

# Introduction to Project 3

## Nonlinear PDE

**HPC Software Atelier**

**Prof. Dr. O. Schenk**

**Assistants: Dr. J. Kardoš, T. Holt, M. Lechekhab, Z. Panthakkalakath**

# Overview

- **Overview of Project 3**
- **Short review of the code**
- **Running the code and visualizing the output**

# The HPC PDE Application

The code solves a **reaction diffusion** equation known as **Fischer's Equation**

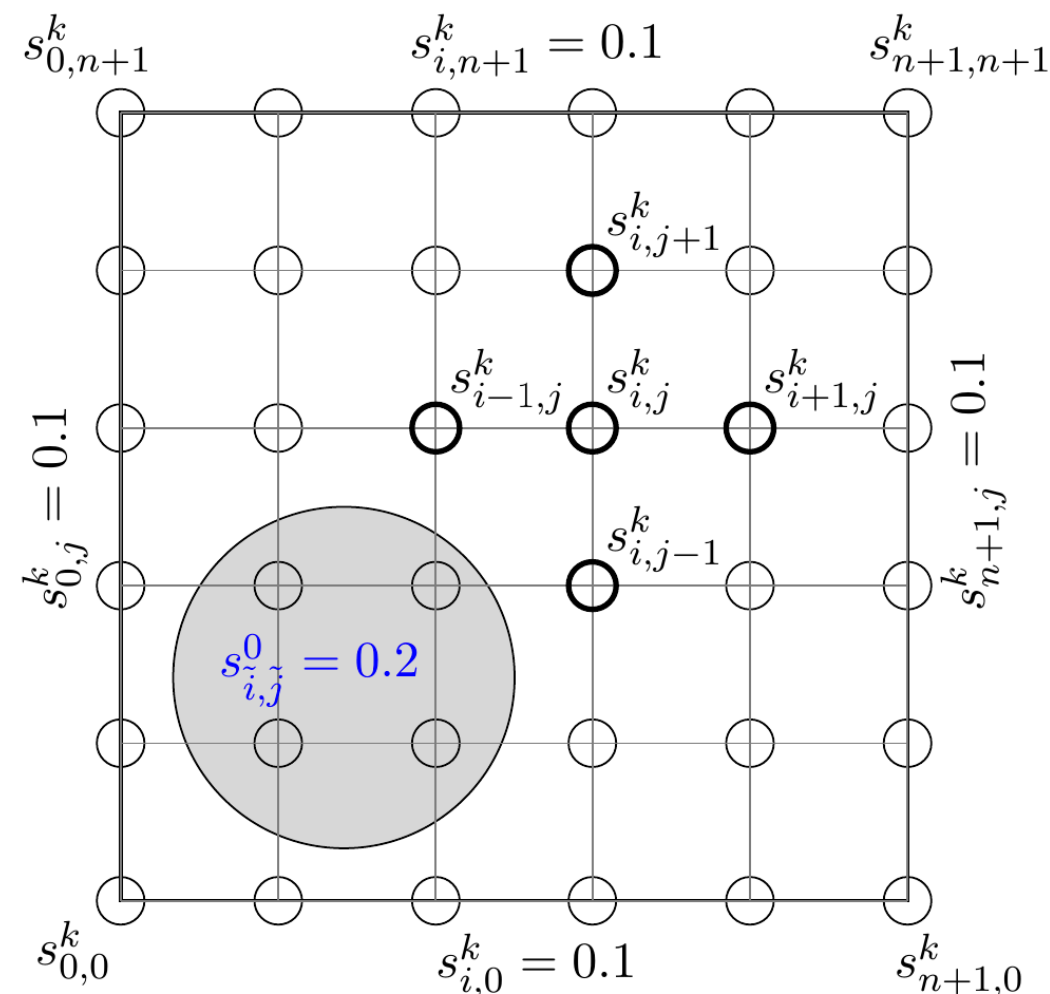
$$\frac{\partial s}{\partial t} = D\left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right) + Rs(1 - s)$$

**Used to simulate traveling waves and simple population dynamics**

- The species **s** diffuses
- And the population grows to a maximum of **s=1**

# Initial and boundary conditions

- set the domain to be the unit square, i.e.  $\Omega = (0,1)^2$
- fixed boundary values with  $s(x) = 0.1$  for  $x \in \partial\Omega$
- Initial values  $s^{init}$  are set to 0.1 except for a circular region with initial values of 0.2 in the lower left corner



# Numerical Solution

The rectangular domain is discretized with a grid of dimension  $(n+2) \times (n+2)$  points

- A **second order finite difference** discretization gives the following approximation for the spatial derivatives

$$\left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right)_{i,j} \approx \frac{1}{\Delta x^2}(-4s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1})$$

- A **first order finite difference** discretization is used for the approximation of the temporal derivative

$$\left(\frac{\partial s}{\partial t}\right)_{i,j}^k \approx \frac{1}{\Delta t}(s_{i,j}^k - s_{i,j}^{k-1}),$$

# Numerical Solution

- Putting these together one obtains

$$\frac{1}{\Delta t}(s_{i,j}^k - s_{i,j}^{k-1}) = \frac{D}{\Delta x^2}(-4s_{i,j}^k + s_{i-1,j}^k + s_{i+1,j}^k + s_{i,j-1}^k + s_{i,j+1}^k) + Rs_{i,j}^k(1 - s_{i,j}^k)$$

- Reformulate problem as

$$f_{i,j}^k := [-(4 + \alpha)s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{i,j}(1 - s_{i,j})]^k + \alpha s_{i,j}^{k-1} = 0$$

$$\alpha := \Delta x^2 / (D\Delta t)$$

$$\beta := R\Delta x^2 / D$$

# Numerical Solution

- **One nonlinear equation for each grid point**

together they form a system of  $N=n*n$  equations

Solve with Newton's method

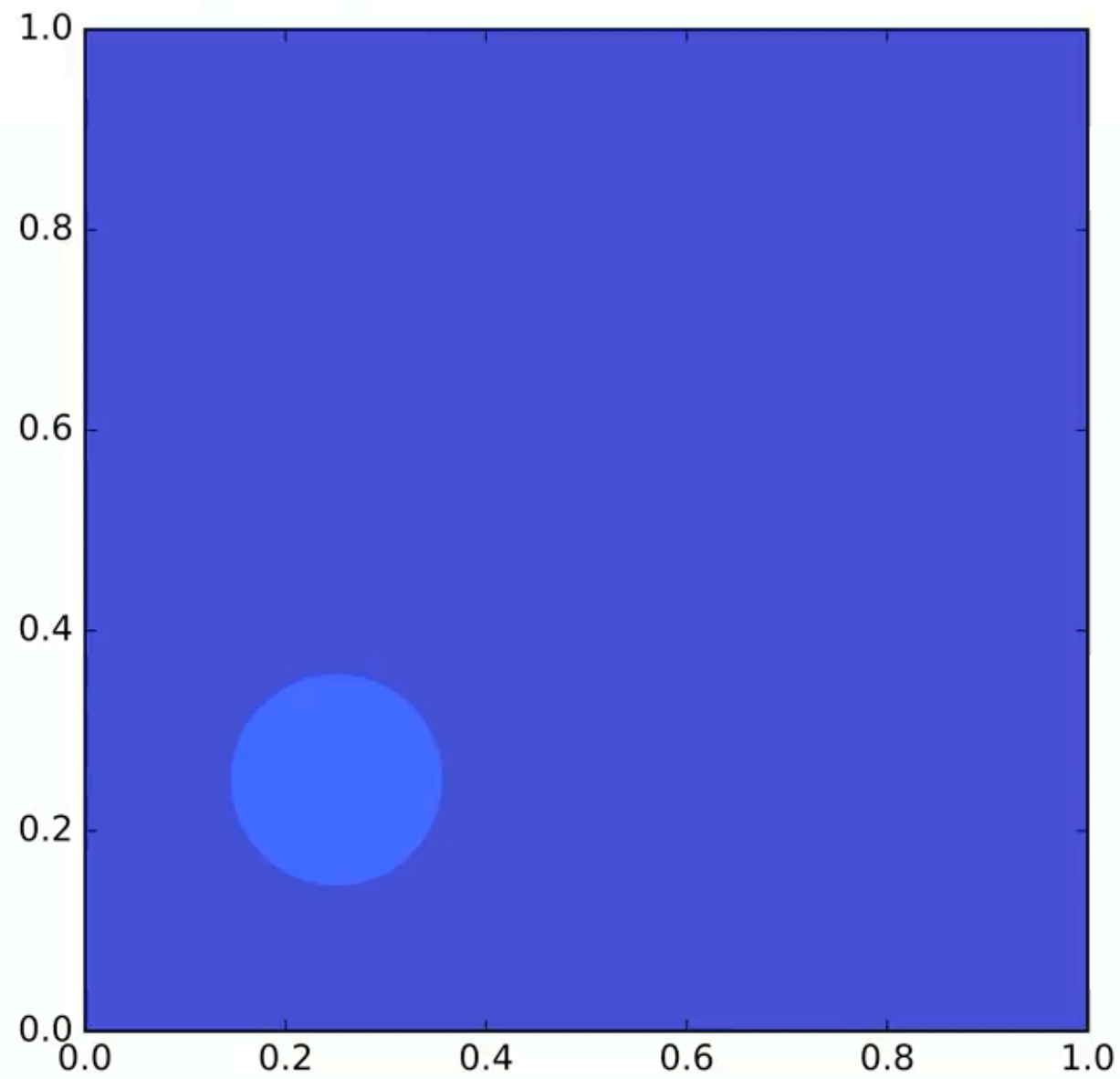
$$y^{l+1} = y^l - [J_f(y^l)]^{-1} f(y^l),$$
$$y^0 := s^{k-1}$$

- **Each iteration of Newton's method has to solve a linear system**

Solve with matrix-free Conjugate Gradient solver

$$[J_f(y^l)] \delta y^{l+1} = f(y^l)$$
$$y^{l+1} = y^l - \delta y^{l+1}$$

# Time Evolution of the Solution






# Numerical Solution

- **Most of the code is already implemented**
- **The focus is on the parallelization of the code using OpenMP**
- **So let's look a little closer at each part of the code**

# Code Walkthrough

**There are three modules of interest**

- [main.cpp](#): initialization and main time stepping loop
- [linalg.cpp](#) : the BLAS level 1 (vector-vector) kernels and conjugate gradient solver
- [operators.cpp](#) : the stencil operator for the finite difference discretization



the vector-vector kernels and diffusion operator are the only kernels that have to be parallelized

# Linear algebra: linalg.cpp

**This file defines simple kernels for operating on 1D vectors, including**

- dot product :  $x \cdot y$  : `hpc_dot()`

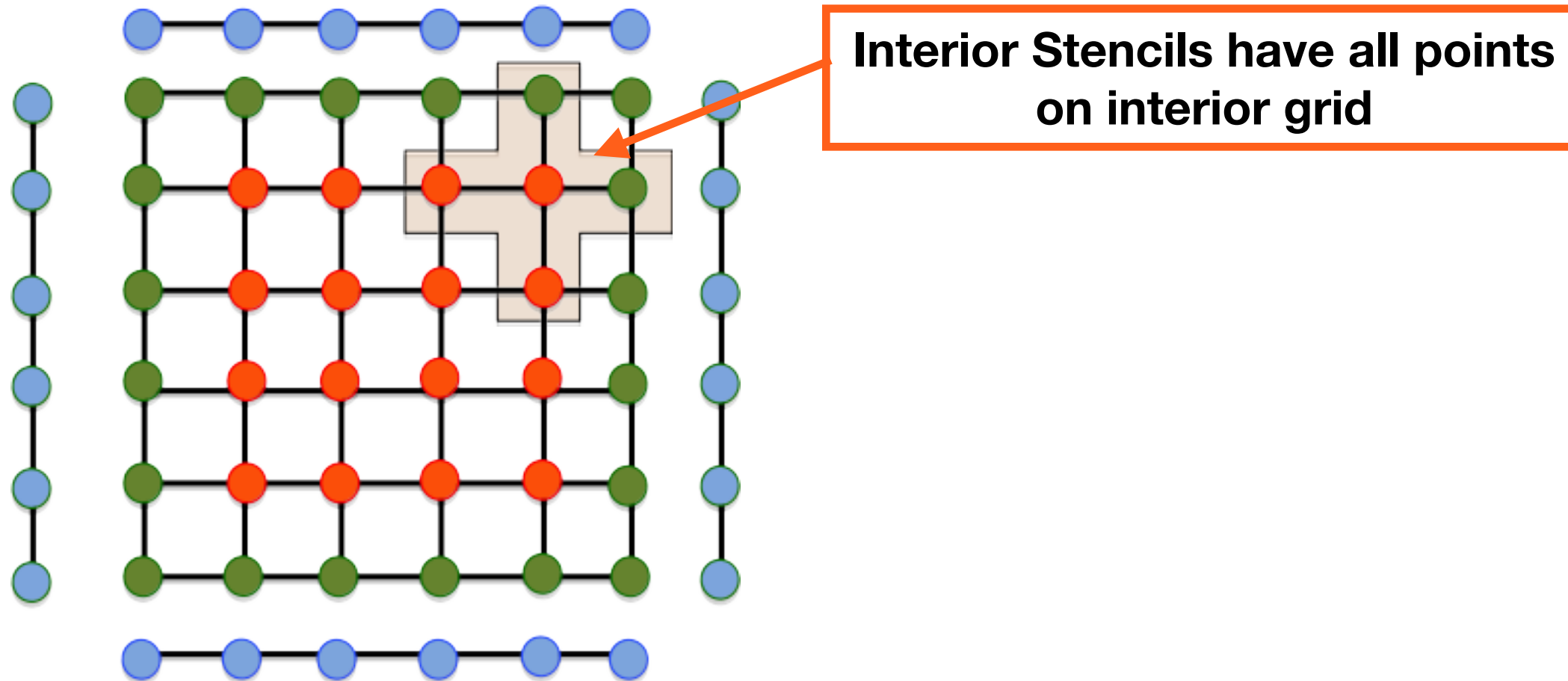
linear combination :  $z = \alpha x + \beta y$  where  $\alpha, \beta \in \mathbb{R}$ : `hpc_lcomb()`

- The kernels of interest start with `hpc_XXXXX()`

hpc == HPC Lab for CSE

For each parallelization approach that we will see (OpenMP, and later on MPI), each of these kernels will have to be considered.

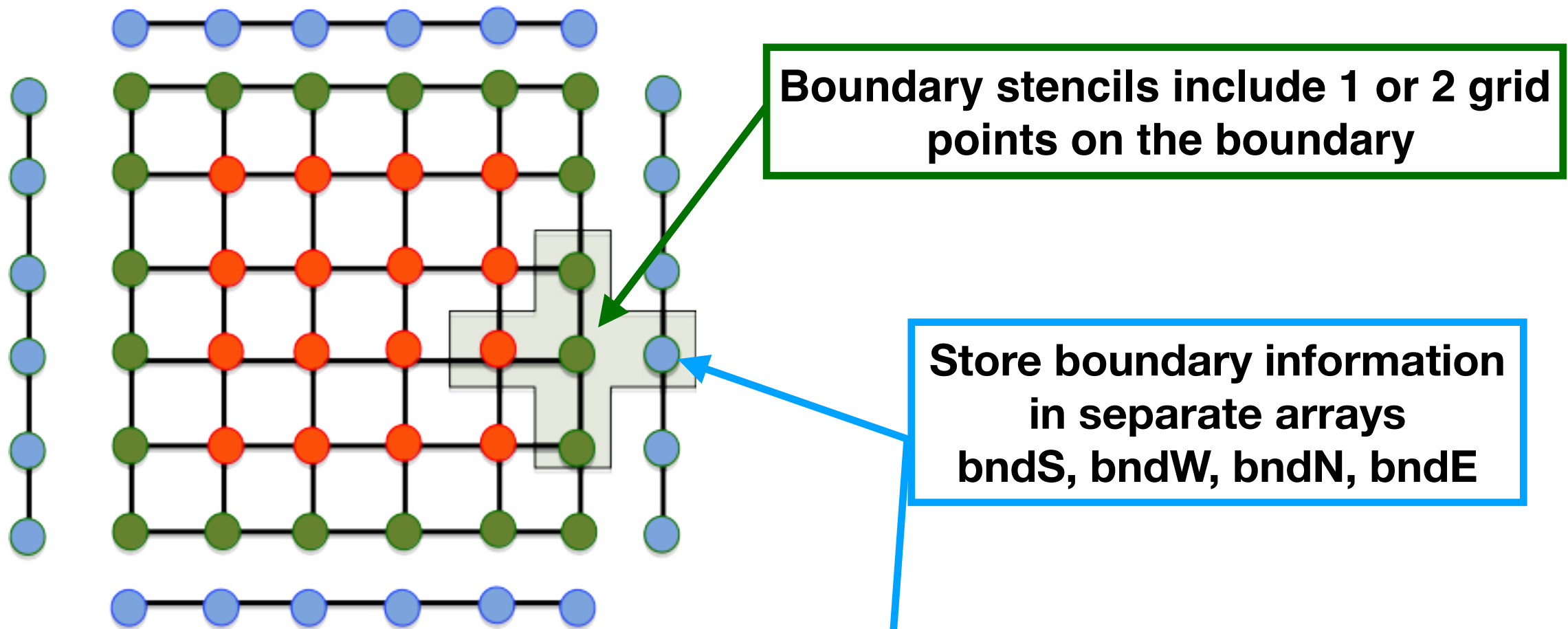
# Boundary-free stencil (interior stencil)



```
for j = 1 : n
  for i = 1 : n
    
$$f_{i,j}^k = [-(4 + \alpha)s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{i,j}(1 - s_{i,j})]^k + \alpha s_{i,j}^{k-1}$$

  end
end
```

# Boundary stencil



```

for j = 1 : n
  for i = n + 1
     $f_{i,j}^k = [ - (4 + \alpha) s_{i,j} + s_{i-1,j} + \mathbf{bndE}_i + s_{i,j-1} + s_{i,j+1} + \beta s_{i,j} (1 - s_{i,j}) ]^k + \alpha s_{i,j}^{k-1}$ 
  end
end
    
```

# Testing the code

- **Get the code from iCorsi or Github**
- **Compile and run**

```
[user@login]$ salloc  
[user@icsnodeXX]$ export OMP_NUM_THREADS=1  
[user@icsnodeXX]$ ./main 128 100 0.005
```

# Output

```
=====
                        Welcome to mini-stencil!
version :: C++ serial
mesh    :: 128 * 128 dx = 0.00787402
time    :: 100 time steps from 0 .. 0.005
=====
step 1 required 4 iterations for residual 7.21951e-07
step 2 required 4 iterations for residual 7.9975e-07
...
step 99 required 12 iterations for residual 9.36586e-07
step 100 required 12 iterations for residual 9.44772e-07
-----
simulation took 1.58408 seconds
2853 conjugate gradient iterations, at rate 6341.35 iters/second
492 newton iterations
-----
Goodbye!
```

**Thank you for your attention and have fun with the Project!**