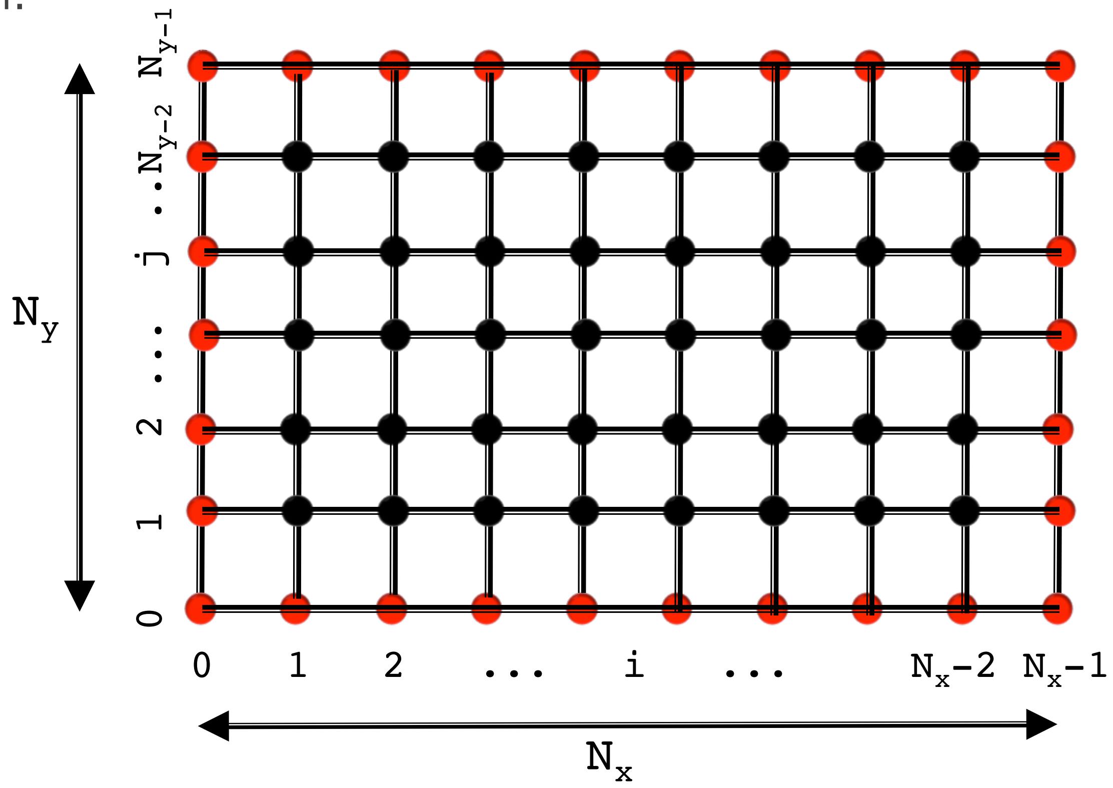
where ***f(x,y)*** is a prescribed function on the boundary of **Ω.**

- The Laplace equation is found in many area of physics, such as fluid dynamics and electrostatic.

- The Laplace equation is an elliptic partial differential equation and its solution depends solely on the boundary values.

Elliptic PDE: Discretization

- We define a 2D lattice of N_x points in the x-direction and N_y points in the y-direction:



- Uniform and equal spacing in both direction is assumed: $h=\Delta x=\Delta y$.
- *Red* points should be specified as boundary conditions while *black* points are the solution values (unknowns).

Elliptic PDE: Discretization

- To begin with, we discretize the Laplacian operator using 2nd-order approximations to the second derivatives:

$$\frac{\varphi_{i+1,j} - 2\varphi_{i,j} + \varphi_{i-1,j}}{\Delta x^2} + \frac{\varphi_{i,j+1} - 2\varphi_{i,j} + \varphi_{i,j-1}}{\Delta y^2} = 0$$

- *Interior points*:
  - $i=1 \ldots N_x-2,\ j=1 \ldots N_y-2$. This is where the solution must be found.

- *Boundary points*:
  - *Bottom:* $i=0 \ldots N_x-1$ $j=0$
  - *Top:* $i=0 \ldots N_x-1$ $j=N_y-1$
  - *Left:* $i=0$ $j=0 \ldots N_y-1$
  - *Right:* $i=N_x-1$ $j=0 \ldots N_y-1$

Jacobi’s Iterative Method

- Suppose we have found a solution of the discretized equation, then at each grid point:

$$\varphi_{i,j} = \frac{1}{4}\left(\varphi_{i+1,j} + \varphi_{i-1,j} + \varphi_{i,j+1} + \varphi_{i,j-1}\right)$$

- This is only formal since the r.h.s. is not known. To find the solution, the equations must be solved simultaneously → solving Poisson's equation is essentially a problem in linear algebra.

- Jacobi's iterative method starts with a guess $\phi^{(0)}$ for the solution at the interior lattice points. Plugging this guess into the r.h.s. yields $\phi^{(1)}$ at all lattice points. Iterating:

$$\boxed{\varphi_{i,j}^{(k+1)} = \frac{1}{4}\left(\varphi_{i+1,j}^{(k)} + \varphi_{i-1,j}^{(k)} + \varphi_{i,j+1}^{(k)} + \varphi_{i,j-1}^{(k)}\right)}$$

- The computation of $\phi^{(k+1)}$ requires neighbor elements at the previous stage: cannot overwrite $\phi^{(k)}$ with $\phi^{(k+1)}$ since that value will be needed by the rest of the computation. Jacobi’s method requires *two* arrays of size **`nxn`**.

Boundary conditions & Convergence Checking

- For simplicity we will only use Dirichlet boundary conditions which require the value of the solution to be known on the four boundary sides:

$$\begin{cases} \varphi(x_{\rm beg}, y) = g_0(y) & \to \quad \varphi^{(k+1)}_{i_{\rm beg},j} = g_0(y_j) & \text{(left)} \\ \varphi(x_{\rm end}, y) = g_1(y) & \to \quad \varphi^{(k+1)}_{i_{\rm end},j} = g_1(y_j) & \text{(right)} \\ \varphi(x, y_{\rm beg}) = f_0(x) & \to \quad \varphi^{(k+1)}_{i,j_{\rm beg}} = f_0(x_i) & \text{(bottom)} \\ \varphi(x, y_{\rm end}) = f_1(x) & \to \quad \varphi^{(k+1)}_{i,j_{\rm end}} = f_1(x_i) & \text{(top)} \end{cases}$$

- Convergence is reached when the relative difference between two successsive iterations falls below some prescribed tolerance

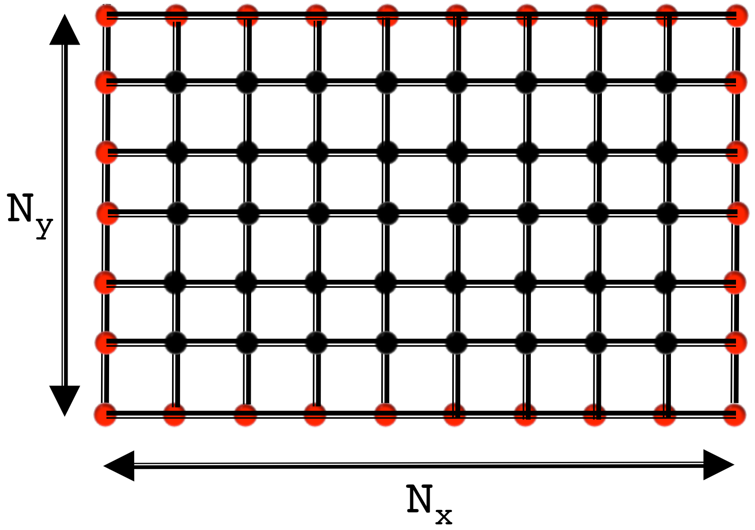
$$\epsilon = \sum_{ij} \left| \varphi^{(k+1)}_{ij} - \varphi^{(k)}_{ij} \right| \Delta x \Delta y$$

where summation should be extended to *interior points only*.

Algorithm Implementation: serial code

- Here’s a sketch on how your code should be correctly written:

```
-   define grid arrays x[i] and y[j];

-   allocate memory for 2D solution array;

-   initialize solution array (e.g. $\varphi^0$[i][j] = 0) in the interior points;

-   Start iterating (unitil res < tol)
       -   Assign boundary conditions
       -   Update 2D solution;
       -   Compute residual;

-   Write solution to disk;
```



Note: interior points are in black, and looping over them can be done using the indices

```
ibeg    = NGHOST;
iend    = ibeg + nx - 1;
```

and similarly for `jbeg`, `jend`.
Boundary points are in red and corresponds to

```
-   φ[0][j], φ[NX-1][j] at left, right bound.;
-   φ[*][0], φ[*][NY-1]  at bottom, top bound.;
```

Problem Details

- Find the steady-state temperature distribution of a rectangular plate $0 \le x \le 1$, $0 \le y \le 1$, subject to the following Dirichlet boundary conditions:

$$\begin{cases} \varphi(0,y) = 1 - y \\ \varphi(1,y) = y^2 \\ \varphi(x,0) = 1 - x \\ \varphi(x,1) = x \end{cases}$$

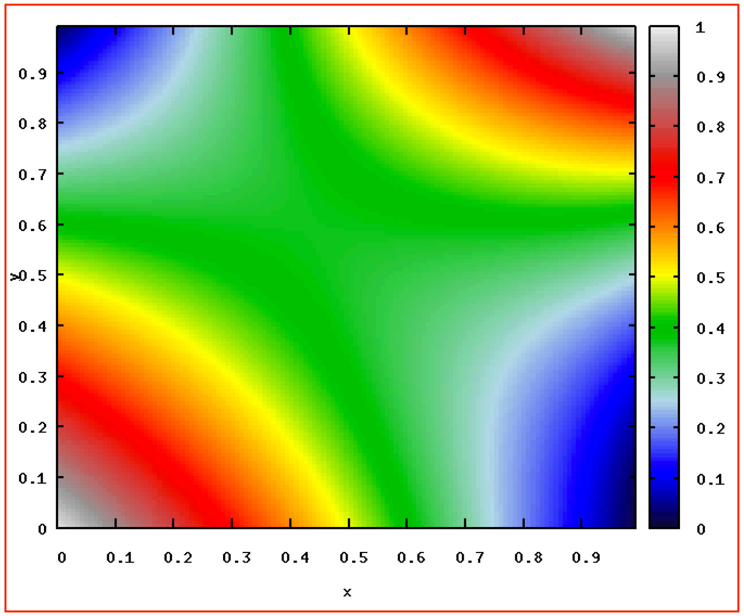
- Use 128 x 128 grid nodes and compute the residual through

$$\epsilon = \sum_{ij} \left| \varphi_{ij}^{(k+1)} - \varphi_{ij}^{(k)} \right| \Delta x \Delta y$$

- Quit iteration loop when $\varepsilon < 10^{-5}$.

- The solution is shown in the right panel and convergence should be achieved in ≈ 7316 iterations.

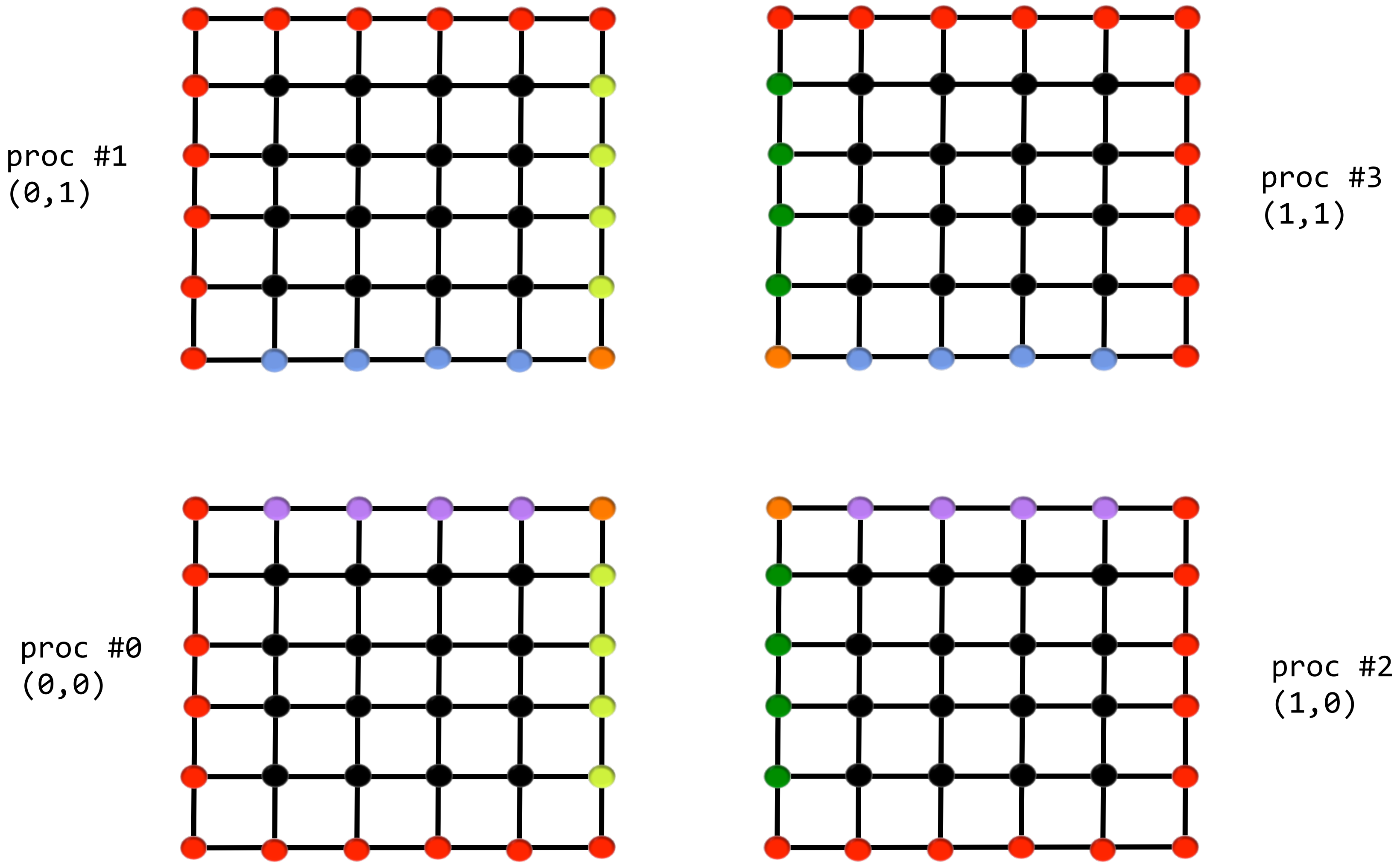
- If you're using Gnuplot, the script `laplace2D.gp` can be used to produce this figure.



Parallel Implementation

Parallel Domain Decomposition

- In parallel, the computational domain is divided into (equally sized) sub-domains using a Cartesian decomposition with `MPI_Cart_create()`.



Parallel Domain Decomposition

- Domain decomposition should be done through the `MPI_Cart_create()` function.

- For efficiency purpose, it is best to define a simple C structure holding all the relevant information:

```c
typedef struct MPI_Decomp_{
  int nprocs[NDIM];     /*  Number of processes in each dimension */
  int periods[NDIM];    /*  Periodicity flag in each dimension     */
  int coords[NDIM];     /*  Cartesian coordinate in the MPI topology */
  int gsize[NDIM];      /*  Global domain size (no ghosts)  */
  int lsize[NDIM];      /*  Local domain size (no ghosts)   */
  int start[NDIM];      /*  Local start index in each dimension           */
  int procL[NDIM];      /*  Rank of left-lying  process in each direction */
  int procR[NDIM];      /*  Rank of right-lying process in each direction */
  int rank;             /*  Local process rank */
  int size;             /*  Communicator size  */
} MPI_Decomp;
```

- This structure can be passed through functions, e.g.

```c
int main()
{
  MPI_Decomp mpi_decomp;
  ...
  DomainDecomposition (&mpi_decomp);
  ...
  BoundaryConditions (&mpi_decomp);
  ...
}
```

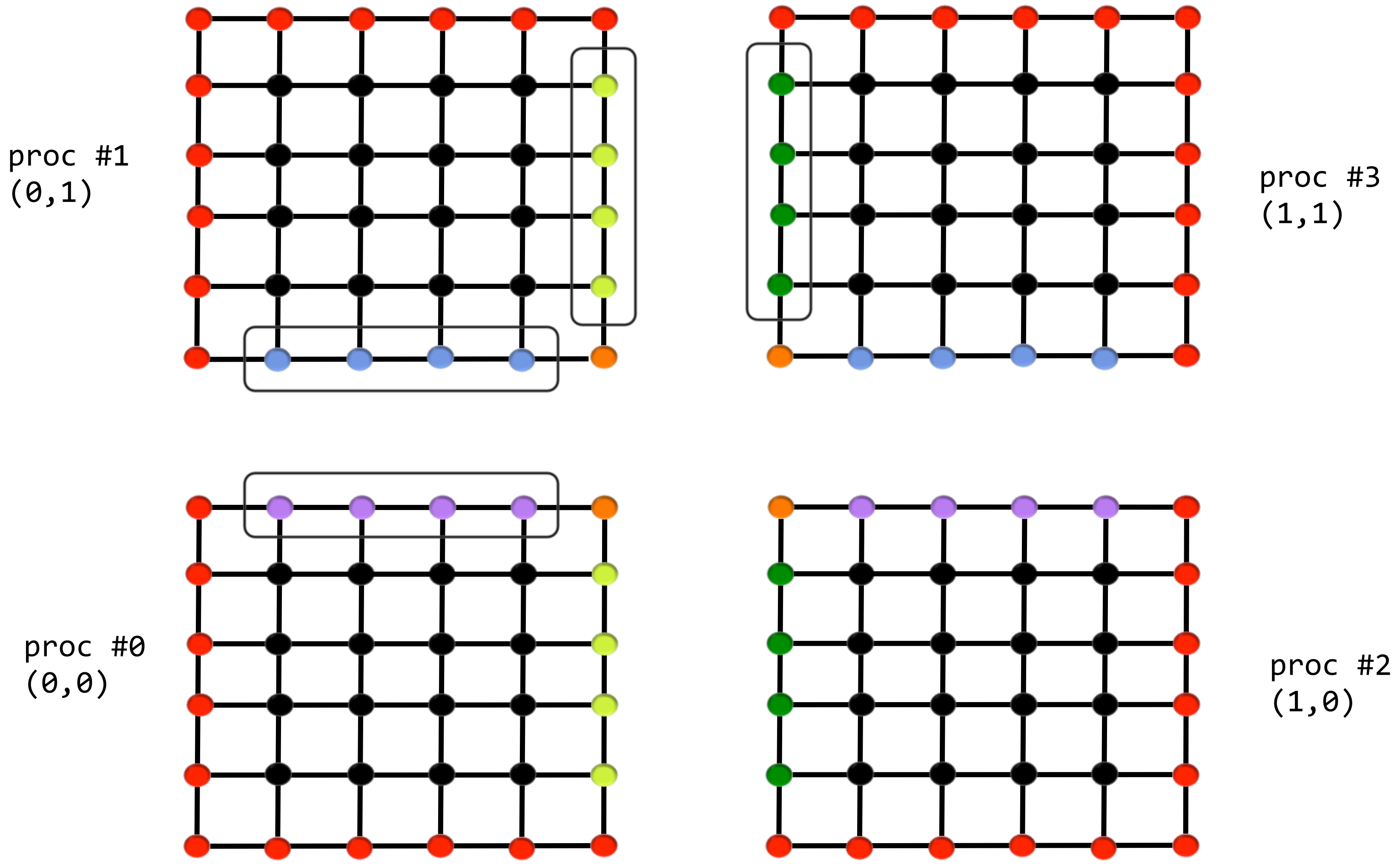
Parallel Domain Decomposition

- The `DomainDecomposition()` function should fill the structure:

```
void DomainDecomposition(MPI_Decomp *mpi_decomp)
{
  // 1. Get rank & size

  // 2. Determine the number of processes in each dimension
  //   (use maximally squared decomp), disable periodicity

  // 3. Use MPI_Cart_create() and MPI_Cart_get() to obtain
  //    the Cartesian coordinates for the current process.

  // 4. Fill structure members.

  // 5. Determine the ranks procL[] and procR[] of the neigbour processes
  //    in each direction. Use MPI_PROC_NULL for physical boundaries.

  // 6. Print relevant information (optional but useful).
}
```

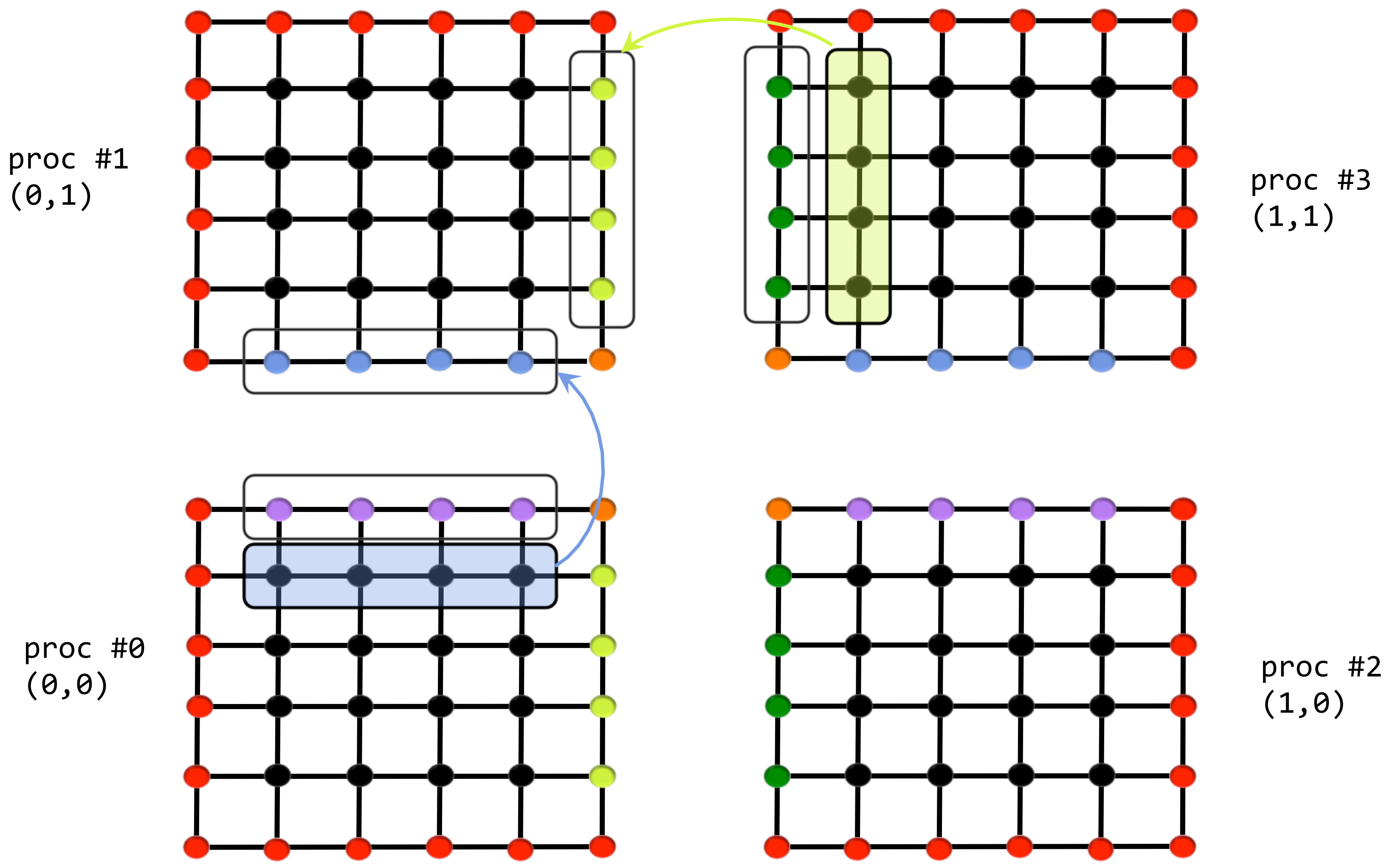
Boundary Conditions in Parallel

- Red points = physical boundary conditions. Inter-processor b.c. are marked with a box. The values here must be exchanged with neighbor processes.



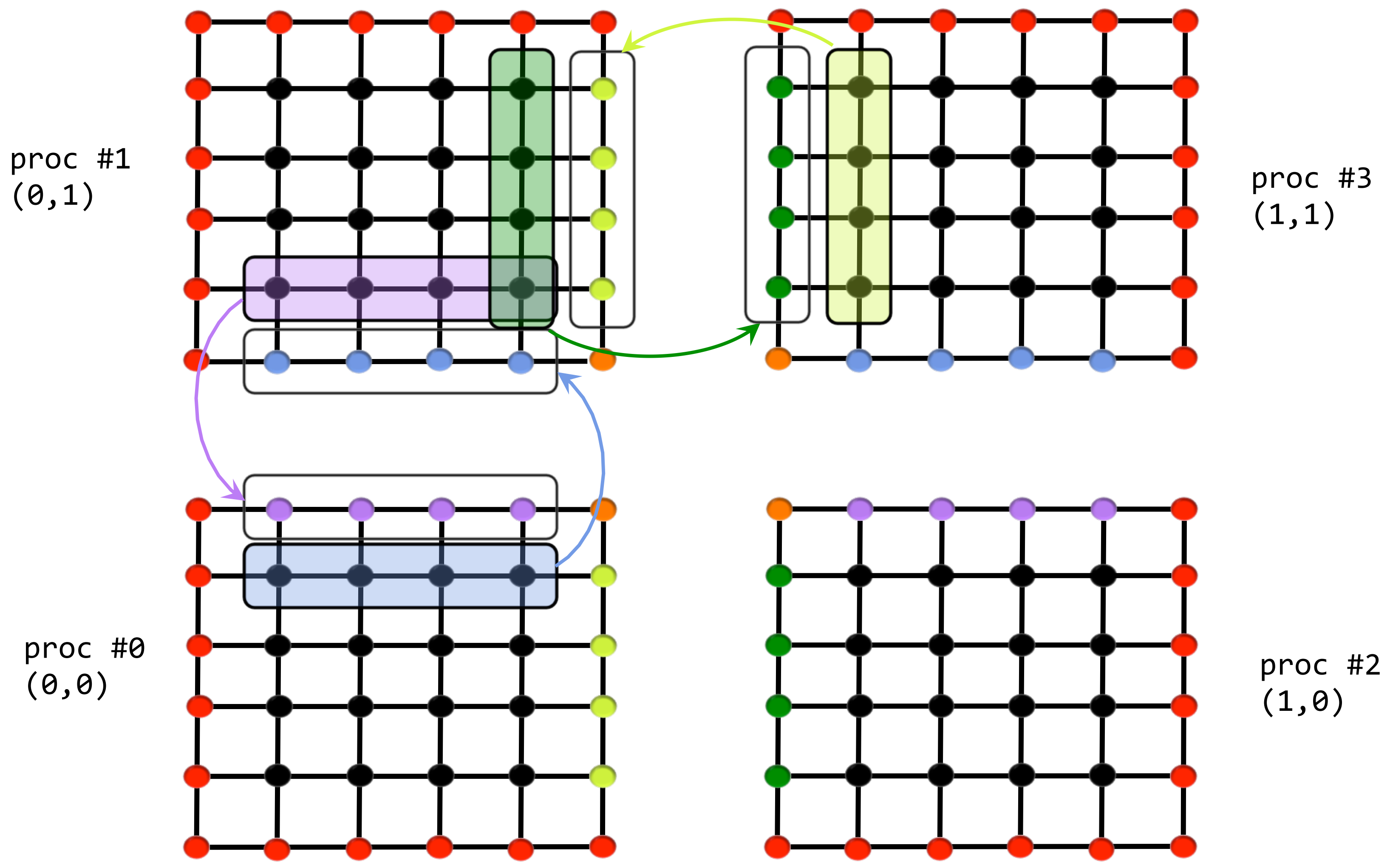
Boundary Conditions in Parallel

- Inter-processor b.c. must be exchanged using `MPI_Send/Recv()` functions (we focus on `proc #1` only).



Boundary Conditions in Parallel

- Inter-processor b.c. must be exchanged using `MPI_Send/Recv()` functions.



Parallel Algorithm:

- We can now modify the serial algorithm in the following way:

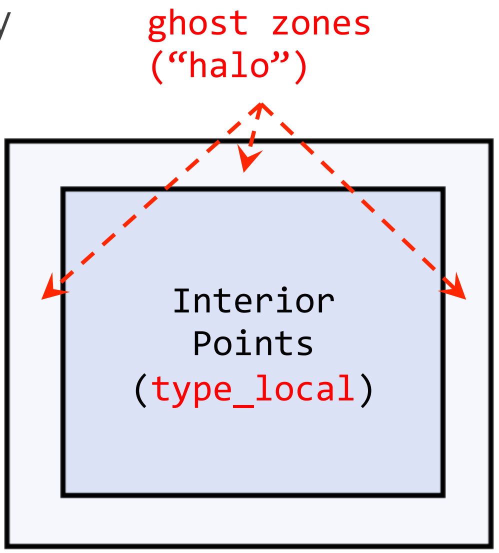
```
-   [Parallel: define a DomainDecomposition() function that does the domain to
    obtain a Cartesian decomposition]

-   define grid arrays x[i] and y[j];
    [Parallel: each process owns the global grid (xg[] and yg[]), but local grid
    should also be defined → use mpi_decomp->start[] for providing offsets]

-   allocate memory for 2D solution array;
    [Parallel: memory allocation for 2D array should be done on local domain
    with the addition of guard cells]

-   initialize solution array (e.g. $\varphi^0$[i][j] = 0) in the interior points;

-   Start iterating (unitil res < tol)
      -   Assign boundary conditions through BoundaryConditions()
          [Parallel: distinguish between physical and inter-proc b.c.]
      -   Update 2D solution;
      -   Compute residual;
          [Parallel: apply reduce operation]

-   Write solution to disk;
```

Writing Files: defining the local array type

- Local arrays are surrounded by a "halo" of ghost zones, but only interior points must be written.

- Need to create a subarray datatype to describe the noncontiguous layout in memory (φ[][] shorn of ghost points) with `MPI_Type_create_subarray()`:

```
void WriteSolution(..., MPI_Decomp *md)
{
...
  // 1. Define the local datatype
  MPI_Datatype type_local;

  gsize[0] = md->lsize[0] + 2*NGHOST; // Local array size including
  gsize[1] = md->lsize[1] + 2*NGHOST; // ghost points
  lsize[0] = md->lsize[0];  // Size of subarray is
  lsize[1] = md->lsize[1];  // local domain size
  start[0] = NGHOST;
  start[1] = NGHOST;

  MPI_Type_create_subarray (NDIM, gsize, lsize, start,
                            MPI_ORDER_FORTRAN, MPI_DOUBLE, &type_local);
  MPI_Type_commit (&type_local);
...
}
```

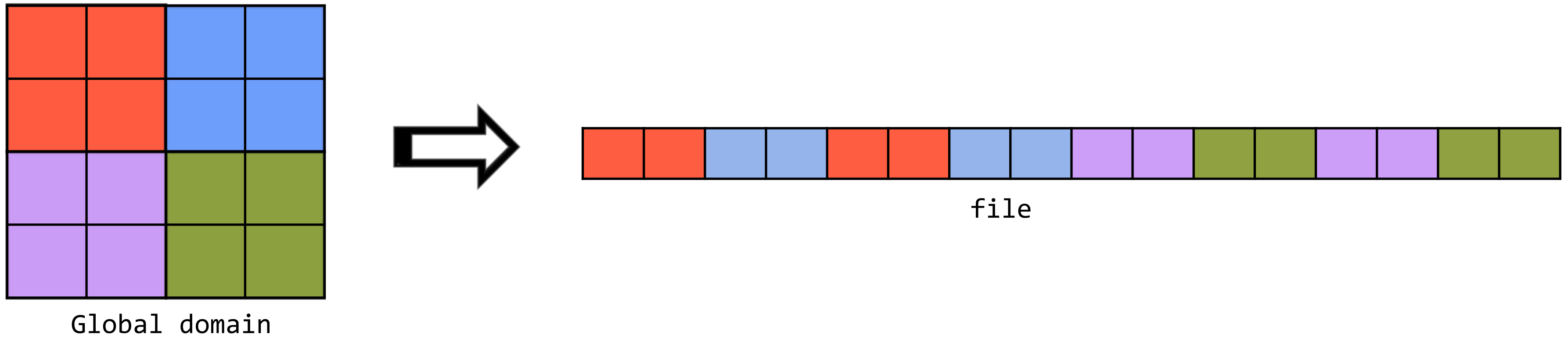


- We will use this as arguments to `MPI_File_write()`.

Writing Files: defining the file view

- The file view must be set by creating a second subarray datatype, defining the process' view on the file:

```
void WriteSolution(..., MPI_Decomp *md)
{
...
  // 2. Define the domain datatype
  MPI_Datatype type_domain;
  gsize[0] = NX_GLOB;       // Global size (entire file)
  gsize[1] = NY_GLOB;
  lsize[0] = md->lsize[0];  // Local size (amount of data accessible by proc)
  lsize[1] = md->lsize[1];

  start[0] = lsize[0]*md->coords[0];  // Starting indices (in grid points)
  start[1] = lsize[1]*md->coords[1];  // for local processor

  MPI_Type_create_subarray (NDIM, gsize, lsize, start,
                            MPI_ORDER_FORTRAN, MPI_DOUBLE, &type_domain);
  MPI_Type_commit (&type_domain);
}
```



Writing Files: putting all together

- Now we can put all together and open file file for writing:

```
void WriteSolution(..., MPI_Decomp *md)
{
...
  // 3. Open file for writing

  MPI_File_delete(fname, MPI_INFO_NULL);

  MPI_File_open(MPI_COMM_CART, fname, amode, MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh, 0, MPI_DOUBLE, type_domain, "native", MPI_INFO_NULL);
  MPI_File_write_all(fh, phi[0], 1, type_local, MPI_STATUS_IGNORE);
  MPI_File_close(&fh);
...
}
```

THE END