# Introduction to High Performance Computing

\\

SDS406 – Fall semester, 2024 - 2025

\\

L09: Parallelizing PDEs with MPI, $25^{\text{th}}$ November 2024

# Numerical solution of PDEs

Partial Differential Equations (PDEs) arise naturally, e.g. describing time-evolution in physical systems

- Consider the PDE:

$$\partial_t u(t, \vec{r}) = \kappa \nabla^2_{\vec{r}} u(t, \vec{r})$$

# Numerical solution of PDEs

Partial Differential Equations (PDEs) arise naturally, e.g. describing time-evolution in physical systems

- Consider the PDE:

$$\partial_t u(t, \vec{r}) = \kappa \nabla^2_{\vec{r}} u(t, \vec{r})$$

- $u(t, \vec{r})$ can be a field of temperatures at position $\vec{r}$ and time $t$. The above PDE describes the time-evolution of $u(t, \vec{r})$, given:
  - An *initial condition*, $u(0, \vec{r})$
  - A set of *boundary conditions*, e.g.:
    - $u(t, \vec{r}) = 0$ for $r^2 > R^2$ (zero if outside of a sphere with radius $R$);
    - $u(t, \vec{r} + \hat{e}_i L) = u(t, \vec{r})$ for $i = 1, 2, 3$ (periodic within box of side $L$).

# Numerical solution of PDEs

Numerical solution of PDEs requires first writing the PDE in discrete form

- We will restrict to the 2-dimensional PDE (2 spatial dimensions, $x$ and $y$, plus time, $t$):

$$\partial_t u(t,x,y) = \kappa[\partial_x^2 u(t,x,y) + \partial_y^2 u(t,x,y)]$$

- One choice of discretizing the derivative

$$f(x+\delta) = f(x) + \frac{\partial f(x)}{\partial x}\delta + \frac{\partial^2 f(x)}{\partial x^2}\frac{\delta^2}{2} + O(\delta^3) \Rightarrow$$

$$\frac{\partial f(x)}{\partial x} = \frac{f(x+\delta) - f(x)}{\delta} + O(\delta)$$

# Numerical solution of PDEs

- For second derivative, take the central difference:

$$f(x+\delta) = f(x) + \frac{\partial f(x)}{\partial x}\delta + \frac{\partial^2 f(x)}{\partial x^2}\frac{\delta^2}{2} + O(\delta^3)$$

$$f(x-\delta) = f(x) - \frac{\partial f(x)}{\partial x}\delta + \frac{\partial^2 f(x)}{\partial x^2}\frac{\delta^2}{2} + O(\delta^3)$$

- Add both sides:

$$\frac{\partial^2 f(x)}{\partial x^2} \simeq \frac{f(x+\delta) - 2f(x) + f(x-\delta)}{\delta^2}$$

- Back to the heat equation:

$$\partial_t u(t,x,y) = \kappa[\partial_x^2 u(t,x,y) + \partial_y^2 u(t,x,y)] \Rightarrow$$

$$\frac{u(t+\tau,x,y) - u(t,x,y)}{\tau} =$$

$$\frac{\kappa}{h^2}[u(t,x+h,y) + u(t,x-h,y) + u(t,x,y+h) + u(t,x,y-h) - 4u(t,x,y)]$$

# Numerical solution of PDEs

This allows us to write the heat equation *iteration*:

$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)]$$

where $\alpha = \frac{\tau\kappa}{h^2}$ and $H$, the *hopping term*:

$$H[u(t, x, y)] = u(t, x+h, y) + u(t, x-h, y) + u(t, x, y+h) + u(t, x, y-h).$$

# Numerical solution of PDEs

This allows us to write the heat equation iteration:

$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)]$$

where $\alpha = \frac{\tau\kappa}{h^2}$ and the hopping term:

$$H[u(t, x, y)] = u(t, x + h, y) + u(t, x - h, y) + u(t, x, y + h) + u(t, x, y - h).$$

- Note that stability analysis requires:

  $$\alpha < 0.25$$

  for stability of the iterations.

# Numerical solution of PDEs

## Serial code for heat equation

- Copy the directory:

  ```
  /onyx/data/sds406f24/l09/ex01/
  ```

  to your home directory.

- Included is a C source code file and a python script
- Study the C file, complete the missing parts, and run it using:

  ```
  [user@front02 l09]$ srun -n 1 -p p100 ./heat 256 6000 0.2
  ```

# Numerical solution of PDEs

## Serial code for heat equation

- Copy the directory:

  ```
  /onyx/data/sds406f24/l09/ex01/
  ```

  to your home directory.

- Included is a C source code file and a python script
- Study the C file, complete the missing parts, and run it using:

  ```
  [user@front02 l09]$ srun -n 1 -p p100 ./heat 256 6000 0.2
  ```

- Arguments are:
  - `L`, the spatial extent,
  - `T`, the number of time iterations, and
  - `alpha`, defined as in the previous slide

# Numerical solution of PDEs

## Serial code for heat equation; Some implementation details

### Indexing and 2-dimensional arrays

- The 2-dimensional array that holds $u(x, y)$ is stored as 1-dimensional array in memory, e.g.: `v[M][N]` $\longrightarrow$ `v[M*N]`

# Numerical solution of PDEs

## Serial code for heat equation; Some implementation details

### Indexing and 2-dimensional arrays

- The 2-dimensional array that holds $u(x, y)$ is stored as 1-dimensional array in memory, e.g.: `v[M][N]` $\longrightarrow$ `v[M*N]`
- We have decided the order of the indices to be such that *x runs fastest*:

`v[y][x]` $\longrightarrow$ `v[y*L + x]`

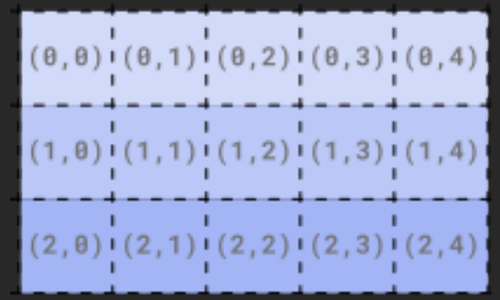
# Numerical solution of PDEs

## Serial code for heat equation; Some implementation details

### Indexing and 2-dimensional arrays

- The 2-dimensional array that holds $u(x, y)$ is stored as 1-dimensional array in memory, e.g.: `v[M][N]` $\longrightarrow$ `v[M*N]`
- We have decided the order of the indices to be such that *x runs fastest*:

`v[y][x]` $\longrightarrow$ `v[y*L + x]`



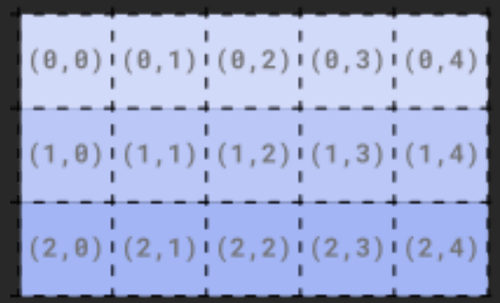

# Numerical solution of PDEs

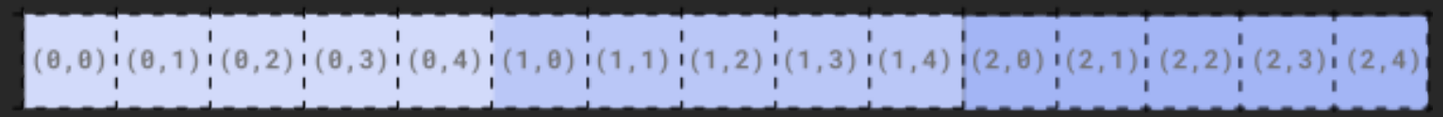
## Serial code for heat equation; Some implementation details

### Indexing and 2-dimensional arrays

- The 2-dimensional array that holds $u(x, y)$ is stored as 1-dimensional array in memory, e.g.: `v[M][N]` $\longrightarrow$ `v[M*N]`
- We have decided the order of the indices to be such that *x runs fastest*:

`v[y][x]` $\longrightarrow$ `v[y*L + x]`




- The C *macro* `IDX()` saves us some typing:

```
#define IDX(y, x) ((L+(y))%L)*L + ((L+(x))%L)
```

Not a function! Macros are expanded at *compile time*

# Numerical solution of PDEs

## Serial code for heat equation; Some implementation details

### Boundary and initial conditions

1. All points with `x == 0` and `x == L-1` are set to 1
2. All points on `y == L/2` are set to 1
3. All other points are zero
4. Conditions 1. and 2. are enforced *in every iteration*

# Numerical solution of PDEs

## Serial code for heat equation; Some implementation details

### Boundary and initial conditions

1. All points with `x == 0` and `x == L-1` are set to 1
2. All points on `y == L/2` are set to 1
3. All other points are zero
4. Conditions 1. and 2. are enforced *in every iteration*

```
/***
 * Set the boundary condition for v[L*L]
 ***/
void
boundary_condition(double *v)
{
  for(int y=0; y<L; y++) {
    v[IDX(y,   0)] = 1;
    v[IDX(y, L-1)] = 1;
  }
  for(int x=0; x<L; x++) {
    v[IDX(L/2, x)] = 1;
  }
  return;
}
...
  for(int i=0; i<T; i++) {
    /* Apply one time-step */
    update(v0, alpha);

    /* Update boundaries */
    boundary_condition(v0);
  }
```

# Numerical solution of PDEs

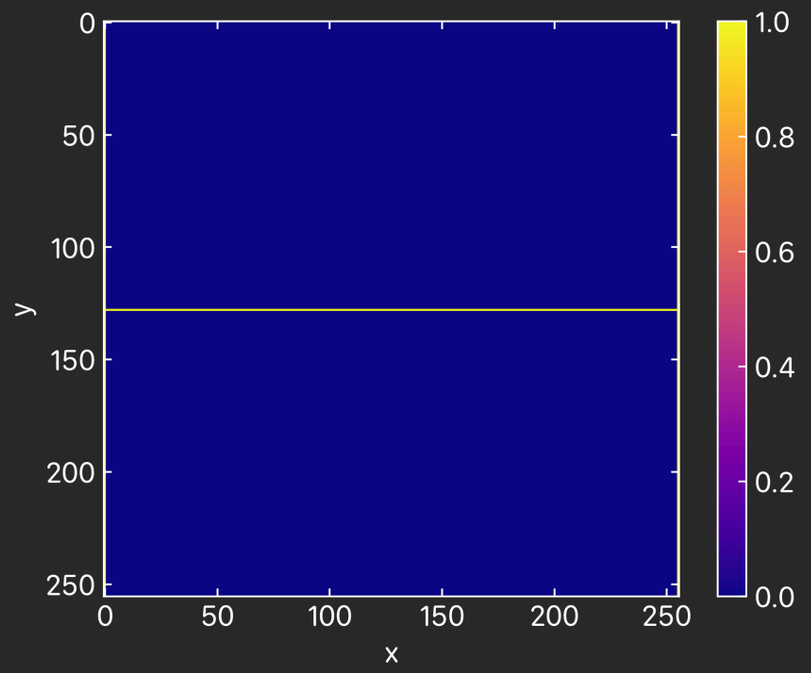
## Serial code for heat equation; Some implementation details

### Boundary and initial conditions

1. All points with `x == 0` and `x == L-1` are set to 1
2. All points on `y == L/2` are set to 1
3. All other points are zero
4. Conditions 1. and 2. are enforced *in every iteration*

```c
/***
 * Set the boundary condition for v[L*L]
 ***/
void
boundary_condition(double *v)
{
  for(int y=0; y<L; y++) {
    v[IDX(y,   0)] = 1;
    v[IDX(y, L-1)] = 1;
  }
  for(int x=0; x<L; x++) {
    v[IDX(L/2, x)] = 1;
  }
  return;
}
...
  for(int i=0; i<T; i++) {
    /* Apply one time-step */
    update(v0, alpha);

    /* Update boundaries */
    boundary_condition(v0);
  }
```



# Numerical solution of PDEs

## Serial code for heat equation

- Copy the directory:

  ```
  /onyx/data/sds406f24/l09/ex01/
  ```

  to your home directory.

- Included is a C source code file and a python script
- Study the C file, complete the missing parts, and run it using:

  ```
  [user@front02 l09]$ srun -n 1 -p p100 ./heat 256 6000 0.2
  ```

- The final distribution is stored in `v0.txt`
- You can plot `v0.txt` using the python file `plot.py`

# Numerical solution of PDEs

## Serial code for heat equation

- Copy the directory:

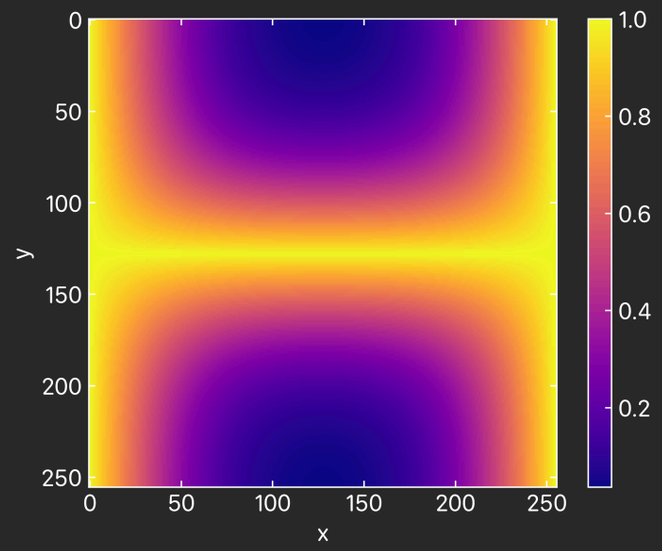
  ```
  /onyx/data/sds406f24/l09/ex01/
  ```

  to your home directory.

- Included is a C source code file and a python script
- Study the C file, complete the missing parts, and run it using:

  ```
  [user@front02 l09]$ srun -n 1 -p p100 ./heat 256 6000 0.2
  ```

- The final distribution is stored in `v0.txt`
- You can plot `v0.txt` using the python file `plot.py`



# Parallelization using MPI

## Domain Decomposition of Partial Differential Equations (PDEs) using MPI

- Parallelizing the 2-dimensional heat equation with MPI
  - Parallelizing the slow-running coordinate
  - Domain decomposition
  - Boundary exchange
  - Boundary conditions

# Domain decomposition of PDEs

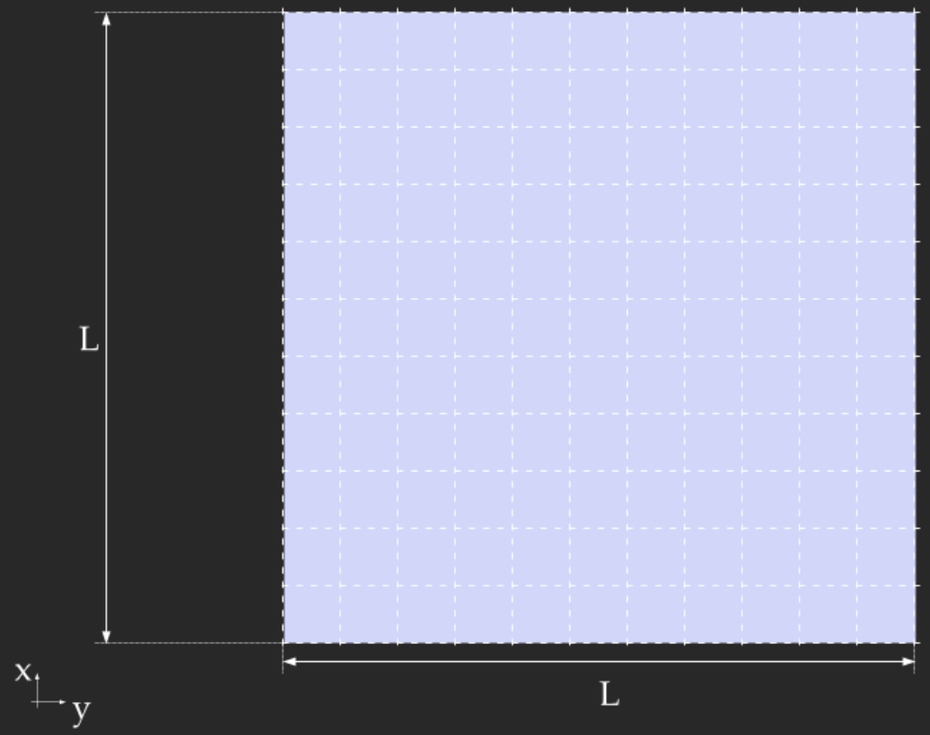
- *Domain decomposition* is referred to when we parallelize a problem by distributing the domain over processes
- In the case of the heat equation, we begin with an initial condition $u(t = 0, x, y)$ and every iteration depends on the previous time $\Rightarrow$ cannot parallelize in time coordinate
- However we *can* parallelize over the spatial domain

# Domain decomposition of PDEs

$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$

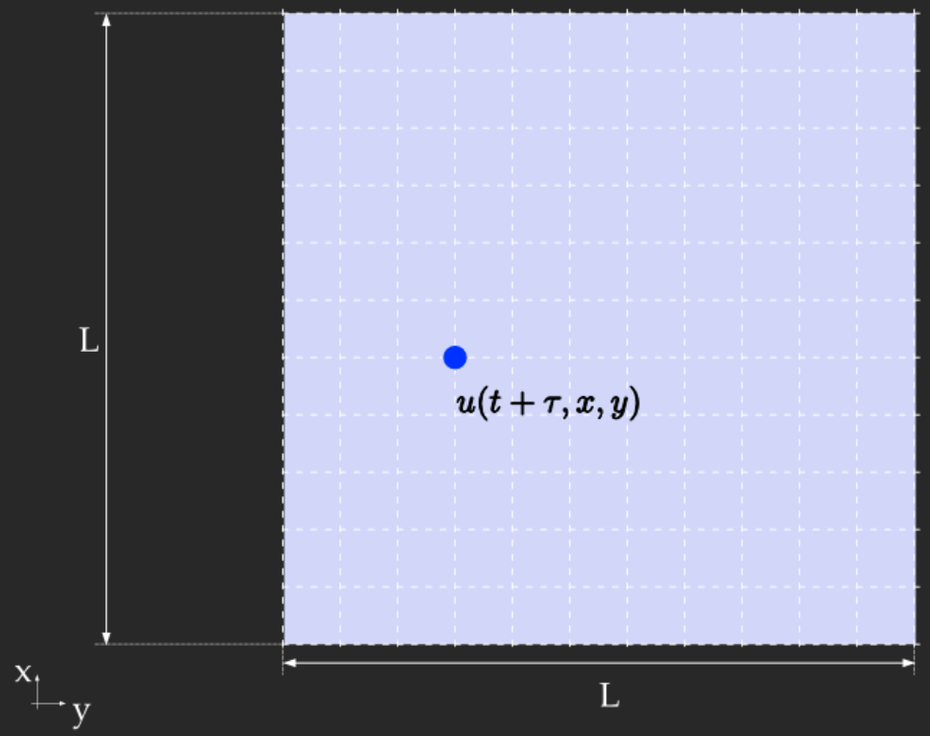
# Domain decomposition of PDEs

$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$



# Domain decomposition of PDEs

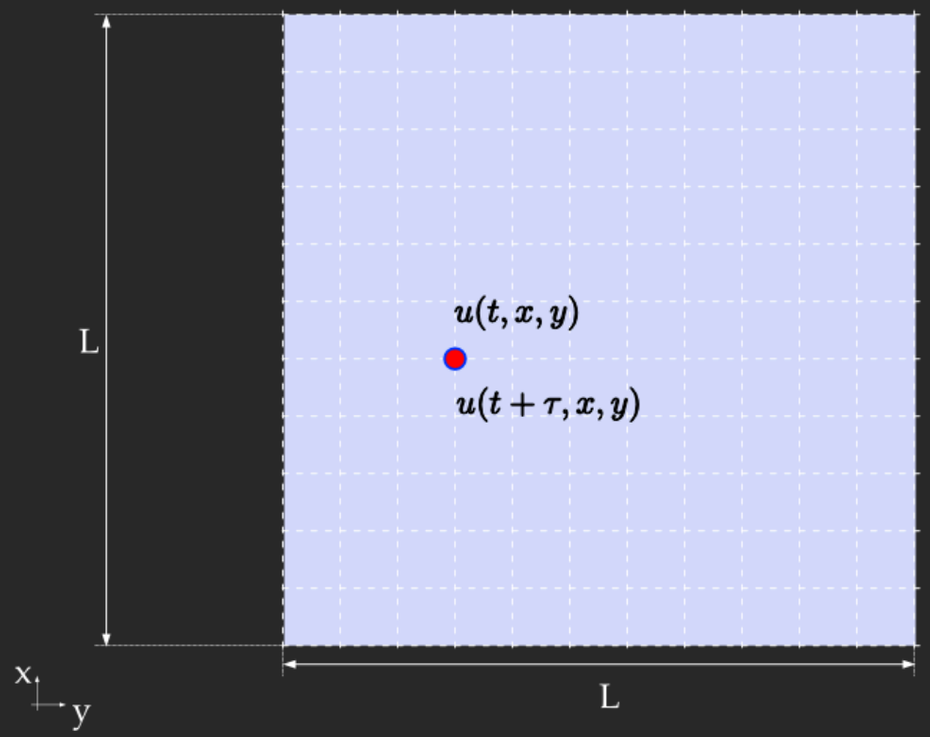
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- Note the dependencies of each grid-point

# Domain decomposition of PDEs

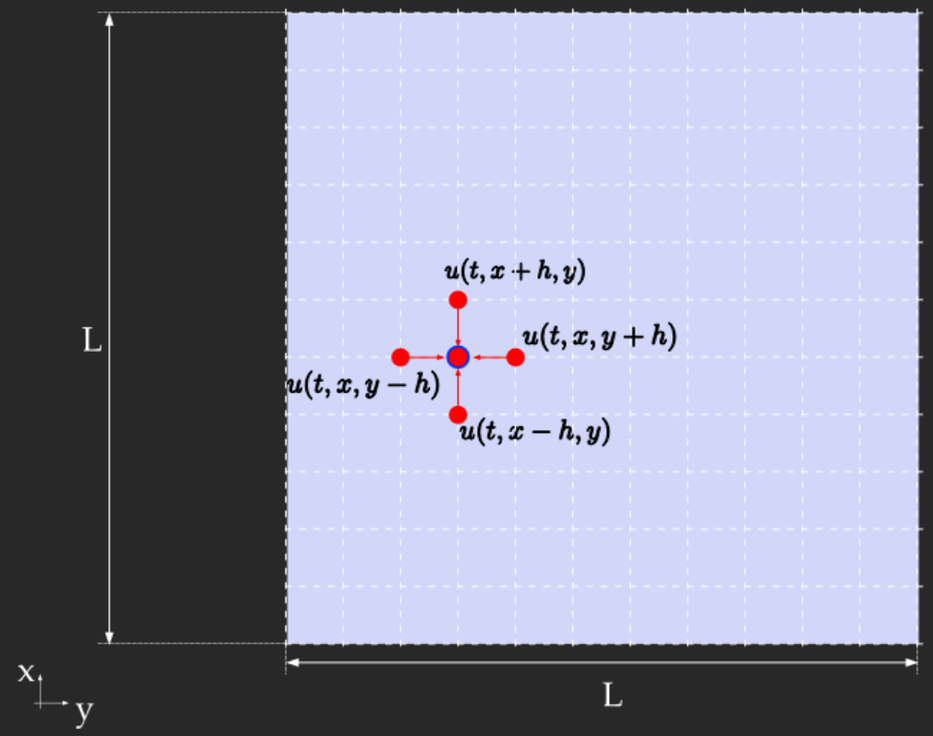
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- Each grid-point $(x, y)$, for $t + \tau$, requires $(x, y)$ at $t$

# Domain decomposition of PDEs

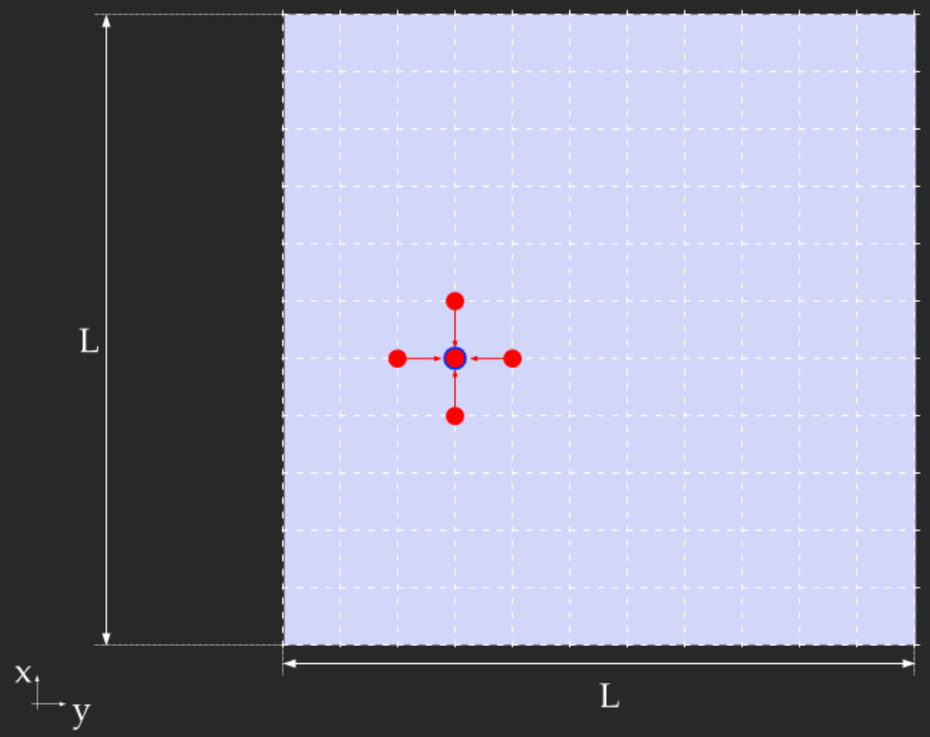
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- ...as well as $(x, y)$'s four nearest neighbors

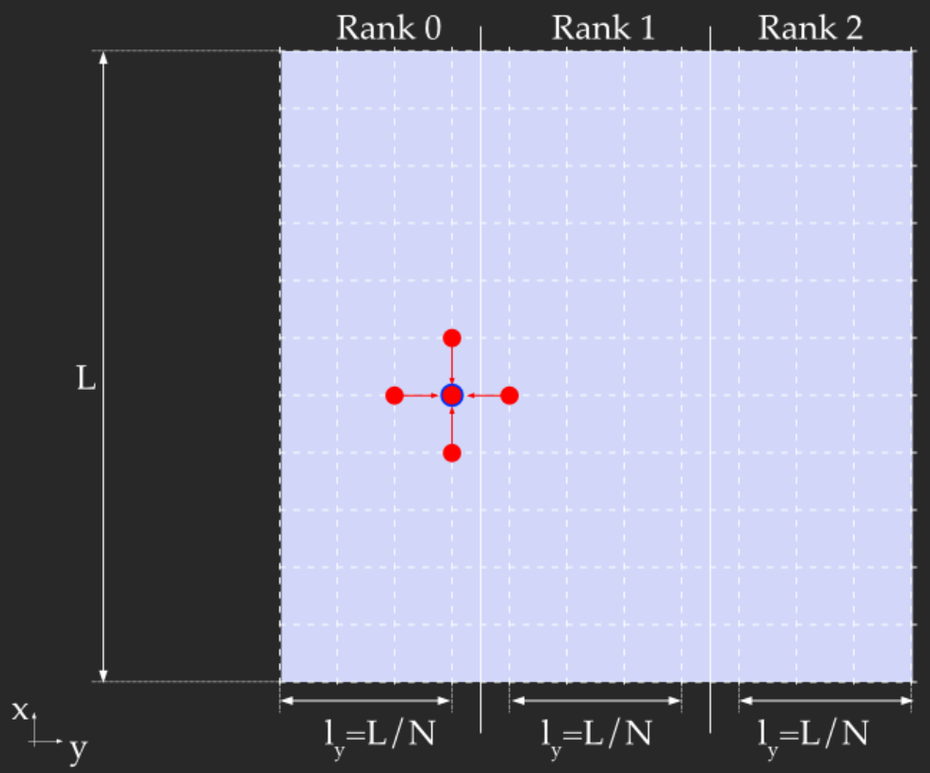
# Domain decomposition of PDEs

$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$



# Domain decomposition of PDEs

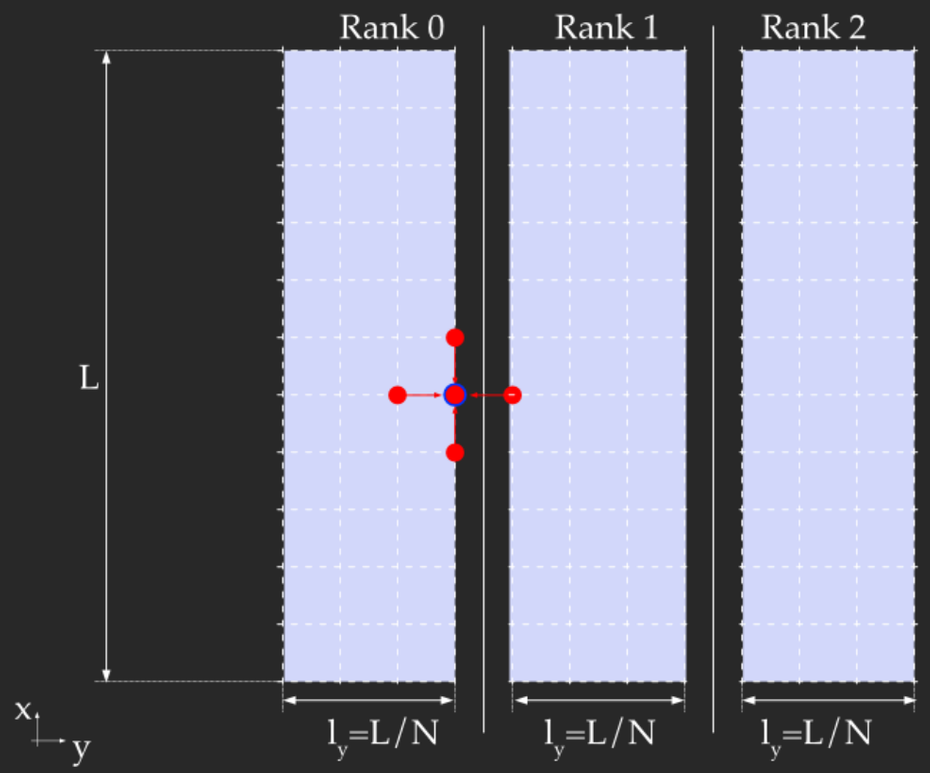
$$u(t+\tau,x,y) = u(t,x,y)(1-4\alpha) + \alpha H[u(t,x,y)],$$




- Assume we are partitioning over the $y$ direction

# Domain decomposition of PDEs

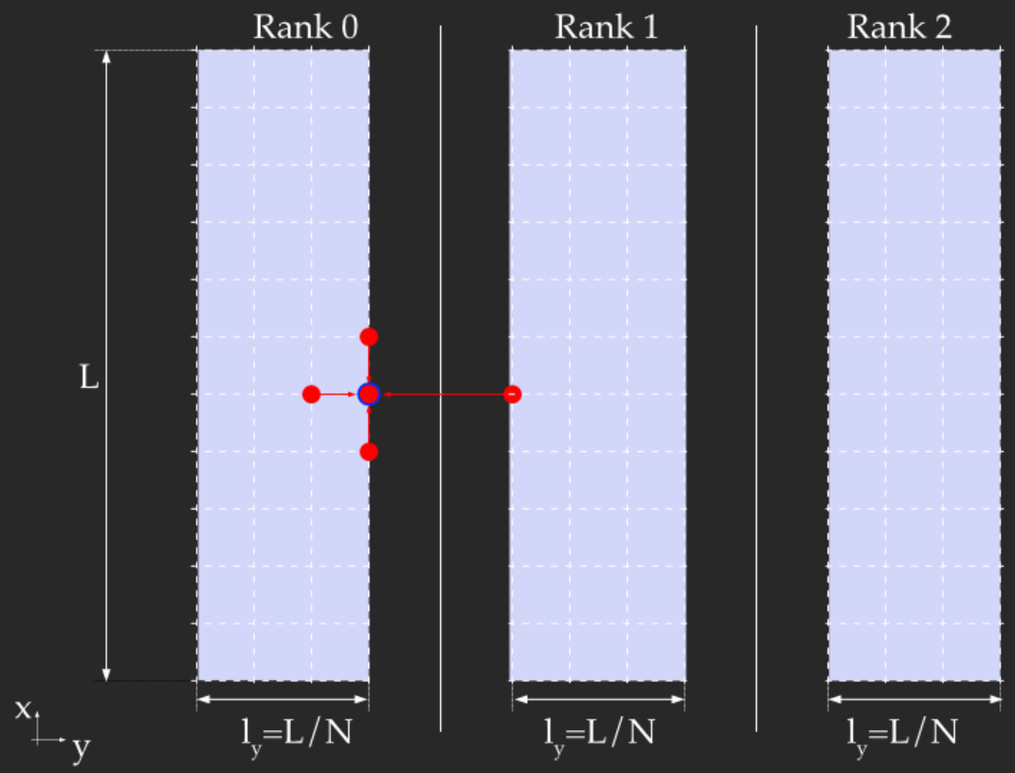
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- Assume we are partitioning over the $y$ direction

# Domain decomposition of PDEs

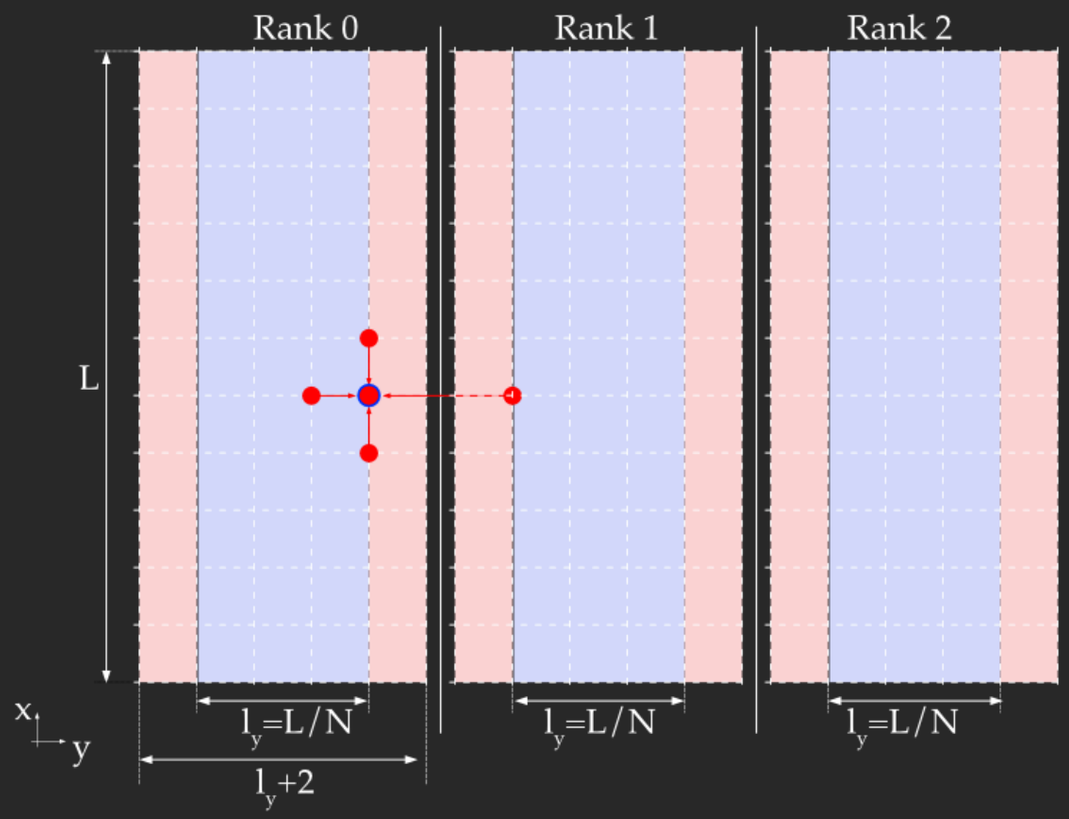
$$u(t+\tau,x,y) = u(t,x,y)(1-4\alpha) + \alpha H[u(t,x,y)],$$




- Assume we are partitioning over the $y$ direction
- For all processes, there are missing dependencies for $y = 0$ and $y = l_y - 1$

# Domain decomposition of PDEs

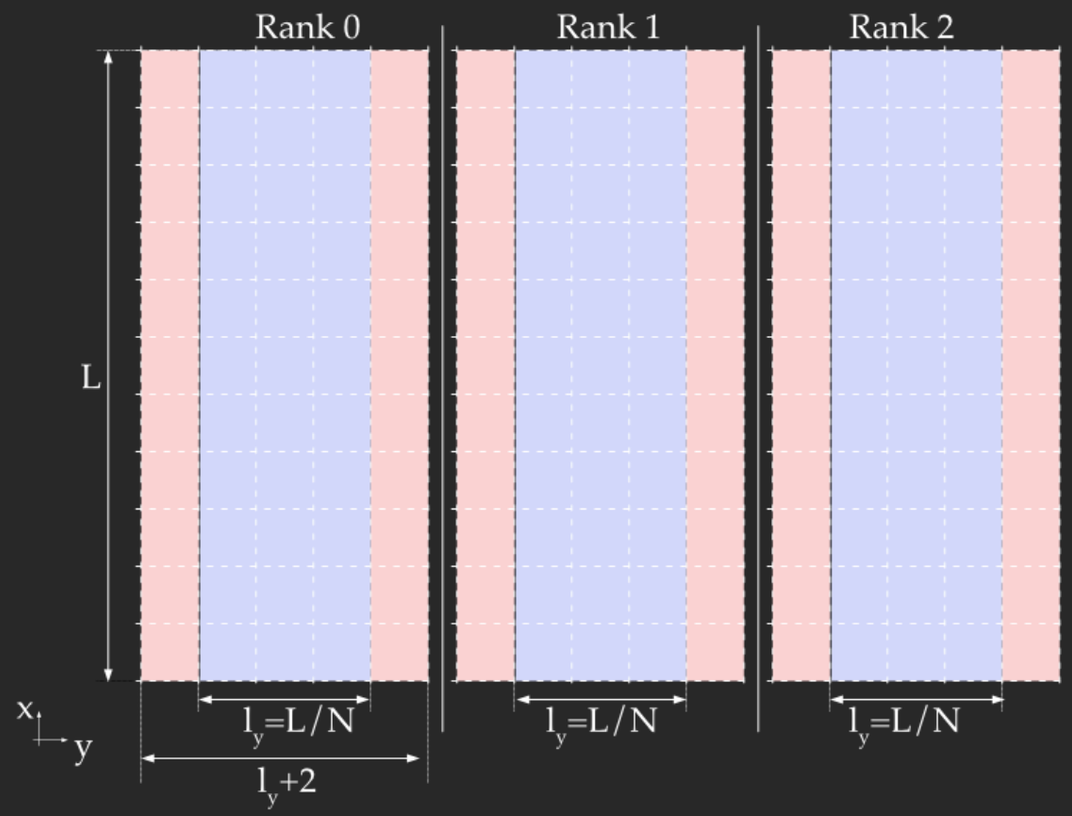
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- The solution is to allocate "halos". Extra space to store the neighboring rank's data

# Domain decomposition of PDEs

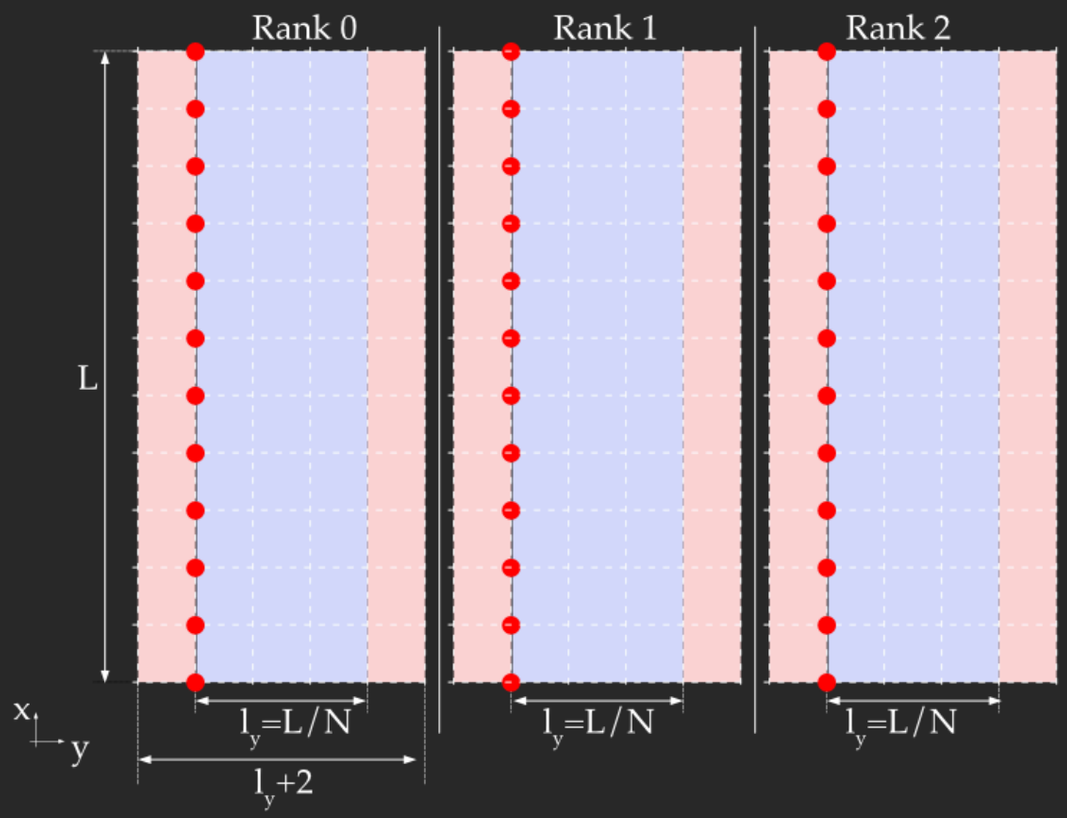
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- Communication proceeds in two steps

# Domain decomposition of PDEs

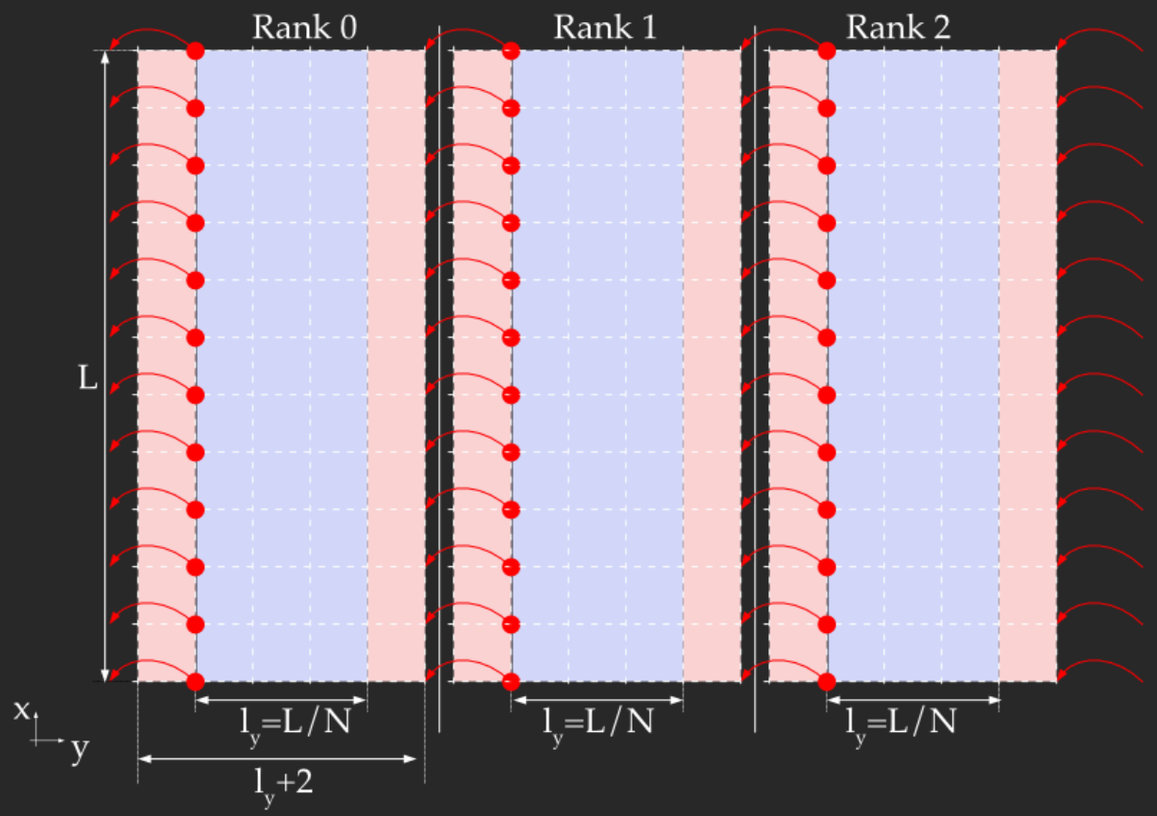
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- Communication proceeds in two steps

# Domain decomposition of PDEs

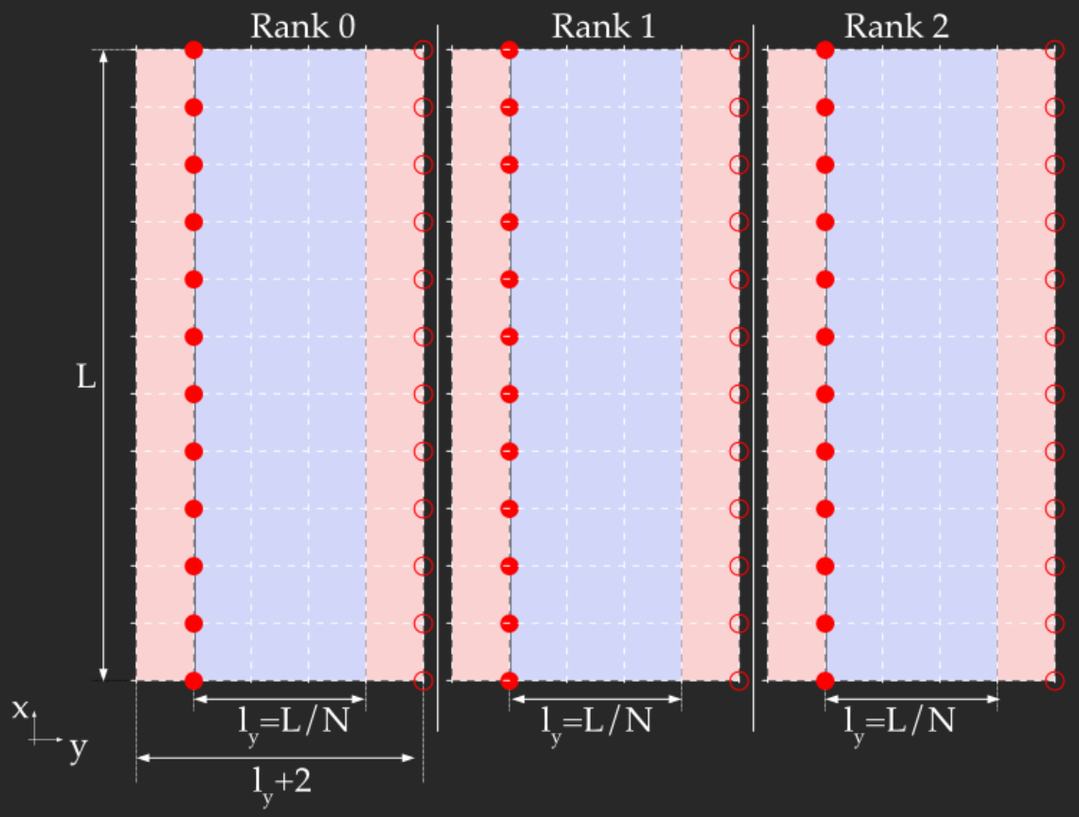
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- The "backwards boundaries" are communicated "backwards" and received from the "forwards" process

# Domain decomposition of PDEs

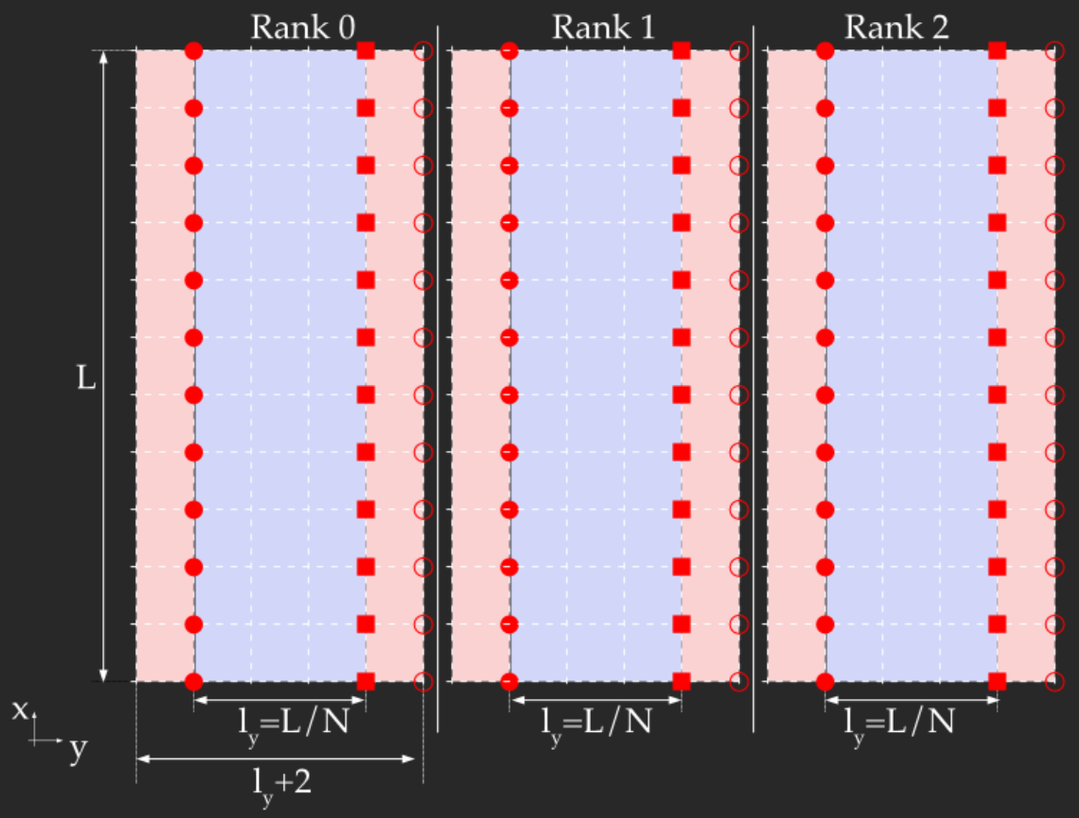
$$u(t+\tau,x,y) = u(t,x,y)(1-4\alpha) + \alpha H[u(t,x,y)],$$




- Now the halos include the (rank+1)'s boundaries; the neighbors for $y = l_y$ are available

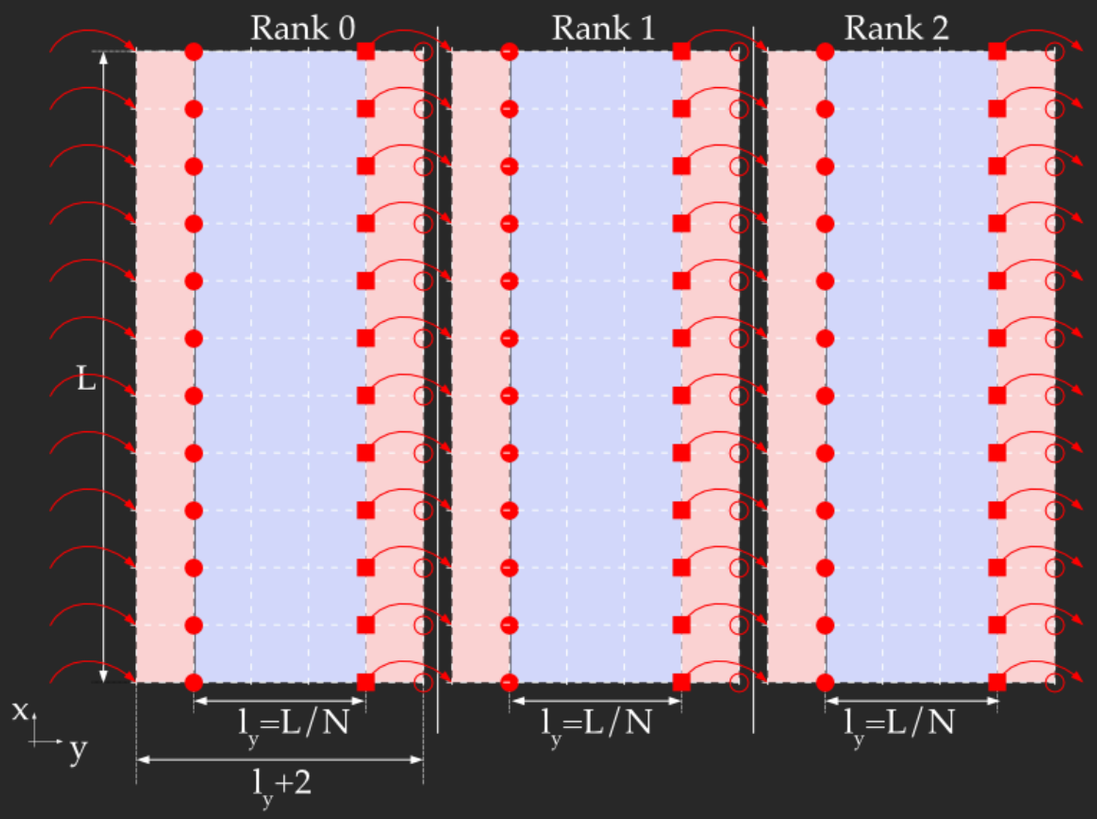
# Domain decomposition of PDEs

$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$



# Domain decomposition of PDEs

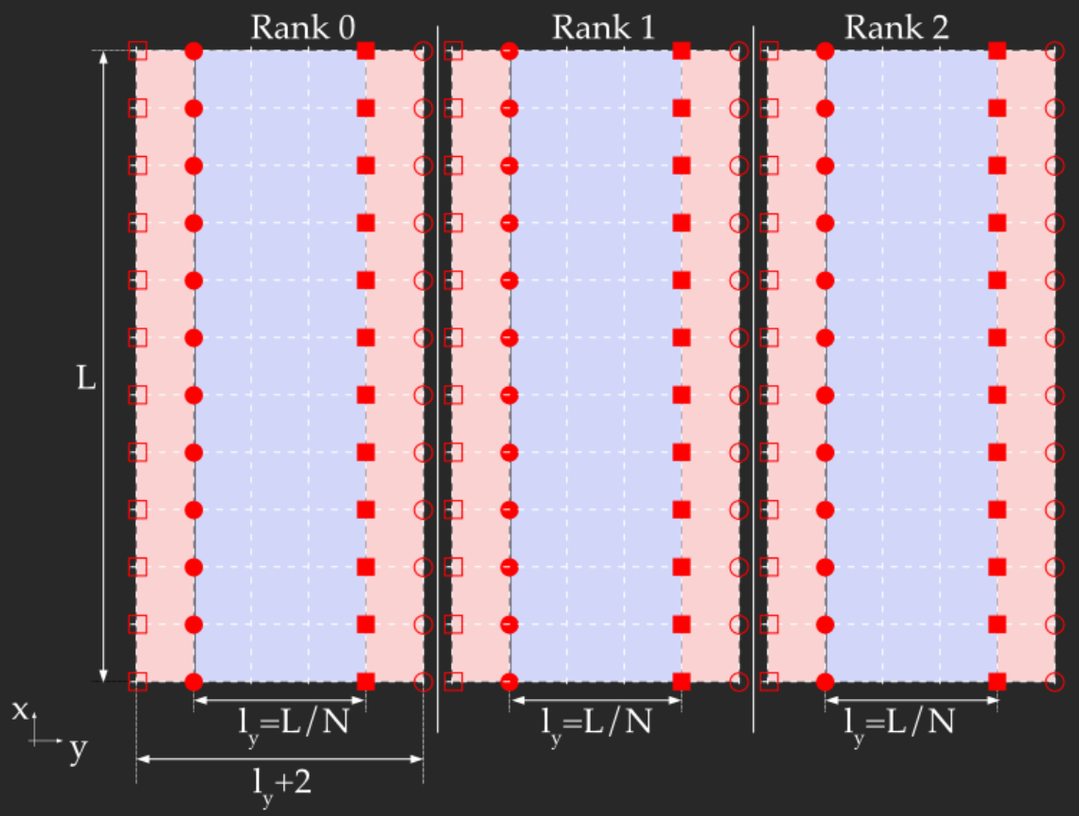
$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- The "forward boundaries" are communicated "forwards" and received from the "backwards" process

# Domain decomposition of PDEs

$$u(t+\tau, x, y) = u(t, x, y)(1 - 4\alpha) + \alpha H[u(t, x, y)],$$




- Now the halos include the (rank-1)'s boundaries; the neighbors for $y = -1$ are available

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

- Copy the directory:

  ```
  /onyx/data/sds406f24/l09/ex02/
  ```

  to your home directory.
- Included is a C file that includes MPI
- It is incomplete, and you must complete the missing sections
- Once done you can run it using the `heat.sh` Slurm script
- It should produce exactly the same result as the serial version

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

### 1st `TODO`

- Fix the `IDX()` macro
- Remember, the extent in `x` is the same
- The extent in `y` is now shorter. But also, there is space allocated before `y=0` and after `y=ly-1`

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

### 2nd `TODO`

- Fix the write-function
- We want each rank to write, in order, its part of the array, appending to a single file
- You need to ensure serialization of the rank writing

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

### 3rd `TODO`

- Fix the function that sets the boundary conditions
- Use the same boundary conditions as in the serial version
- You need to think about which rank holds which grid-points

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

### 4th `TODO`

- Fix the function that communicates the boundaries, filling in the "halos" of each process
- You need to do two communications
    - A Send to the next neighbor/receive from the previous neighbor
    - A Send to the previous neighbor/receive from the next neighbor

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

### 5th `TODO`

- Complete the update of the field `v` in function `update()`
- How will this be different from the serial version?

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

### 5th `TODO`

- Complete the update of the field `v` in function `update()`
- How will this be different from the serial version?

### When done

- Compile with `mpicc`
- Run with `heat.sh`
- The resulting output `v0.txt` should be *identical* to the serial version

# Domain decomposition for the heat equation

## MPI parallelization of the heat equation

### 5th `TODO`

- Complete the update of the field `v` in function `update()`
- How will this be different from the serial version?

### When done

- Compile with `mpicc`
- Run with `heat.sh`
- The resulting output `v0.txt` should be *identical* to the serial version
- E.g.

```
[ikoutsou@front02 ex02]$ diff v0.txt ../ex01/v0.txt
[ikoutsou@front02 ex02]$
```

No output means no differences found

## Strong scaling of the MPI-parallelized heat equation solution

*For next lesson*:

$\Rightarrow$ Plot against inverse time-to-solution rather than speed-up

$\Rightarrow$ Include in the plot the scalar version of `ex01` at $n_{\mathrm{proc}} = 1$